Vrije Universiteit Amsterdam

Parallel Programming Practical
MPI: Naïve N-Body Simulation

Christopher Esterhuyse
2553295 – cee600
September 12, 2019

1 Introduction & Background

From the simulation of a large number of individual 'bodies' in a simulation space, each interacting with the others, complex and interesting behavior arises. Such simulations can come in the form of tiny elementary particles or enormous stars in a galaxy. Complexity of such simulations increases quadratically with the number of bodies, necessitating enormous work in order to observe emergent properties at the most vast of scales. Here, 'naïve' refers to the lack of lossy optimization techniques, such as those used in the Barnes-Hut algorithm.

MPI is a popular industry-standard library for lifting traditional, sequential programming languages to distributed ones by enriching them with message-passing primitives. In our case, we use the variant available for the C programming language. MPI coordinates otherwise-isolated processes by supplying each with it's own view of the problem: processes are provided unique 'rank' identifiers along with awareness and mechanisms with which they can communicate with their peers. Each process runs the same program. As such, the programmer implements the logic of all participants *simultaneously* by differentiating their behaviour based on the runtime-evaluation of its 'rank'. As a result, a coordinated solution effort emerges at run-time.

This report details the design, development and evaluation of an MPI-C implementation of a N-body simulation of simplified celestial bodies, gravitationally-attracted toward one another in a bounded 2-dimensional space. The Vrije Universiteir's in-house computing cluster 'DAS-4' provides the infrastructure necessary to support a parallel and distributed solver. We focus on maximizing the effective utilization of these resources to achieve maximal speedup.

2 Design & Implementation

This section details the properties of the naïve N-body simulation problem. A set of promising approaching to achieving speedup are explained. Finally, the set of implementations relevant to experiments in section 3 are defined.

2.1 Foundations

We approach the problem given a reference implementation. Structurally, it consists of an outermost loop over simulation time steps. For each, the overwhelming work (with complexity $O(N^2)$) consists of a double-for loop, visiting all pairs of bodies (a,b). For each pair, the instantaneous force of a on b (written $\overrightarrow{ab}$) is computed, aggregated into the 'force' field of b. Once all forces have been computed in this way, all bodies' instantaneous velocities for the time step are computed from the force. Likewise, their new positions are computed. The reference implementation uses double-buffering, reading from one and writing new positions to the other. Between steps, the pointers are effectively switched. All-in all, the complexity of the work is bounded approximately by $O(TN^2)$ with T time steps and N bodies.

2.2 Design Choices

This section explores the properties of the naïve N-body implementation and discusses various design choices that arise from parallelizing it. Below, we define a set C of design choices that are applicable to solvers of this problem.

C_{opp} **Opposing Forces**
For two bodies a and b, the instantaneous force $\overrightarrow{ab}$ is equal in magnitude but opposite in direction to $\overrightarrow{ba}$. For each such $\overrightarrow{ab}$ computed, $\overrightarrow{ba}$ can thus be inferred immediately. Concretely, whilst traversing the pairs of bodies in a double-for-loop, indices can be computed as to avoid redundant pair visits, safe in the knowledge that they will be computed when the reciprocal is traversed. This halves the number of total force-computations. This optimization is already present in the reference implementation.

C_{bod} **Parallelizing over Bodies**
The field-data of a celestial body is completely independent of the field data of all other bodies within the same time step. For some arbitrary body a and time step t, position on a at t is a function of its position at $t-1$ and the force on a at $t-1$.[1] Likewise, velocity is a function of force at t and the body's velocity at $t-1$. However, force on a at t is a function of *all* body-positions at $t-1$. As the body data can be arbitrarily-initialized, there is no meaningful way of representing body field data at t without complete knowledge of the state at $t-1$. Any concurrent

[1] There is of course an exception for the very first time step 0, in which all body-field-data comes from a seeded pseudo-random number generator.

computation which does not share a memory space must either re-compute the entirety of the state each time step (removing the possibility of any speedup) or *fully communicate* the needed body data (positions) at $t-1$. Fortunately, as the descriptions above suggest, all field-data aside from force is independent of other bodies. Additionally, the computation of the force on some body a requires only reading operations of the body data from $t-1$, meaning that computations of body data in a new time step is side-effect free and can be trivially parallelized.

C_{arr} **Arrays of Fields**
The reference implementation represents the state of the simulation as an array of structures, where each structure represents a single celestial body, containing its own field-data. As explored for C_{bod}, position data of bodies must be communicated for every time step above 1. Practically, writing and reading data that occurs as contiguous sequences in memory is easier, as it requires more uniform indexing and is more cache-coherent. As such, an implementation can instead re-structure the memory layout to be represented as a single 'world' structure of arrays. Each array contains an element per celestial body, representing a class of field datum.

C_{thr} **Thread Pool**
As explored in C_{bod}, parallelism within a time step promises speedup. However, approaching concurrency with only processes (which do not share a memory space) necessitates the dissemination of *messages*. Depending on how messages are communicated between these processes, the message complexity is somewhere in the range of $O(Nlog(N))$ to $O(N^2)$ where N is the number of processes. Threads do not share this weakness, and aside from the overhead required to explicitly synchronize their control flow (as well new cache behaviour with respect to a sequential implementation) threads may work without exchanging messages, essentially sharing information *implicitly* by agreeing beforehand on some non-colliding work partition (for example, each thread may compute $\overrightarrow{ab}$ for all a, but for all b in some owned *subset* of the bodies). Race conditions can be avoided by restricting the utilization of the thread pool to only perform tasks statically-guaranteed not to incur race conditions. This has implications for C_{eag} where multiple tasks may be created to write to the same bodies. However, this must be avoided by placement of `wait()` calls to ensure correctness. As such, all implementations can perfectly predict the number of tasks they would ideally run in parallel using the thread pool. The pool can be initialized once, and each time tasks spawned such that they map one-to-one with the number of available worker threads. Additionally, this means that the degenerative case of requiring 1 worker can revert to a (per process) sequential program in order to avoid overhead of managing a worker thread.

C_{eag} **Eager Chunk Computation**
C_{thr} touches on the message complexity of communication. MPI's collectives offer an efficient means of disseminating messages in its *collectives*. As time is often a factor, an MPI broadcast will rely on *rounds* of messages, where each round the number of processes that have datum x will double (initially, only the broadcast

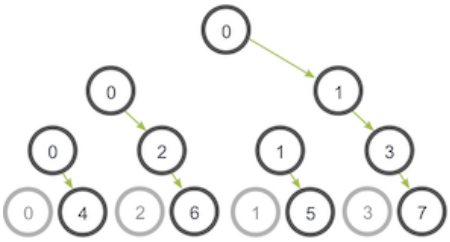


Figure 2.1: MPI Broadcast tree. In round 0, node 0 sends the message to 1. In round 1, nodes 0,1 send the message to 2,3 respectively. Each round, double the number of nodes are sending the message, until all nodes have received it. Image source: `http://mpitutorial.com`.

root has x). Figure 2.1 illustrates such a broadcast tree. While all processes participate in the collective, it is clear that they do not participate to the same *extent*. Processes may also finish their work before others when the latency in a message 'on the wire' is considered. Here we consider the possibility of overlapping the computation of time step t with the arrival of body-position data from time step $t-1$. The force on some body a can be represented (with B being all body-data in time step $t-1$):

$$\sum^{b \in B} \overrightarrow{ba}$$

This summation can be decomposed, allowing $\overrightarrow{ba}$ to be computed *partially*, subject to the incremental availability of B. Concretely, this property can be exploited by, each time step, all processes initiating asynchronous *broadcasts* of their own chunks of freshly-computed body data from $t-1$. Each broadcast (`MPI_Ibcastv`) can be distinctly registered to a distinct `MPI_Request` which can be polled for completion. The process then begins to eagerly poll for completed requests, computing changes to $\overrightarrow{ba}$ for all a in their own chunk until all chunks have arrived (meaning all b's have been considered). The utility of this approach hinges on the assumption that the added overhead of this eager polling and additionally-fragmented traversal of the body-space incurs *less overhead* than the variability in collective-completion times allow for latency-hiding.

2.3 Implementations

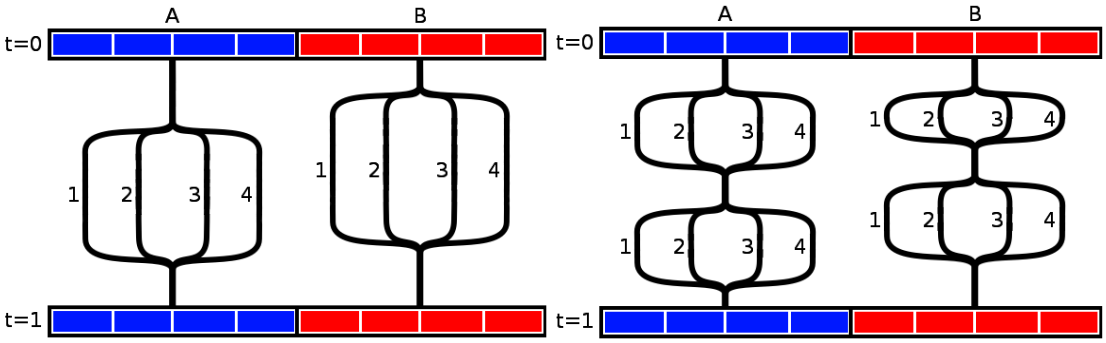
The design choices in section 2.2 suggest a possible *solution space* for the naïve N-body problem. This space is represented by a selection of just three implementations. Here these implementations are defined with respect to their mappings to the design choices.

seq This is the sequential reference implementation. As there is no parallelism, it leverages only C_{opp}. While C_{arr} is applicable also, its use is in this case not necessary owing to the lack of message passing.

par This implementation represents the culmination of all the design choices, aiming for a balance between maximizing concurrency and simplicity. Concretely, it applies design choices $\{C_{arr}, C_{bod}, C_{thr}\}$. C_{opp} is not applied, as with a non-trivial number of processes for some $\overrightarrow{ab}$ where b is being updated by some process p, there is high likelihood that a is assigned to be updated by a process other than p.

This implementation assumes that there are *not* significant benefits in splitting the chunk-communication as described for C_{eag}. It generates precisely w tasks per time step per process (where w is the number of worker threads), and performing its process-communication each time step as a synchronous `MPI_Allgatherv` call. Velocity and new position data are computed sequentially, as their $O(N)$ are soon dwarfed by the force-computation complexity in the case of realistic problems.

eag This implementation overlaps much with **par**, but instead assumes that there *are* worthwhile gains to be made from latency hiding. Concretely, this implementation applies $\{C_{arr}, C_{bod}, C_{thr}, C_{eag}\}$. Each time-step, p collective `MPI_Ibcast` calls are initiated in an asynchronous, non-blocking fashion, where p is the number of processes. The main thread then begins eagerly polling (busy-waiting) the set of p `MPI_Request` handles. The moment a handle signifies that a chunk of data has arrived (the position-data from time step $t-1$ has arrived in-place), w tasks are spawned to traverse the fresh chunk and mutate the computed forces on its own chunk's bodies. Once all chunks have arrived, the final force values are correct and the program proceeds as **par** does. The difference in how **eag** parallelizes work compared to **par** is illustrated in figure 2.2.



(a) par **(b) eag**

Figure 2.2: Illustration of parallelism to generate the body data for time 1, given the data for time 0 during a run of **par** or **eag** with 4 worker threads and 2 processes. In both cases, each process in $\{A, B\}$ is assigned a chunk of the entire body space, and for each, it splits that chunk further into parts, one per worker per process. **par** creates one task per worker. **eag** eagerly computes the data per chunk as it becomes ready, creating 4×2 tasks per process, but must serialize the processing of chunks to avoid data races.

3 Experiments

This section evaluates the implementations defined in section 2.3 according to their ability to provide speedup over the reference implementation, **seq**.

3.1 Experimental Setup

Evaluation of the implementations was conducted using the Distributed Asci Supercomputer (mark 4) cluster at the Vrije Universiteit Amsterdam. It offers dual quad-core worker nodes, each with 2.4 GHz processors, 24 GB of memory, networked using 1 Gbit/s bandwidth links. MPI jobs were launched to run *solo* on DAS-4 worker nodes. Run times were measured from the averages of re coded times per node, repeated twice (run times were found to be incredibly invariant on the DAS-4).

3.2 Results

First, figures 3.1 and 3.2 explore run time and speedup relative to **seq** in response to the number of processes. These tests made no use of multi-threading. Figure 3.1 illustrates that for problem instances that generate few coarse-grained parallel tasks (arising with a large proportion of bodies to time steps), both **par** and **eag** experience nearly identical linear speedup with respect to the number of processes, albeit uniformly diminished by a factor of 2 owing to these implementations not leveraging C_{opp} as **seq** does. For the case in figure 3.2, once again it takes two processes to match the performance of **seq**. However, speedup is noticeably dampened with increased processes. For both **par** and **eag**, the messaging overhead reduces speedup with more processes. This effect on **eag** is magnified, as it must distinguish multiple operations and delay computation *multiple times* per time step.

Figure 3.3 controls for the size of the problem[2], with the x-axis capturing the effect of changing the granularity of parallel steps for modest-sized problems. As before, both parallel implementations experience degraded linear speedup owing to the overhead incurred by the repeated collective messaging. Here, **eag** can be seen to degrade slightly faster with increasing number of time steps, experiencing a magnified overhead from having a parallel granularity increasing quadratically with the number of processes.

Finally, figure 3.4 explores the maximal potential of **par** and **eag** to achieve speedup, with the ideal circumstance of having the maximal number of bodies (10^4) and a modest number of time steps (100). In this case, both use multi-threading in addition to 16 processes. As before, owing to **seq** benefiting from C_{opp}, not using the thread pool results in speedup of approximately 8. Speedup is here not quite linear as both implementations must pay for the repeated spawning and waiting of worker threads. A new problem arises when the number of workers approaches the number of logical cores available to the DAS4 nodes: worker threads begin to share floating point units between them. Workers sharing these units cannot effectively parallelize their tasks, as their operations get serialized in

[2]It is not possible to always select a pair of $(timestep, bodynumber)$ to precisely result in 10^8 force computations. However, at these scales the variance is negligible, varying by less than 0.5% at most.

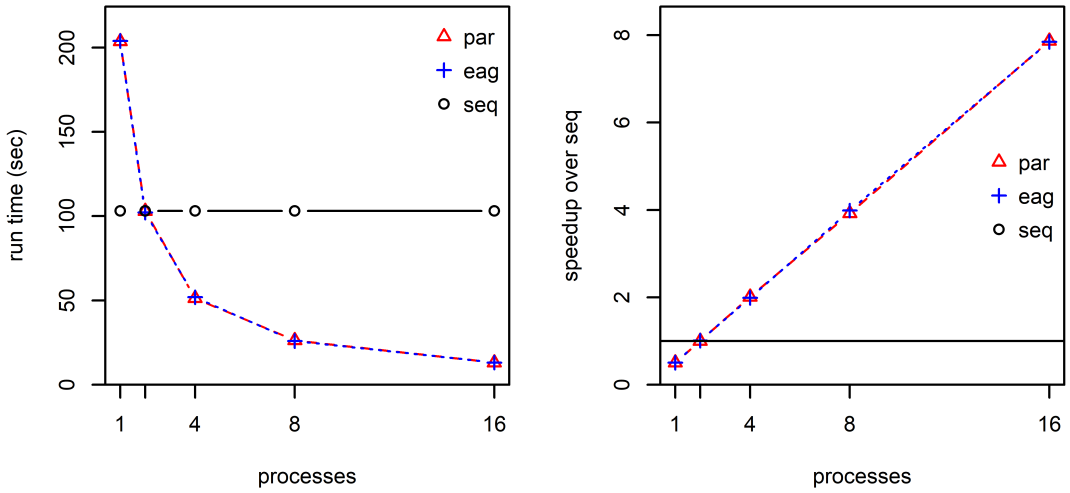


Figure 3.1: Run time and speedup of **par** and **eag** compared to **seq** for problem instances with few time steps (150) but many bodies (3000). The x-axis distinguishes results for runs using a different number of processors. **par** and **eag** are not multi-threading.

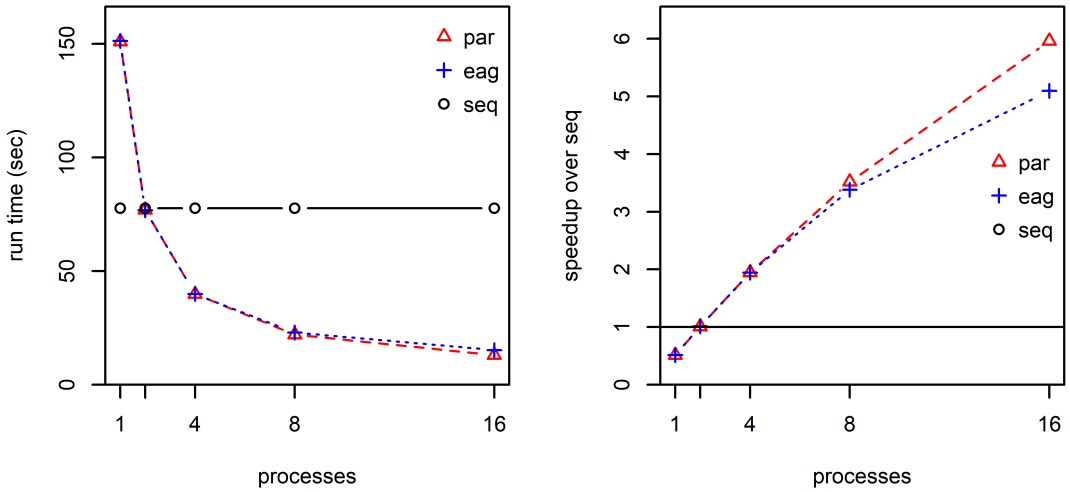


Figure 3.2: Run time and speedup of **par** and **eag** compared to **seq** for problem instances with many time steps (10^5) but few bodies (100). The x-axis distinguishes results for runs using a different number of processors. **par** and **eag** are not multi-threading.

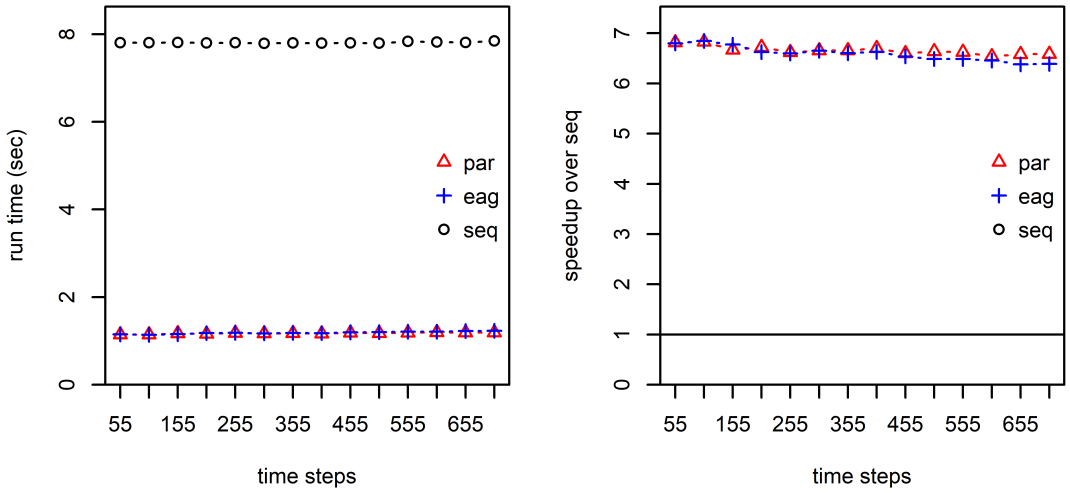


Figure 3.3: Run time and speedup of **par** and **eag** using 16 processes run on problems resulting in 10^8 force-computations overall. Problems with fewer time steps have, thus, greater b^2 where b represents the number of bodies. **seq** is provided for comparison with the same *input problem*, thus having half as many force-computations.

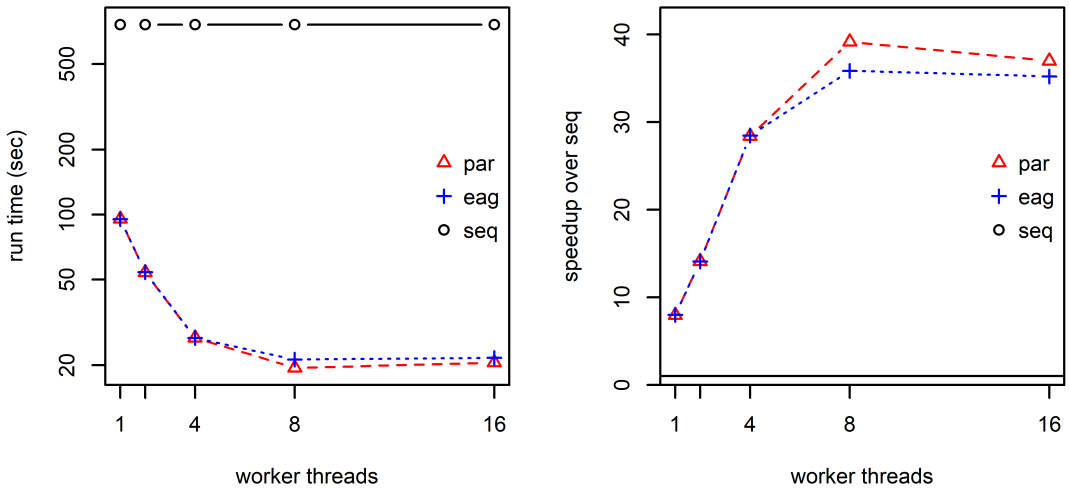


Figure 3.4: Run time and speedup of **par** and **eag** using 16 processes compared to **seq**. The problem instance has the maximal number of bodies (10^4) but only few (100) time steps. The x-axis distinguishes the results when using different numbers of worker threads.

lockstep. Once again, **eag** falls behind, with the overhead inherent to the management of the thread pool being magnified by the number of processes (and thus, the number of thread pool `wait()` operations per process per time step). Nevertheless, the use of a thread pool (particularly with 8 workers) allows **par** in one case to achieve the best speedup seen so far: 39,16.

4 Conclusion

Exploiting the data parallelism inherent to the naïve N-body problem facilitated the development of two solvers, **par** and **eag** that achieved nearly linear speedup with the number of available machine cores. As expected, equivalent-sized problems were more effectively sped up if work was distributed over fewer time steps.

Counter to expectations, multi-processing proved to be more effective than multi-threading. This can be explained as a combination of factors. Firstly, this implementation layers the multi-threading partitioning of work *within* the multi-processing steps, thus causing all costs of worker management to be paid repeatedly. Secondly, for problems of with many bodies, the size of messages between processes is dwarfed by the work per time step; this reduces the only source of overhead for multi-processing. Secondly, worker tasks could be assigned to threads that share floating point units, degenerating effective concurrency. As multi-threading is not the focus of this work, we omit further detail and conclude that multi-threading did not translate as neatly to speedup as expected.

Ultimately, **par** is best at reliably achieving speedup the computation of a naïve N-body simulation with realistic parameters, offering multi-threading as a cherry on top.