Performance Analysis of N-body Solvers Parallelized with OpenMP using Different Loop Partitioning Techniques:

Parallel and Distributed Systems

Kai Koo

Introduction

N-body simulations find applications in diverse fields such as astrophysics, molecular dynamics, and computational biology, where understanding the interactions among particles is crucial for gaining insights into complex physical phenomena. Hence, the accurate and efficient simulation of N-body systems has been a subject of significant interest and research. By leveraging parallelization, using tools such as OpenMP (OMP), one can optimize the performance of N-body solvers.

This paper presents a comprehensive review of four N-body solvers, all implemented in C with OpenMP, aiming to investigate the impact of different loop partitioning techniques on their runtime performance. The four solvers under examination are:

1. The basic N-body solver, which employs the default block scheduling technique provided by OpenMP.
2. The reduced N-body solver, which also employs the default block scheduling, but features additional optimization strategies.
3. The reduced N-body solver utilizing cyclic scheduling for a single loop, which aims to enhance load balancing among threads.
4. The reduced N-body solver with cyclic scheduling applied to all relevant loops, aiming to exploit further parallelism and load balancing opportunities.

Each solver was run on a 4-core USS system and their runtimes recorded. These results were used to give an indication of the effect the partitioning technique has on the overall efficiency of the solver.

Methodology

Each program is run five times, with 1, 2, and 4 threads on a USS type system. This is to flatten variance in runtime due to the influence of other users/processes running on the system; note that the minimum runtime of the five runs is used, not the average. A bash script (run.sh) is used to run all four n-body solvers with 1, 2, or 4 threads, redirecting the output to files in the **/perfdata** directory; which contain text files with corresponding names to the n-body solver they hold the data of (eg. **/perfdata/basic.txt** holds the data for the basic n-body solver program). The run.sh script uses the bash script command "**wait"** in between executions of the n-body solver programs, this ensures the sequential execution of each program - one after another, so that each instance of execution does not interfere with each other. The script also keeps all input (apart from number of threads) constant so that runtime measurements are accurate and consistent. Hence, **number of particles (400)**, **number of timesteps (1000)**, **size of timestep (0.01)**, **output frequency (100)**, and **G** (randomly generated initial conditions) are all kept constant.

The macro #define NO_OUTPUT was used in all four n-body solvers. This is to keep consistent with the experiment done by ***Pacheco*** stating "[n-body solvers] run on one of our systems with no I/O]", hence the NO_OUTPUT macro was used to augment the n-body solvers so that only runtime results were output and no particle data. In the *n-body reduced forces cyclic*, as well as *n-body reduced all cyclic*; cyclic partitioning was achieved using the **clause (static, 1)** after the relevant **omp for** pragmas. This instructs OMP to use cyclic (round-robin style) partitioning for the loop iterations, instead of the default block scheduling.

The minimum runtime collected for each of the four n-body solver programs, run with 1, 2, and 4 threads were then tabulated into a table which shows the difference in runtime between the basic, reduced default, reduced cyclic force computation, and reduced all cyclic n-body solvers. These results are used to determine which of the n-body solvers is most efficient and what are the disadvantages, if any, when using one over the other.

Results

Table of Runtime Data in Seconds				
Threads	Basic	Reduced default shed	Reduced forces cyclic	Reduced all cyclic
1	2.89s	1.67s	1.67s	1.67s
2	1.50s	1.26s	0.88s	0.89s
4	0.81s	0.74s	0.46s	0.56s

Figure 1. Runtime data in seconds of four n-body solvers run with 1, 2 and 4 threads using OMP

As shown in **Figure 1.** The *Basic* n-body solver, using only block partitioning, scaled well with an increase in number of threads (~50% reduction in runtime as thread count was doubled). The block partitioning likely contributed to this fairly linear speed up due to even load balancing among the threads, resulting in minimized idle time of threads and maximized overall parallel efficiency. Additionally, due to threads being allocated a block of iterations, the spatial locality of the block's contiguous memory likely reduces cache misses.

The *Reduced* n-body solvers, due to the "mirror computation" effect of calculating force using Newton's law of **Action & Reaction**, where forces for two particles can be calculated by only calculating for one particle and equating the signed opposite force to the other particle; workload on the threads decreases as the loop proceeds. Hence, cyclic scheduling may be beneficial as it assigns relatively equal amounts of work to each thread. As seen in **Figure 1.** By using Newton's 3rd law, runtime for all *Reduced* n-body solvers are almost identical. All are almost half the runtime of the *Basic* solver when run with a single thread and are all consistently faster than the *Basic* solver even at 2 and 4 threads.

When two threads are used, there is little difference (only a 0.01s difference) between the *Reduced* solver using cyclic scheduling for only **Compute_force()** and the *Reduced* solver using cyclic scheduling for all omp for loops. However, both are reasonably faster than the *Reduced* solver using default scheduling which had a runtime of 1.26s while *Reduced forces cyclic* and *Reduced all cyclic* had runtimes of 0.88s and 0.89s respectively (~30% faster runtimes). This shows that the superior load balancing when using cyclic scheduling, even with increased chance of cache misses, outperforms the block scheduling of the default *Reduced* solver.

However, as we increase thread count to 4, the performance of the *Reduced all cyclic* solver begins to degrade, with the *Reduced forces cyclic* solver now outperforming it as shown in **Figure 1**. Though, both are still faster than the *Basic* and *Reduced default sched* solvers by 30 to 45%.

The reason for the degradation in performance of the *Reduced all cyclic* solver as thread count increases is the overhead involved in false sharing begins to outweigh the overhead of partitioning all iterations in a round-robin fashion. As block partitioning allocates contiguous blocks of iterations, there is less likelihood of false sharing as memory for these iterations can be stored in the same cache line and accessed by a single thread. However, with cyclic partitioning, iterations are not contiguous (due to round-robin scheduling), so they are likely not stored in the same cache line. In the case that we increase the number of threads, we increase the chance that at least two threads will access different parts of the same cache line at the same time, causing the caching protocol to reload the cache line, significantly degrading performance.

Conclusion

The *Basic* solver takes between 30 to 60% longer than the *Reduced* solvers using cyclic scheduling for just force computation or all omp for loops. Moreover, the block partitioning of the *Reduced default shed* solver, even with less cache misses, is seen to be inferior to its cyclic partitioned counterparts. Therefore, if memory complexity is not an issue, using these two *Reduced* solvers is recommended. However, due to the extra memory requirement for the additional forces to be stored by a factor of the number of threads; when trying to compute larger numbers of particles, the reduced solver may not be possible. Finally, using cyclic scheduling only for the omp for loop responsible for executing the **Comput_force()** function, results in the best performance at larger thread counts due to less chances of false sharing between threads.

Appendix

Table of Raw Data				
Threads	Basic	Reduced default shed	Reduced forces cyclic	Reduced all cyclic
1	2.884642s	1.666566s	1.667472s	1.665815s
2	1.495200s	1.259377s	0.8759760s	0.8921908s
4	0.8113307s	0.7374821s	0.4635176s	0.5615546s

Appendix 1. Raw runtime data collected