1 Introduction

Our company, TERMITE SIMULATION, INC., have developed software that can simulate thousands of termites wandering about a rectangular grid whilst picking up and dropping wood chips. Our initial simulations gave interesting results: Despite the fact that all termites followed the same basic set of rules, their collective behavior appeared coordinated. The termites started aggregating the wood chips into piles. In a sense, the aggregation of many simple and independent agents can give rise to a seemingly complex global behavior.

![Figure 1: Illustration of a small termite simulation. Left: An initial configuration. Right: A configuration after 1000 time steps. Initially, the wood chips (marked by the symbol 0) are distributed uniformly at random. After a while, the termites (marked by the symbol +) begin to assemble the wood chips into larger piles. Note that the grid is periodic in the sense that it wraps around in both the north–south and the east–west directions.](image)

To give you a better understanding of what we mean, study Figure 1. On the left side you see a small grid with many scattered wood chips marked by the symbol 0. You also see a small number of termites marked by the symbol +. This is the initial state of our simulation. After we let the simulation run for 1000 time steps, we get the configuration shown on the right. Notice that the wood chips have become more ordered and have begun to form large piles. If we were to let the simulation continue even further, then the number of piles tends to decrease and the size of each pile tends to increase.

As we briefly mentioned above, the termites follow a small set of basic rules. The rules are local in the sense that they describe how one termite acts based purely on local information. The termites act independently and do not take orders or directions from other termites or any other entity. The simulation proceeds in discrete time steps. The basic rules are as follows and are also applied in the following order:

1. The termite randomly chooses to either stay on the same course as before, turn left, or turn right.
2. If the termite is carrying a wood chip and there is a wood chip directly in front of the termite, then the termite drops the wood chip and turns 180 degrees.
3. If the termite is not carrying a wood chip and there is a wood chip where the termite is currently located, then the termite picks up the wood chip and turns 180 degrees.
4. The termite moves one step forward (if possible).
Note that a termite never performs both actions 2 and 3 in the same time step since that would imply that it picked up the wood chip it just dropped. The movement action 4 cannot be applied in the following situations:

1. The cell ahead is occupied by another termite; only one termite is allowed to occupy a cell at any point in time.

2. The termite is carrying a wood chip but the cell ahead already contains a wood chip; only one wood chip is allowed to occupy a cell at any point in time.

In these situations, the termite simply stays in its current cell.

Note that two termites are not allowed in the same cell at the same time. Therefore, there is a risk that two (or more) termites simultaneously try to move to the same cell. One of them will succeed and the others will fail and stay in their original cells. We currently believe that the precise details of how these conflicts are resolved will not significantly influence the quality of the simulations. In the current version of our software, the termites are processed sequentially in an arbitrary order and the conflicts are resolved using a “first-come, first-served” policy.

2 Our predicament

We would eventually like to simulate more realistic termite models using an even more realistic environment model. In particular, we would like to include a third dimension, obstacles, food sources, predators, etc. Note that real termite colonies can encompass several million individuals. Unfortunately and despite of our best efforts, the current software cannot handle much more than a few thousand termites, even with our simple termite and environment models.

We have therefore procured a new powerful supercomputer that consists of thousands of processor cores connected with a high-performance interconnect. Each node in the network consists of 48 processor cores that share the same memory and can be programmed with a so-called shared-memory programming model. To use more than one node, however, requires the use of the distributed-memory programming model MPI. We believe that using this new machine, we will be able to significantly improve the speed of the termite simulation software and eventually simulate hundreds of thousands or perhaps even millions of termites.

Our current predicament is that none of us here at the company knows how to program this machine effectively. This brings us to why you are reading this document. Our sources have informed us that you are quite adept at programming and can design efficient algorithms and data structures. More importantly for us, you are currently enrolled in a class that teaches parallel programming. We would therefore like to ask you for help in improving our software. If you do well, then you will be rewarded. There is one catch, however: Others have received the same offer as you. We have selected a number of highly qualified candidates for this job and have given all of you the same task. On March 18, 2013, we will gather you all and evaluated your proposed solutions. A series of benchmarks will be used to determine the most effective solution.

Prior to the deadline you should also prepare a 10 minute presentation in which you explain the major aspects of your work. You will be expected to present in front of the rest of the class and the team with the most interesting and informative presentation will be
rewarded. Your presentation should focus on the major performance issues that you identified and fixed as well as some details regarding your parallel implementation.

3 Your task

Your task is straightforward: Improve the speed of the termite simulation software so that we can simulate as many termites as possible in a given amount of time. Parallelization is, of course, one important aspect of this task, but it is not the only one. The data structures and the algorithms also play an important role in the performance of the software. We believe that it might be possible to improve these. You are allowed and encouraged to alter any data structures and/or algorithms, as long as the underlying termite and environment models remain the same. In order for your solution to be acceptable, it must have a significant and effective parallel component. You may select the programming model of your choice. The available programming models are PThreads, OpenMP, and MPI. Keep in mind that while the shared-memory programming models allow for incremental parallelization and more rapid development, the distributed-memory model enables the use of more processor cores. Choose wisely; the deadline is strict.

4 Requirements and evaluation

You are allowed to recruit up to two of your fellow classmates enrolled in the course 5DV011/VT13 to form a team of up to three individuals. All team members are expected to be familiar with all aspects of the solution. You are allowed to cooperate freely within the team and seek information and inspiration from any available sources, with the following restrictions:

1. You may not discuss and collaborate with classmates who are not on your team; remember that they are your competitors.
2. All external sources must be acknowledged.
3. All code and documents that you hand in must have been written by your team members.

Recall that the details of how the conflicts between termites are resolved is not important (see Section 1). In particular, your parallel implementation does not have to be faithful to the way our sequential software handles these conflicts. As is explained in Section 1, the sequential code uses an arbitrary but convenient method to resolve these conflicts. Maybe some other method is more convenient in a parallel environment.

Accompanying your parallel software should be a document that explains the modifications you made and the parallelization scheme. For each performance major modification, you should provide the following:

1. A justification for why the change is necessary and/or why it is expected to improve the performance. For example, evidence could be provided in the form of a profile of the program that identifies a hot spot, or it could be a theoretical argument for why the proposed alternative is preferable to the original.
2. An **explanation** of the modification. What did you modify and what were the consequences for the rest of the system?

3. An **evaluation** of the modification. Did the modification actually improve the performance or not? Did the parallelization scheme scale to many processors? Did the modification break the program?

*It is very important that you strive to use a scientific approach instead of making ad hoc changes.*

### 4.1 Evaluation criteria

In order to compare your solutions, we have decided to use a fixed set of evaluation criteria. They are currently as follows *but are subject to change should we discover that they are inappropriate*:

1. The **grid must be square**, but the size is otherwise not limited.
2. The initial fraction of **cells with termites** must be 1%.
3. The initial fraction of **cells with wood chips** must be 10%.
4. The number of **time steps** must be 5000.
5. **Wall clock time** must be measured using the following procedure:
   - (a) Initialize the parallel environment
   - (b) Prepare for the simulation
   - (c) Synchronize the processes/threads with a barrier
   - (d) Start a wall clock timer in each process/thread
   - (e) Simulate for the given number of time steps
   - (f) Synchronize again with a barrier
   - (g) Stop the timers
   - (h) Report the maximum measured wall clock time
   - (i) Repeat from Step (b) any number of times
   - (j) Clean up and shut down the parallel environment
6. The **evaluation metric** will be the size of the largest grid that can be simulated for the given number of time steps within **10 seconds of wall clock time**.

### 5 User’s guide

This section explains how to obtain, build, and run our sequential termite simulation software.

#### 5.1 How to obtain the software

Navigate to the course web page and download the gzipped tarball named “termites.tar.gz”.
5.2 How to extract and build the software

Extract the sources with the command

```
tar -zvxf termites.tar.gz
```

This produces the directory `termites/` in which the source code can be found together with a `Makefile` and a `README` file with some further instructions. The software can be built using the provided `Makefile`. The software can be built either for debugging or for performance testing (assertions turned off). To build the program for debugging, type

```
make debug
```

and to build the program for performance testing, type

```
make release
```

When switching from one type of build to another, you must first issue the command

```
make clean
```

to remove the executable and all intermediate object files. Failure to issue this command might result in a mixed build where some portions are built for debugging and others for performance testing.

5.3 How to run the software

The software executable is named `.run.x` and takes the following optional arguments:

- `-w N` Sets the width of the grid.
- `-h N` Sets the height of the grid.
- `-s N` Sets the number of time steps.
- `-t F` Sets the fraction of cells with a termite ($0 < F < 1$).
- `-c F` Sets the fraction of cells with a wood chip ($0 < F < 1$).
- `-v` Prints partial state information to `stdout` after each time step.
- `-?` Displays usage information and exits the program.

All the parameters have default values, so the program can be run without any arguments.