Parallel discrete crowd simulation using MPI

PA177 HPC Project

Roman Plášil

June 16, 2009

PA177 HPC Project

1 Introduction

The goal of this project was to implement a discrete crowd simulation parallelized using MPI. The problem is simulated using a 2D matrix where each cell (*patch*) represents either a free place or a solid wall. An *agent* (pedestrian) occupies one cell and can move to a neighbouring free cell. The world is divided between the available processes each of which moves the agents in its region. The simulation runs in discrete steps in which every agent moves by one cell, the regions are ajdusted to balance the load. Each step involves a barrier synchronization between processes so that all agents move at the same speed.

The implementation uses Boost.MPI, Boost.Serialization, png++ and is written in C++. Boost.MPI is an object oriented, type-safe wrapper over MPI (not an MPI implementation), which makes using it in C++ more convenient. Together with Boost.Serialization it is possible to send C++ classes using MPI in a straightforward manner.

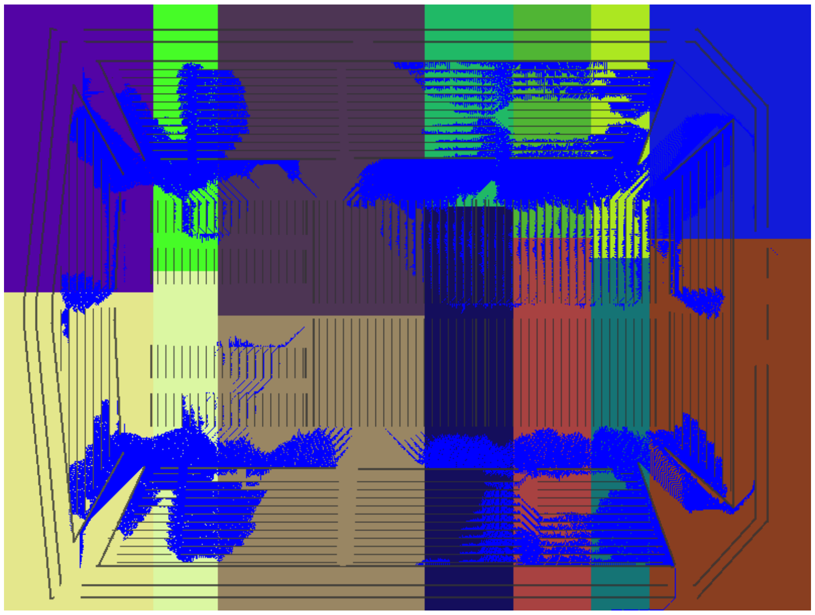


Figure 1: 111th frame of simulation. Blue pixels are agents, black are walls, colored are regions.

2 Algorithms

The simulation involves several algorithms which are briefly described in this section. We begin with an overview of the entire simulation.

There are in the world some patches which are marked as exit. This is where agents go to. In order for them to be able to know the way, each patch is assigned a value which represents the shortest distance to the nearest exit patch. Agents then try to walk to the patch with the least distance in their immediate neigbourhood. The distances are calculated once at the beginning of the simulation using the *Distances algorithm*.

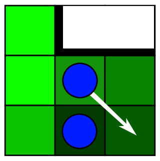


Figure 2: An agent decides to go to the patch which is closest to the exit and is free.

When distances are calculated, we partition the world into n regions where n is the number of available processors. This is the task for the *Partition algorithm*.

At this point initialization is done and we can start the simulation. The simulation runs in separated steps, called frames. In each frame, each process moves all agents placed in its region which is described in the *Agent algorithm*. Once all processes are done, they adjust region borders in order to mantain balanced system load. See *Region algorithm* for more details. Once the frame is done, the state of the world is written as an PNG image to the disk.

2.1 Distances algorithm

The code for this algorithm is in the file `Occupancy.cpp`, method `CalcDistances`. It is a parallel implementation where each process works on a horizontal slice of the world.

```
create an empty stack;
push all exit patches on the stack with distance 0;
while termination is not detected do
    receive messages and put them on stack;
    pop a patch;
    if if current distance of the patch is greater than that taken from stack then
        update distance;
        for all patches in neighbourhood do
            add that patch on the stack with distance derived from the new distance of current patch;
            if the patch lies in the area of another process, send it instead of adding on my stack;
        end
    end
end
```

Because it's a distributed algorithm, we cannot terminate simply when one stack gets empty, because it might receive a new stack item later. To address this issue, we use the Dijkstra-Scholten termination detection algorithm.

2.2 Partition algorithm

In order to ensure an even load balancing of agents between the processes, the regions must be placed so that each process has roughly the same number of agents. The initial set up of regions is built by this algorithm and mantained during simulation by the *Region algorith*.

The world is divided into $n/2$ vertical slices (n being the number of processes) so that each slice has approximately the same number of agents. The borders of these slices are called *major borders*, because they represent the first division. These borders can be adjusted during the simulation. Each vertical slice is then divided into two parts horizontally along a *minor border*. Each minor border is moved up or down during simulation. This approach ensures that each region has almost the same number of agents while keeping the complexity relatively low.

The implementation can be found in `RegionsManager.cpp`, method `initRegions`.

```
optimal ← $\frac{allagents}{n/2}$;
walk one column by one and count agents;
if close to optimal agents seen then
    create major border;
    find best horizontal dividing point using a similar algorithm;
    store two newly created regions;
end
```

2.3 Agent algorithm

Moving agents can be found in `Agent.cpp`, method `Frame`. Here we must find which of the neighbouring patches is empty and closest to the exit. If the patch is in a different region, we don't know if there's an agent and must send a message to find out. Distances and walls are known to each process, so we don't need to send any messages to find that out.

```
make a list of neighbouring patches which are not occupied by a wall or a local agent;
while the list is not empty do
    if a request arrived then
        if requested patch is free then
            place the agent here and accept;
        else
            deny the request;
        end
    end
    find the patch in the list which is closest to the exit;
    remove that patch from the list;
    if the patch is in my region then
        if the patch is not occupied then
            move there;
            done;
        end
    else
        send a request to the region which owns the patch;
        if accepted then
            remove this agent from current process;
            done;
        end
    end
end
```

To detect that all processes have finished moving their agents we use a different termination detection algorithm. It assumes that once a process finishes, it cannot be restarted by receiving a message. When a process finishes its work, it broadcasts this fact to everyone else and also sends the total count of messages sent to each process so they know how long to wait for messages from this process.

2.4 Region algorithm

After each frame the regions are updated to mantain even load balancing. The code for this task can be found in `Region.cpp`, method `UpdateSize`. First the major borders are moved and then the minor borders. In each pair of processes (top, bottom), only the top process makes decisions and only about the bottom and right border (as the left border is decided by another process).

Processes exchange information so that the top process can decide in which direction to move the border. The decision is made by counting how many agents would this vertical slice contain if we increased it by one to the left, one to the right or not at all. The alternative closest to optimal number of agents is chosen. Then we announce the decision and exchange agents on the column which changed its owner.

Updating minor borders is based on the same principle, but is simpler.

3 Usage

A binary for skirits is provided in the package in `build/run/`. The directory also contains sample worlds. To start the simulation, use mpich-p4 and provide the name of world image and how many frames to simulate as commandline arguments, for example:

```
./paracrowd stadium.png 400
```

The number of processes must be either 1 or an even number. The program produces a file called `distances.png` which displays the distance field for the world and then the frames. As writing the images is time-consuming, only every 3rd frame is written to a file called `framexx.png`.

To create a custom world, use a bitmap editor capable of writing png files (such as Gimp, or even MS Paint in Windows XP). Draw walls black (RGB 0 0 0), exit patches blue (RGB 0 0 255) and use red (RGB 255 0 0) to place agents. Other colors are considered as empty. The image can be of arbitrary size.

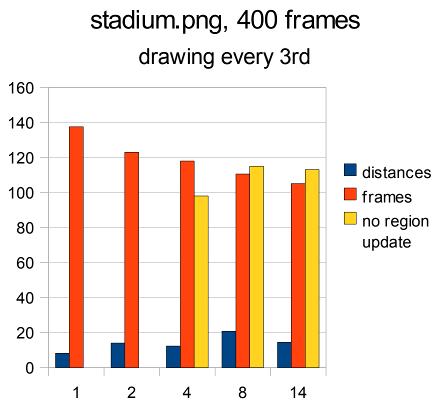
To rebuild the source, python must be installed, because it is used by the build system. Running `loc_scons.py` runs the build system scons (included in the package) and builds the code.

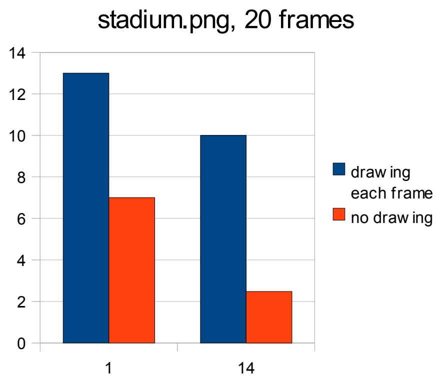
4 Results

Unfortunately, the application practically does not scale. The graphs show how many seconds a simulation takes using given number of processors. Drawing the frames represents a major bottleneck and is not parallelized. Even when drawing is reduced by only drawing every third frame, the program does not scale. As can be seen from the yellow bars in the graph, the effect of removing regions update is questionable, but it probably does not help. The only remaining communication which can hinder scalability is therefore sending agents over the borders and the barrier synchronization.

Additionally, calculating distances is always slower when more computers are involved. This may be partially caused by the termination detection, which sends a reply for every received message.

These results were obtained on a relatively large world (800x600 patches) with many agents (approx. 100 000).





5 Conclusion

Due to lack of time, the implementation works, but does not scale. More work would need to be invested in accurately identifying the bottlenecks and addressing them. Writing the results could be replaced by HDF or CDF using some parallel I/O library.

The simulation algorithm is not the best either – agents only choose the shortest path locally which is suboptimal globally. However, this could be changed without major changes to the communication scheme.