INF560 - Project

MPI, OpenMP & CUDA

Lukas Köstler, Ezequiel Morcillo Rosso

March 13, 2019

OpenMP

- Use OpenMP [1] for shared memory parallelization
- Parallelize main compute loop: Particle simulation
- Each thread has zero-initialized weights absorbed array
- Critical section to combine weights absorbed arrays
- `num_threads` can be set programmatically

Speedup

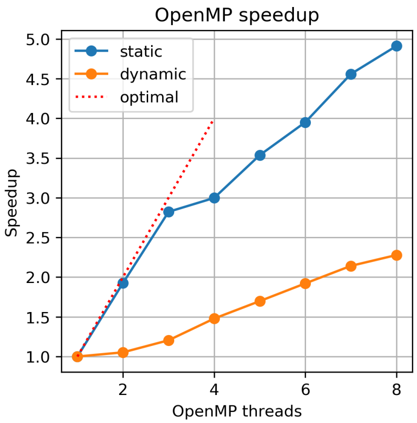


Figure: OpenMP speedup for `schedule(static)` vs. `schedule(dynamic)`.

CUDA

- Use CUDA [2] to parallelize for one node
- Each particle gets mapped to one CUDA thread
- Cell attributes in shared memory per thread-block
- Accumulate within thread-block with `atomicAdd`
- Accumulate into global array with `atomicAdd`
- After fixed number of steps transfer to CPU
- CPU sorts into active & inactive particles and calls kernel
- `nvprof`: more than 97% in kernel

Speedup

	Runtime [s]	**Speedup**
Sequential	25.1	1
OpenMP	4.84	5.18
CUDA	0.45	**55.8**

MPI General

- Divide domain of the simulation into several layers
- Each layer has particles inside it that need to be simulated
- Because the layers don't need to know about particles in other layers, the problem allows for the use of a distributed memory model (MPI)
- Each MPI process or rank gets a different layer
- Once a simulation step takes place, they communicate with the neighbouring layers the particles transmitted or received
- How should this communication take place?

MPI Communication - Simple

- First approach: use MPI Send and MPI Recv in a structured order
- A layer can't send nor receive more than once at a time, and neither can it send and receive at the same time
- Four stages of a communication, order is not important
- A receiving layer provides a big receive buffer for worst case scenario and gets the amount of particles received through the MPI status

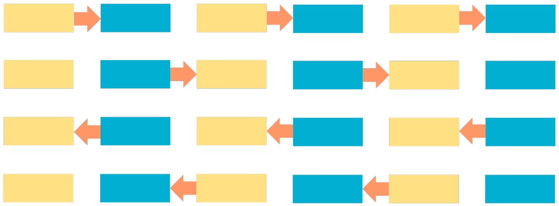


Figure: Communication schema - simple. Blue: Odd numbered layers. Yellow: Even numbered layers.

MPI Communication - Advanced

- Second approach: use MPI Sendrecv in a structured order
- Now a layer is sending and receiving at the same time
- Half the stages of a communication as the previous schema, order is not important
- Once again the buffer provided to receive is big enough to hold all particles in a worst case scenario
- Similar to a compact version of the previous communication schema, by grouping the previous rows into pairs

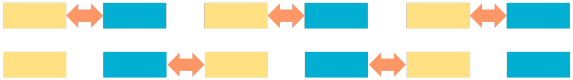


Figure: Communication schema - Advanced. Blue: Odd numbered layers. Yellow: Even numbered layers.

MPI Finalizing

- Once communications has taken place, all layers once again simulate the particles in their layer
- This process is repeated until the global simulation end
- In order to track when the simulation ends, all MPI processes keep track of the amount of particles disabled so far through MPI All Reduce
- Once the amount of particles disabled equals the total amount of particles to simulate, the simulation ends

Mixed Models

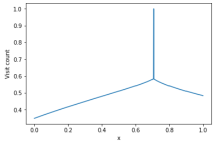
- MPI is the only model doing domain decomposition
- As such, it is easy to mix MPI with either CUDA or OpenMP, because the latter parallelize in the simulation of a layer
- CUDA and OpenMP might run together in the same node if said node has multiple processes in it
- The final version could be said to be a combination of the three; MPI for the domain decomposition, and in the layer simulation prioritizing the use of the GPU if available and if not use multiple OpenMP threads

Resource Sharing

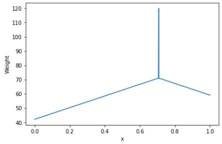
- Not all MPI processes have the same available resources
- Some might have to share a node with another MPI process, others might not have a GPU, and some might even need to share the resources with another program running on the node
- **Solution:** distribute the resources to the processes based on who is sharing them
- If only one process is in a node, give all available resources
- If more than one process is in a node and a GPU is available, assign it to the lowest ranked process along with one thread and evenly distribute the remaining threads to the remaining processes in the node
- If more than one process is in a node and no GPU is available, evenly distribute all possible threads among the processes sharing the node

Cell Weight

- Not all cells in the domain imply the same amount of work load
- Because particles get disabled as they go, the edges of the domain do not require that much computation
- **Solution:** Assign a weight to each cell according to their proximity to the initial x position of the particles, the closer the greater
- Cell weight assignment based on visit count of a simulation



(a) Simulation tracking the visit count



(b) Final cell weights

Figure: Comparison between visit count of a cell and the cell weight assigned to it

Final Domain Decomposition

- Having assigned the resources to each MPI process, it benchmarks its computing power by running a small scale simulation (with the assigned resources) and gets the result as the inverse of the time taken
- The program decomposes the domain so that the sum of the cell weights in a layer divided by the computing power of the process is the same for each one.
- Theoretically this should make all MPI processes take the same time to simulate a step, independently of resources and the layer it has assigned
- Due to the synchronous nature of this implementation, this helps reduce the idle time of some processes since they don't have to wait as much to receive from their neighbours

Demo

Node	Num. GPUs	OMP threads	MPI ranks
0	1	8	2
1	0	8	2
2	1	8	1
3	0	8	1
4	0	2	1

Table: Demo node specification.

Bibliography

OpenMP Architecture Review Board. **OpenMP**. 2018. URL: https://www.openmp.org.

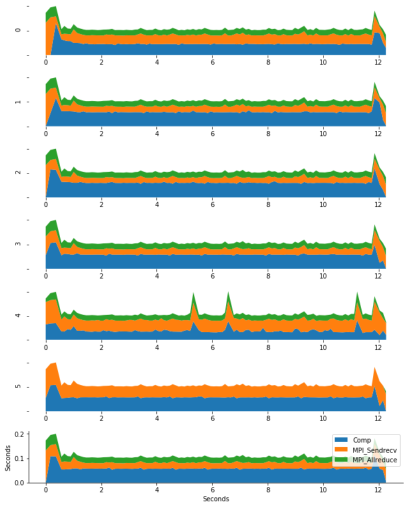
John Nickolls, Ian Buck, and Michael Garland. "Scalable parallel programming". In: **2008 IEEE Hot Chips 20 Symposium (HCS)**. IEEE. 2008, pp. 40–53.

Demo: Work Load Distribution

MPI rank	Num. GPUs	OMP threads	Num cells
0	1	1	420
1	0	7	96
2	0	4	58
3	0	4	55
4	1	8	208
5	0	8	117
6	0	2	46

Table: Demo resource allocation.

Demo: Results


Seconds
Seconds
Comp
MPI_Sendrecv
MPI_Allreduce