DOI: http://dx.doi.org/10.7551/978-0-262-31709-2-ch043

Using MapReduce Streaming for Distributed Life Simulation on the Cloud

Atanas Radenski

Chapman University, Orange, California
Radenski@chapman.edu

Abstract

Distributed software simulations are indispensable in the study of large-scale life models but often require the use of technically complex lower-level distributed computing frameworks, such as MPI. We propose to overcome the complexity challenge by applying the emerging MapReduce (MR) model to distributed life simulations and by running such simulations on the cloud. Technically, we design optimized MR streaming algorithms for discrete and continuous versions of Conway's *life* according to a general MR streaming pattern. We chose *life* because it is simple enough as a testbed for MR's applicability to a-life simulations and general enough to make our results applicable to various lattice-based a-life models. We implement and empirically evaluate our algorithms' performance on Amazon's Elastic MR cloud. Our experiments demonstrate that a single MR optimization technique called strip partitioning can reduce the execution time of continuous *life* simulations by 64%. To the best of our knowledge, we are the first to propose and evaluate MR streaming algorithms for lattice-based simulations. Our algorithms can serve as prototypes in the development of novel MR simulation algorithms for large-scale lattice-based a-life models[1].

Introduction

A-life has long relied on software simulations of the behavioral characteristics of living systems to facilitate the discovery of natural laws. Living systems involve vast numbers of evolving objects, and their software simulations can be data and computationally intensive. Large-scale life models may not fit in the memory of an average workstation, a challenge that can be overcome with the development of distributed a-life software. Distributed scientific simulations are usually implemented by using low-level libraries, such as MPI, which are difficult to program and require the development of custom fault-tolerance and load-balancing schemes --- a major challenge for scientists. Compute clusters to run distributed simulations can be expensive to build and complex to maintain. We believe that the complexity challenges of distributed life simulations can be overcome by applying the emerging higher-level MapReduce (MR) model to life simulations and by running such simulations on the cloud.

MR was initially developed to specifically satisfy Google's needs for large-scale distributed processing of unstructured text data [Dean and Ghemawat, 2008]. The subsequent implementation of MR within Apache's open-source Hadoop framework [White, 2012] stimulated the development of wide range of MR applications in diverse areas, including sets and graphs; AI, machine learning and data mining; bioinformatics; image and video; evolutionary computing; and statistics and numerical mathematics [Radenski and Norris, 2013]. MR users develop serial code that is automatically executed in parallel by the MR engine in a fault-tolerant and load-balanced manner. The simplicity of the MR model and the built-in fault tolerance and load-balancing functionality of the MR engine can be beneficial in the development of data-intensive distributed scientific applications in general and life software simulations in particular.

Our general goal in this paper is to investigate the applicability of the MR model to large-scale distributed a-life simulations. To do so, we focus on life models that are based on cellular automata (CA) because CA are relatively easy to parallelize and have been used in life modeling and simulation from the early days of a-life. Indeed, a-life and cellular automata share a closely tied history which began with the CA works of John von Neumann [Von Neumann and Burks, 1966] and continued with the development and exploration of the game of life --- or simply *life* --- by John Conway [Gardner, 1970; Bays, 2010]. Conway's *life* is a 2D CA with two states (alive/dead). A popular *life* variation adopts a nondiscrete representation with states that are continuously valued between zero and one [Peper et al., 2010]. We use the terms *discrete life* and *continuous life* correspondingly to distinguish between the two *life* models.

CA has been recognized as historically the most fundamental paradigm of a-life [Conti, 2008], and *life* has become known as the prototypical CA [Hoekstra et al., 2010]. Hence, we consider *life* to be a most suitable candidate for this first study of the applicability of the MR model to large-scale distributed life simulations. Technically, we develop and empirically evaluate MR *life* algorithms for the discrete and continuous *life* models. We also discuss how our algorithms can be transformed into algorithms for other versions of *life*, such as 3D *life* and *life* with larger neighborhoods. We outline a general MR streaming pattern that encapsulates the common general features of the *D-Life* and *C-Life* algorithms and can be followed for the design of other lattice-based simulation algorithms in the MR streaming model. Our algorithms are by no means limited to *life* and can be used as prototypes for developing large-scale MR simulation algorithms for other CA-based life models.

[1] This work was performed by Atanas Radenski as a guest faculty at Argonne National Laboratory in Illinois while on a sabbatical leave from Chapman University, California in spring 2013.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/284/1901671/978-0-262-31709-2-ch043.pdf by guest on 10 December 2024

Cloud computing is the use of hardware and software as service --- remotely, on-demand, and on a pay-per-use basis. Cloud computing can be viewed as a descendant of the well-established grid computing, enhanced with instant provisioning and utility computing. Elastic MR is one of the services provided by the Amazon's cloud computing platform, Amazon Web Services (AWS). Elastic MR is in fact Apache Hadoop hosted on AWS's Elastic Compute Cloud. We develop MR *life* algorithms in the Hadoop version of the MR model and then host the algorithms' execution on Elastic MR for the purpose of empirical performance evaluation. With AWS, we can launch various Elastic MR clusters as they are needed for our cost-effective experiments. We choose AWS because it is the first and currently the largest publicly available cloud computing platform.

Previously unknown complex self-organizing behavior of life models can be discovered by simulating vast numbers of generations for very large-scale model configurations. Such large-scale simulations may not fit on individual workstations but can be conveniently implemented in MR and executed on the cloud in a cost-effective, pay-per-use fashion. Our MR *life* algorithms and their empirical performance evaluation are intended to enhance scientists' understanding of the potential of MR and cloud computing in a-life research and open new opportunities for distributed large-scale life simulations.

The rest of this paper contains three sections. The first section describes how *life* simulations can be represented in the MR model. The second section discusses related work. The third section presents conclusions and possibilities for future wok.

Placing Life on MapReduce

This section introduces selected features of the MR model, defines MR algorithms for discrete *life* and for continuous *life*, offers a general MR streaming pattern, and ends with an empirical evaluation of the MR algorithms on the cloud.

MapReduce Models and Frameworks

Standard MR model. In the *standard MR model*, user-defined serial *Map* and *Reduce* methods transform *in parallel* an input set of key-value (KV) pairs into an output set of KV-pairs. Initially, *Map* is applied in parallel to individual KV-pairs from the input set to produce a first intermediate set of KV-pairs.

$$Inter1 = \{(k2,v2)\} = \{Map(k1, v1) \mid (k1,v1) \in Input\}$$

This set of KV-pairs is then automatically transformed by MR into a second intermediate set of KV-pairs in which all intermediate pairs with the same key are sorted and grouped together, creating a single KV-pair for each intermediate key.

$$Inter2 = \{(k2,list(v2))\} = MR\text{-}Sort\text{-}and\text{-}Group(Inter1)$$

Reduce then is applied in parallel to individual KV-pairs from the second intermediate set to produce an output set of KV-pairs.

$$Output=\{(k3,v3)\}=\{Reduce(k2,list(v2))\}|(k2,list(v2)\in Inter2\})$$

Input, intermediate, and output keys and values may or may not belong to different domains.

Consider for example the problem of counting word frequencies in a text document. In standard MR word count, input KV pairs represent document's lines (Table 1). The standard MR engine parses the input keys and values and provides them as ready-to-use KV arguments to *Map* and *Reduce*. *Map* method invocations (Figure 1) parse individual lines (automatically submitted to *Map* through the *value* parameter) and produce the first intermediate set of KV pairs, where individual words serve as keys and 1s serve as values (Table 1). MR then automatically sorts and groups the first intermediate set into a second intermediate set (Table 1). Finally, *Reduce* method invocations (Figure 1) sumup grouped values to output the final count for each word (Table 1). Note that this particular *Map* method ignores all input keys. For practical convenience, MR frameworks automatically generate some default keys for existing text documents.

Table 1: Standard MR – data sample

Input	*Inter1*	*Inter2*	*Output*
1 to be or 2 not to be	to 1 be 1 or 1 not 1 to 1 be 1	be 1 1 not 1 or 1 to 1 1	be 2 not 1 or 1 to 2

```
1: class Mapper:
2:     method Map (key, value):
3:         for word ∈value:
4:             Emit (key=word, value =1);
1: class Reducer:
2:     method Reduce (key, list-of-values):
3:         sum = 0;
4:         for value ∈list-of-values:
5:             sum +=value;
6:         Emit (key, value = sum);
```

Figure 1: Word count in the standard MR model

MR frameworks. The MR model has been implemented in three principal types of frameworks: distributed MR (targeted at clusters of workstations) [Dean and Ghemawat, 2008], shared-memory MR (targeted at multicore, multiprocessors workstations) [Talbot et al., 2011], and GPU MR [He et al., 2008]. Distributed MR frameworks are the most popular in practical computing. Any distributed MR framework incorporates an MR engine and a distributed file system (DFS) to hold the input, intermediate, and output datasets.

Google was the first to develop a proprietary distributed MR framework, which has been available and used only internally [Dean and Ghemawat, 2008]. The popularity of the MR model grew significantly with the release by Apache of the open-source Hadoop framework [White, 2012], which extended the standard MR model with additional functionality, such as *MR streaming*. Hadoop is now the defacto standard MR framework, and we therefore target our distributed MR *life* algorithms to Hadoop (see [Lee et al., 2012] for advantages and pitfalls of Hadoop MR). For the rest

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/284/1901671/978-0-262-31709-2-ch043.pdf by guest on 10 December 2024

of this paper we omit "Hadoop" in references to the Hadoop MR.

The MR engine invokes *Map* and *Reduce* methods within persistent tasks that are distributed over the Hadoop cluster; we refer to such persistent tasks as *mappers* and *reducers*. The MR engine uses intermediate keys to partition all intermediate KV-pairs among available reducers and to sort all KV-pairs that are fed into the same reducer. Hence, all intermediate KV-pairs with the same key are submitted to the same reducer in sorted order, although the same reducer can be assigned to handle several different keys [Radenski, 2012].

The MR framework is implemented in Java, and standard MR algorithms target the MR Java API, thus requiring significant Java expertise. In contrast to standard MR, MR streaming algorithms target higher-level languages such as Python. Because MR streaming is easier to understand and modify by domain scientists, who may not be Java experts but can work well with Python, we chose to develop MR streaming algorithms for distributed *life* simulations. Such MR streaming algorithms can be straightforwardly implemented in Python. If needed, MR streaming algorithms can be transformed into equivalent standard MR algorithms.

MR streaming model. In an essential departure from the standard MR semantics, MR streaming sorts but does not group intermediate same-key KV-pairs at all, and the *Reduce* method must handle multiple occurrences of the same key with corresponding partial values (rather than a single list of grouped values as in standard MR). The semantics of the *MR streaming model* can be specified as follows.

$$Inter1 = \{(k2,v2)\} = \cup\{Map(\{(k1, v1)\})|\{(k1,v1)\}\subseteq Input\}$$
$$Inter2 = \{(k2,v2)\} = MR\text{-}Sort\ (Inter1)$$
$$Output = \{(k3,v3)\} = \cup\{Reduce(\{(k2,v2)\})|\{(k2,v2)\}\subseteq Inter2\})$$

While the standard MR engine parses data sets' keys and values and provides them as ready-to-use KV arguments to *Map* and *Reduce*, the streaming MR engine provides all data via the standard input stream. Consequently, *Map* and *Reduce* must parse the input into keys and values on their own. Parsing input in a higher-level language is not necessarily a disadvantage of MR streaming: such parsing can be more straightforward and flexible than using the relatively complicated and rigid Java API to manage the input format in standard MR [Radenski and Norris, 2013].

In MR streaming word count, document's lines are viewed as input KV pairs represent with empty keys (Table 2). The *Map* method (Figure 2) parses individual lines and produces an intermediate set of KV pairs in which individual words serve as keys (Table 2). MR then automatically sorts the first intermediate set into a second intermediate set (Table 2). Finally, the *Reduce* method (Figure 2) uses a loop to sumup sorted values then outputs the final count for each word (Table 2).

The distributed execution of mappers and reducers forms a single MR streaming job (Figure 3). Several MR jobs can be iterated so that the output from one job is used as the input for the next one. All input datasets, intermediate datasets, and output datasets for MR jobs are stored in the DFS. Iterative MR processing can be achieved by using functionality that is external to the MR model or by using non-standard extensions of the MR model itself.

Table 2: MR streaming – data sample

Input	*Inter1*	*Inter2*	*Output*
to be or not to be	to 1 be 1 or 1 not 1 to 1 be 1	be 1 be 1 not 1 or 1 to 1 to 1	be 2 not 1 or 1 to 2

```
1: class Mapper:
2:     method Map ():
3:         for line ∈ stdin:
4:             for word ∈ line:
5:                 Emit (key=word, value =1);
1: class Reducer:
2:     method Reduce ():
3:         last-word = None;
4:         for line ∈ stdin:
5:             current-word, value = Parse (line);
6:             if current-word ≠ last-word:
7:                 if last-word ≠ None:
8:                     Emit (last-word, sum)
9:                 last-word = current-word; sum = 0
10:            sum +=value;
11:        Emit (last-word, sum)
```

Figure 2: Word count in the MR streaming model

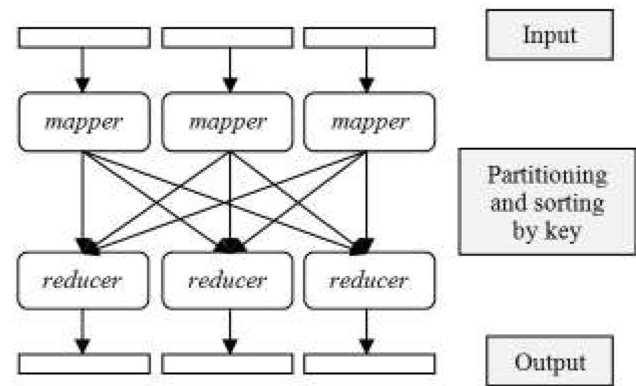

Figure 3: MR streaming job dataflow

Discrete Life in MapReduce

Discrete life is a CA with two states (dead/alive) over an infinite 2D lattice that evolves according the *2,3/2 rule*: An alive cell with 2 or 3 alive neighbors in its Moore neighborhood stays alive, and a dead cell with 2 alive neighbors becomes alive. (As defined earlier, *discrete life* is our term for Conway's *life,* as opposed to continuous life.) This rule can be generalized as $E_1, E_2, \ldots / F_1, F_2, \ldots$ to define a family of *life*-like CA [Bays 2010]. The evolution of *discrete life* is deterministic and is completely defined by its initial state.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/284/1901671/978-0-262-31709-2-ch043.pdf by guest on 10 December 2024

Standard MR and MR streaming both operate on KV-pairs. In MR streaming, each KV-pair has to be represented as a single line of text. Input, output, and intermediate datasets in 2D *discrete life* simulations represent living cells as KV-pairs in which the key part consists of the cells' coordinates *(row, col)* and the value part is empty. Cells that are not included in the dataset are assumed dead (Table 3).

Table 3: *Life* data representation – data sample

Discrete life	*Continuous life*
	1 1 0.0
	1 2 0.5
	1 3 0.0
2 1	2 1 1.0
2 2	2 2 1.0
2 3	2 3 1.0
	3 1 0.0
	3 2 0.5
	3 3 0.0

The initial *life* state is stored in one or more text files on the DFS before processing. An MR streaming job automatically splits the input dataset into independent blocks, which are submitted to mappers, one line per cell at a time (Figure 3). Mappers process individual cells and emit intermediate KV-pairs that are partitioned and sorted by the MR engine and then input into reducers. Reducers process intermediate KV-pairs and emit output results that are stored back on the DFS, one output file per reducer. The output dataset can then be used as the input for a subsequent MR streaming job. Hence, a single *life* simulation step is implemented as a single MR streaming job. A multi-step *life* simulation can be realized as a MR streaming job iteration by using tools that are outside of the MR streaming model. This pattern is followed for both *discrete life* simulations (discussed in this subsection) and *continuous life* simulations (discussed in the next subsection).

A single *discrete life* simulation step can be implemented by means of a technique known as MR *message passing* [Lin and Schatz, 2010]. For each input cell (by definition alive) a mapper can emit intermediate KV-pairs that are interpreted as messages to all of the cell's neighbors, alive and dead. Such messages from a living cell simply notify all of the cell's neighbors --- living or alive, and including the cell itself --- that the living cell belongs to those cells' neighborhoods. Messages to the same cell are dispatched to the same reducer, and each reducer receives all its messages in sorted order. This enables a reducer to count the living neighbors of individual cells, to determine the cells' next states (dead/alive) and to emit only living cells.

MR message passing can generate numerous small messages. Higher message volume can become a MR network bottleneck and be detrimental to performance. The number of messages can be reduced by applying local in-mapper aggregation (LA) optimizations [Lin and Dyer, 2010]. LA is applied to *discrete life* simulation in MR streaming as follows. For each neighbor of each input cell, the mapper increments (rather than emit immediately) a cell's counter within a local in-memory hash. Aggregated counts for each cell are emitted just before the mapper termination. Such aggregated counts for the same cell are summed up by a single reducer that determines the next state of the cell. A *discrete life* single-step simulation algorithm in MR streaming referred to as *D-Life*, is shown in Figure 4. This algorithm assumes no size limits on the lattice and can operate on very large *discrete life* instances.

```
1: class Mapper:
2:    method Map ():
3:        hash = ∅
4:        for line ∈ stdin:
5:            cell = (row, col) = Parse (line)
6:            Emit (cell, tag = Alive, count = None)
7:            for neighbor in Neighborhood (cell):
8:                hash[neighbor] += 1
9:        for cell in hash:
10:           Emit (cell, tag = None, count = hash[cell])
1: class Reducer:
2:    method Reduce ():
3:        last-cell = None;
4:        for line ∈ stdin:
5:            current-cell, tag, count = Parse (line);
6:            if current-cell ≠ last-cell :
7:                if last-cell ≠ None:
8:                    if Next-State-Is-Alive (last-cell):
9:                        Emit (last-cell)
10:               last-cell = current-cell; alive-neighbors = 0
11:           alive-neighbors += count;
12:       if Next-State-Is-Alive (last-cell): Emit (last-cell)
```

Figure 4: Single-step *discrete life* simulation algorithm in MR streaming (*D-Life*)

In the *D-Life* algorithm (Figure 4), the 2,3/2 rule and the Moore neighborhood of Conway's *discrete life* are encapsulated in methods Next-State-Is-Alive and Neighborhood. Hence, the *D-Life* algorithm can be applied to simulate other *life*-like CA by merely adapting the two methods to alternative rules and neighborhoods. *Larger than life (LtL)*, for example, is a family of CA generalizing *discrete life* to large neighborhoods and general birth and survival thresholds [Evans, 2010]. Any *LtL* instance can be simulated by the *D-Life* algorithm with methods Next-State-Is-Alive and Neighborhood defined to properly implement the specific rule and neighborhood. Variations of *discrete life* in larger dimensions [Bays, 1987] can be simulated with the *D-Life* algorithm by merely extending the cell representation to a larger number of dimensions.

Continuous Life in MapReduce

Continuous life is a CA with continuously valued states from the *0..1* range. *Continuous life* employs a transition rule that is formulated as a nonlinear expression with a temperature parameter T and two parameters: $E0$ (energy shift parameter) and $x0$ (state shift parameter). In the limit $T \rightarrow 0$ and with appropriately chosen values for $E0$ and $x0$, the behavior of *continuous life* coincides with that of *discrete life* [Adachi et al., 2004].

We rewrote the original *continuous life* transition function [Adachi et al., 2004] in an equivalent form that is more suitable for a MR implementation. Let us denote the state of a given *cell* at time *t* as *S(cell, t)*. In *continuous life*, each *cell*

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/284/1901671/978-0-262-31709-2-ch043.pdf by guest on 10 December 2024

undergoes transitions at discrete time steps according to the following set of equations.

(1) $S(cell, t + 1) = F(E(H(cell, t)))$
(2) $F(z) = b / (b + 1)$ where $b = exp(2*z/T)$
(3) $E(x) = E0 - (x - x0)^2$
(4) $H(cell, t) = S(cell, t) + 2*\sum_{cell' \in Neighborhood(cell)} S(cell', t)$

For *continuous life* simulations in the MR streaming model, input, output, and intermediate datasets represent living cells as KV-pairs in which the key part consists of the cells' coordinates *(row, col)* and the value part is the cell's *state*. Hence, all cells are explicitly represented in *continuous life*'s datasets, in contrast to *discrete life's* datasets which contain living cells alone (Table 3).

To implement a single *continuous life* simulation step in the MR streaming model, we combine the MR message passing technique with the *strip partitioning* optimization technique. MR message passing was introduced in the previous subsection for simulation of *discrete life*. Strip partitioning was proposed originally for large-scale relaxation algorithms [Radenski and Norris, 2013].

A mapper inputs KV-pairs of the form *(cell, state)* from the DFS. For each neighbor *cell'* of the input *cell*, the mapper aggregates in memory partial values of the term *H(cell', t)* according to equation (4). Locally aggregated partial values for each *cell'* are emitted just before the mapper termination as intermediate KV-pairs. Such aggregated values for the same *cell'* are summed up by a single reducer to determine the complete value of *H(cell', t)*. The reducer then uses this complete value to calculate the next state of the *cell'* according to equations (1) --- (3) and to emit the new state to the DFS.

In MR, the intermediate output of each mapper is directed to specific reducers by the MR *partitioner*. In general, given an intermediate key-value record, the default MR partitioner hashes the key into a reducer index for the record, balancing the load among reducers [Radenski and Norris, 2013]. In *continuous life* simulation, intermediate records are in the form *(key= cell',* $H_{partial}$*(cell', t))*. Because of hashing, adjacent cells are likely to be directed to different reducers, and thus output into different DFS files (because each reducer's output is placed by the MR engine into a separate DFS file). At the next simulation step, adjacent cells are likely to be submitted to different mappers (because such adjacent cells are likely to have been output to different DFS files at the previous simulation step and because different input files are submitted to different mappers). Thus, the default MR partitioner tends to disperse neighborhoods and reduce data locality, which impairs local in-mapper aggregation and is detrimental for performance. [Radenski and Norris, 2013] Fortunately, the strip partitioning optimization technique can help reduce dispersion and preserve data locality. With strip partitioning, a mapper sends whole strips of consecutive CA lattice rows to the same reducer. Technically, the mapper outputs intermediate KV-pairs of the form:

(key=(strip, cell'), value=$H_{partial}$*(cell', t))*,

where *strip* is an index that identifies individual strips of CA lattice rows. The *strip* index is calculated as s*trip = row / strip-length* where *strip-length* is a simulation parameter. Because the *strip* index is part of the mappers' intermediate output, all cells from the same strip will be directed to the same reducer during partitioning (see Figure 3), possibly from different mappers. Therefore, such adjacent cells will remain in the same output file as produced by the reducer. This strategy preserves data locality during iterative simulation and promotes performance. The *strip* index is emitted by mappers to facilitate partitioning alone and is ignored by reducers upon its receipt as part of the key.

A *continuous life* single-step simulation algorithm in MR streaming based on local in-mapper aggregation and strip partitioning (referred to as *C-Life*) is shown in Figure 5.

```
1: class Mapper:
2:     method Map ():
3:         hash = ∅
4:         for line ∈stdin:
5:             cell, state = Parse (line)
6:             hash[cell] += state
7:             for neighbor in Neighborhood (cell):
8:                 hash[neighbor] += 2*state
9:         for cell in hash:
10:            strip-number = cell.row / strip-length
11:            Emit (cell, strip-number, hash[cell])
1: class Reducer:
2:     method Reduce ():
3:         H = 0; last-cell = None
4:         for line ∈stdin:
5:             strip-number, current-cell, in-value = Parse (line);
6:             if current-cell ≠ last-cell :
7:                 if last-cell ≠ None:
8:                     Emit (last-cell, state=F(E(H))
9:                 H = 0; last-cell = current-cell
10:            H += in_value
11:        Emit (last-cell, state=F(E(xi))
```

Figure 5: Single-step *continuous life* simulation algorithm in MR streaming (*C-Life*)

For practical purposes, we assume that *continuous life* evolves over a finite rectangular lattice with periodic boundary conditions. This assumption prevents the otherwise unlimited growth of datasets in iterative *continuous life* simulation. While the *continuous life's* lattice is assumed finite, it can be potentially very large in an MR implementation.

In the *C-Life* algorithm (Figure 5), the *continuous life's* transition rule and the neighborhood are encapsulated in separate methods that can be adapted to alternative rules and neighborhoods, without any changes to the *C-Life* algorithm proper.

General MapReduce Streaming Pattern

We have designed the *D-Life* and *C-Life* algorithms by following a general MR streaming pattern. Our pattern is outlined in Figure 6. The pattern describes a family of MR streaming algorithms that execute as follows (Figure 6):

- *Map* processes and aggregates locally each input KV-pair. Processing may consist of various actions, such as

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/284/1901671/978-0-262-31709-2-ch043.pdf by guest on 10 December 2024

emitting the KV-pair (as done in *D-Life*) and/or performing mathematical operations (as done in both *D-Life* and *C-Life*). Aggregation involves storing intermediate results locally in a hash.

- Just before termination, *Map* emits all aggregated results as intermediate KV-pairs, with the optional use of strip partitioning (as done in *C-Life*).
- *Reduce* processes and accumulate locally each intermediate KV-pair. Partitioning and sorting by key (Figure 3) guarantee that all intermediate KV-pairs with the same key are submitted to the same reducer in an uninterrupted sequence. Processing may consist of various actions, such as performing mathematical operations (e.g., increment in both *D-Life* and *C-Life*). Accumulation involves storing intermediate processing data locally (such as *alive-neighbors* in *D-Life* and *H* in *C-Life*).
- *Reduce* ends the processing of each uninterrupted same-key KV-pair sequence by calculating the key's final value and emitting an output KV-pair accordingly. (In *D-Life*, this involves deciding whether a cell will be dead or alive; in *C-Life* this involves calculating the cell's next state.)

```
1: class Mapper:
2:     method Map ():
3:         for input-kv-pair ∈stdin:
4:             Process-and-Aggregate ()
5:         Emit-All-Aggregated ()
1: class Reducer:
2:     method Reduce ():
3:         for intermediate-kv-pair ∈stdin:
4:             if Current-Key-Is-Different-From-Previous-Key ():
5:                 Emit (previous-key, Final-Value ())
6:                 Initialize-Current-KV-Pair-Processing ()
7:             Process-and-Accumulate ()
8:         Emit (last-key, Final-Value ())
```

Figure 6: General MR streaming pattern

Cloud Implementation and Empirical Evaluation

We implemented our *D-Life* and *C-Life* MR streaming algorithms in Python and then used the implementations for empirical algorithm evaluation on Amazon's Elastic MR cloud. We chose Python because it is higher-level language that significantly shortens development efforts and time in comparison with other mainstream languages, such as Java or C++. Our experiments were performed with Hadoop 1.0.3 on an Elastic MR cluster of up to 17 large instances, a master instance and up to 16 core instances.

We experimented with two versions of the *C-Life* algorithm, designated as *C-Life-16* and *C-Life-0*. The *C-Life-16* version uses a strip size of 16. Our preference to a strip size of 16 is based on some preliminary performance experiments with various strip sizes. The *C-Life-0* version does not use the strip partitioning optimization at all, hence its strip size of 0.

We ran *D-Life*, *C-Life-16*, and *C-Life-0* on the Elastic MR cloud to measure their execution times. Execution times for the first simulation step can be influenced by the initial data layout; but once data are shuffled by the first simulation step, execution times stabilize. We measured execution times for the algorithms' second simulation steps over randomly generated square lattices. The initial datasets for *D-Life* was generated with alive cell probability of $p=0.5$. Recall that only alive cells are represented in *D-Life*'s datasets; in contrast, all cells and their states are explicitly represented in *C-Life* datasets. Given the same lattice size, *D-Life*'s datasets are smaller than *C-Life*'s datasets proportionally to p.

Table 4: *Same-data* performance (in min) on up to 16 nodes

Nodes	1	2	4	8	16
D-Life	16.3	22.6	11.6	6.9	4.2
C-Life-0	32.2	39.7	29.4	17.2	11.0
C-Life-16	37.4	19.9	9.5	6.8	4.0

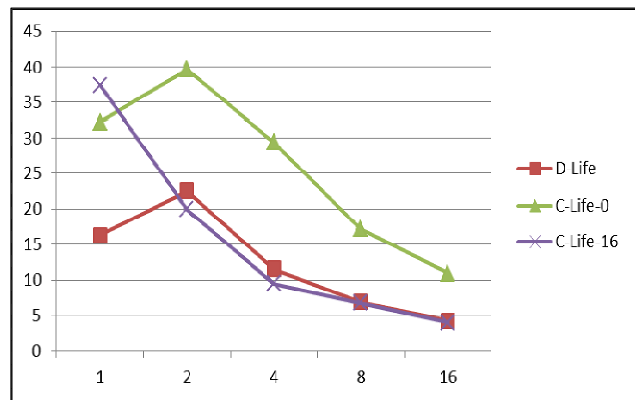

Figure 7: Visualization of *same-data* performance (in min, vertical axis) on up to 16 nodes (horizontal axis)

Table 5: *Same-load* performance (in min) on up to 16 nodes

Nodes	1	2	4	8	16
D-Life	2.0	3.1	3.2	4.0	4.2
C-Life-0	2.5	4.9	7.8	10.9	11.0
C-Life-16	2.9	3.6	3.3	3.8	4.0

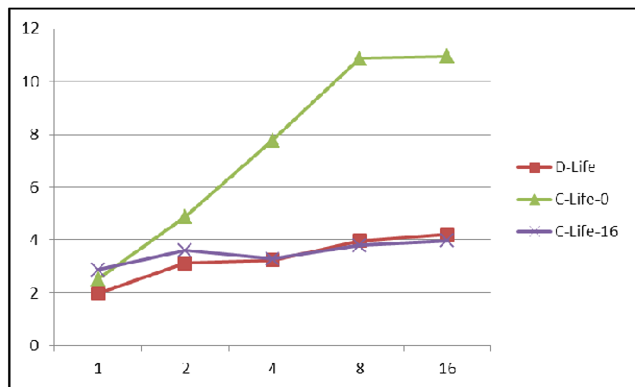

Figure 8: Visualization of *same-load* performance (in min, vertical axis) on up to 16 nodes (horizontal axis)

With all algorithms, we performed *same-data* and *same-load* performance evaluations. For same-data evaluation, we

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/284/1901671/978-0-262-31709-2-ch043.pdf by guest on 10 December 2024

measured algorithms' execution times in minutes on MR clusters with *i* core instances, *i = 1, 2, 4, 8, 16* over randomly generated square lattices of approximately $16*10^7$ cells (Table 4 and Figure 7). For same-load evaluation, we measured the algorithms' execution times in minutes on MR clusters with *i* core instances, *i = 1, 2, 4, 8, 16* over randomly generated square lattices of approximately $i*10^7$ cells (Table 5 and Figure 8).

Our performance measurements demonstrate that strip partitioning optimization --- used in *C-Life-16* but neither in *C-Life-0* nor in *D-Life* --- gives a performance advantage of *C-Life-16* over *C-Life-0* and *D-Life*.

- The execution time of *C-Life-16* for a single simulation step on 16 task nodes is 64% less than the execution time of *C-Life-0* for the same task (see data in last columns of Table 4 or 5).
- The execution time of *C-Life-16* decreases in a smooth and predictable manner with the increase of task nodes, in contrast to both *D-Life* and *C-Life-0* (Figure 7).
- *C-Life-16* scales much better than *C-Life-0* and a little better than *D-Life* (Figure 8).
- In terms of absolute execution time, the performance of *D-Life* seems to rival that of *C-Life-16*, especially for larger numbers of task nodes (Figures 7 and 8); yet simulation of *discrete life* is computationally less intensive than simulation of *continuous life* and *D-Life* operates on smaller datasets than *C-Life* (given the same lattice size).

Related Work

Cellular automata have been extensively studied since the early days of a-life [Langton, 1986]. A recent book offers a representative collection of approaches to the simulation of complex systems by CA [Hoekstra et al., 2010]. Another recent book covers current developments specifically in the game of Conway's *life* research [Adamatzky, 2010].

Our work on distributed MR *life* simulation builds on the MR strip-partitioning optimization originally introduced for a MR relaxation algorithm [Radenski and Norris, 2013]. Message passing was first studied in the context of data-intensive graph algorithms [Lin and Schatz, 2010] and later adapted to MR relaxation [Radenski and Norris, 2013]. Local in-mapper aggregation was originally designed to speed-up data-intensive text processing [Lin and Dyer, 2010] and was adapted to DNA sequence analysis [Radenski and Ehwerhemuepha, 2013] and relaxation [Radenski and Norris, 2013].

Our proposed *D-Life* and *C-Life* algorithms perform only a single *life* simulation step. A multistep *life* simulation is an iterative relaxation process that cannot be directly expressed in the pure MR parallelism model. We are among those who iterate pure MR steps by means of custom scripts expressed in common general-purpose languages. Others modify the pure MR model and implement new MR frameworks to facilitate iterative MR processing, such as iMapReduce [Zhang et al., 2012] and Twister [Ekanayake et al., 2010]. Potential ease of use and performance benefits of such iterative frameworks for multistep large-scale *life* simulations are yet to be studied.

Distributed MR relies exclusively on the DFS for the representation of intermediate datasets, including messages passed by our *D-Life* and C-life algorithms. Using the file system for message passing can be detrimental to performance but can be avoided with problems that fit entirely in memory. *In-memory MR* frameworks, such as Phoenix [Talbot et al., 2011] and M3R [Shinnar, 2012] aim to accelerate relatively small MR parallel applications by using hash tables to store intermediate key-value records in memory rather than on the DFS. Substantial speed-up benefits of in-memory frameworks for *life* simulations seem likely but are yet to be investigated.

Discrete life, a simple CA capable of generating diverse complex behavior, has stimulated many to design basic and advanced algorithms for its simulations and implement them in software. Basic serial and parallel implementations of *discrete life* have proven so worthy as to be incorporated in the computing curriculum [Wick, 2005; Hochstein et al., 2005]. Advanced *discrete life* simulation algorithms have been studied in traditional parallel computing models: shared memory [Ma et al., 2012], distributed [Xia et al., 2004], and mixed-mode [Smith and Bull, 2001]. To the best of our knowledge, we are the first to apply and evaluate the emerging MR model's applicability to distributed *life* simulations.

Various software frameworks have been developed and used to emulate of lattice-based a-life models since the early days of a-life, a trend that eventually began with the first *discrete life* programs. Lattice-based a-life software emulators continue to be used and developed [Komosinski and Adamatzky, 2010, Part II]. Notable examples include Discrete Dynamics Lab (DDLab), a set of tools for simulation of CA and other discrete structures [Wuensche, 2011]; NetLogo, a multiagent programmable modeling environment [Tisue and Wilensky, 2004]; and EINSTein, a multiagent simulator of land combat [Ilachinski, 2004]. We are the first to study the usability of MR in the a-life context.

Conclusions and Future Work

In this paper, we investigate the applicability of the MR streaming model to the simulation of discrete and continuous *life* CA. We chose *life* CA because of their simplicity, a feature that makes them attractive as an initial test bed for distributed MR simulation approaches. We use MR message passing, local in-mapper aggregation, and strip partitioning to design the *D-Life* and *C-Life* algorithms for the simulation of discrete and continuous *life* correspondingly in the MR streaming model. We also formulate a general MR streaming pattern that we have followed in our design of *D-Life* and *C-Life* and that can be followed for the design of other CA simulation algorithms in the MR streaming model. We implement *D-Life* and *C-Life* on Amazon's Elastic MR cloud and empirically evaluate their performance. Our experimental results show that strip partitioning can reduce the execution time of continuous *life* simulations by 64%. To the best of our knowledge, we are the first to propose and evaluate MR streaming algorithms for lattice-based simulations.

In future projects, our proposed MR streaming algorithms can be used as prototypes in the development of novel MR simulation algorithms for large-scale CA in general and for lattice-based a-life models in particular. The field of applications of our approach can possibly be extended to the

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/284/1901671/978-0-262-31709-2-ch043.pdf by guest on 10 December 2024

field of multi agent simulation (MAS). MAS can be done in the standard MR model on a small Hadoop cluster [Sethia and Karlapalem. 2011] but the feasibility of MR streaming for larger scale MAS on the cloud is yet to be investigated.

Future work should aim at performance improvements. Performance improvements can be achieved by using standard MR instead of MR streaming and by using in-memory MR instead of distributed MR.

- The MR streaming engine does not aggregate intermediate KV-pairs at all, while the standard MR engine does it automatically; aggregation by the engine can be more efficient than custom aggregation in a higher-level language such as Python. For similar reasons, I/O, including KV-pair parsing, can also be more efficient in standard MR in comparison with MR streaming. With the use of standard MR instead of MR streaming, the tradeoffs is simplicity and ease of use for speed.
- Distributed MR frameworks use a DFS for all input, intermediate, and output datasets. The total I/O time can be much larger than the actual processing time. I/O performance losses can be offset by using in-memory MR frameworks, instead of distributed ones, for datasets that can fit in memory. With the use of in-memory MR instead of distributed MR, the ability to process unlimitedly large datasets is traded for speed.

As future work, our *D-Life* and *C-Life* MR streaming algorithms and our general MR streaming pattern can be translated into the standard MR model and ported onto an in-memory MR framework, to evaluate the performance gains with standard MR and in-memory MR.

Acknowledgement. Thanks are due to the anonymous reviewers for their valuable comments and recommendations.

References

Adachi, S., Peper, F., and Lee, J. (2004). The Game of life at finite temperature. *Physica D: Nonlinear Phenomena*, 198:182–196.

Adamatzky, A., editor. (2010) *Game of Life Cellular Automata*. Springer, London, UK.

Bays, C. (1987). Candidates for the Game of Life in three dimensions. *Complex Syst*. 1:373–400.

Bays, C. (2010). Introduction to cellular automata and Conway's Game of Life. In [Adamatzky, 2010], pages 1-7.

Conti, C. (2010). The enlightened Game of Life. In [Adamatzky, 2010], pages 453-464.

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *CACM* 51:107-113.

Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. (2010) Twister: A runtime for iterative MapReduce, In *19th ACM International Symposium on High Performance Distributed Computing*, pages 810-818, ACM, New York.

Evans, K. (2010). Larger than Life's extremes: Rigorous results for simplified rules and speculation on the phase boundaries. In [Adamatzky, 2010], pages 179-221.

Gardner, M. (1970). Mathematical games: The fantastic combinations of John Conway's new solitaire game "Life". *Sci. Am.* 223:120–123.

He, B., Fang, W., Luo, Q., Govindaraju, N., and Wang, T. (2008). Mars: a MapReduce framework on graphics processors. In *17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260-269. ACM, New York.

Hochstein, L., Carver, J., Shull, F., Asgari, S., and Basili, V. (2005). Parallel programmer productivity: A case study of novice parallel programmers. In *Supercomputing 2005*, page 35. IEEE Computer Society, Los Alamitos, CA.

Hoekstra, A., Kroc, J., and Sloot, P., editors. (2010). *Simulating Complex Systems by Cellular Automata*. Springer, Berlin.

Ilachinski, A. (2004). *Artificial War: Multiagent-Based Simulation of Combat*. World Scientific Publishing Co., Inc., Hackensack, NJ.

Komosinski, M. and Adamatzky, A. (2009). *Artificial Life Models in Software*, 2nd ed. Springer, Berlin.

Langton, C. (1986). Studying artificial life with cellular automata. *Physica D: Nonlinear Phenomena*, 22:120-149.

Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y. D., and Moon, B. (2012). Parallel data processing with MapReduce: a survey. *SIGMOD Rec.*, 40:11-20.

Lin, J. and Dyer, C. (2010). *Data-Intensive Text Processing with MapReduce.* Morgan and Claypool, San Francisco, CA.

Lin, J. and Schatz, M. (2010). Design patterns for efficient graph algorithms in MapReduce, In *8th Workshop on Mining and Learning with Graphs*, pages 78-85. ACM, New York.

Ma, L., Chen, X., and Meng, Z. (2012). A Performance Analysis of the Game of Life Based on Parallel Algorithm, Technical Report, Computer Science and Engineering Department, Sichuan University.

Peper, F., Adachi, S., and Lee, J. (2010). Variations on the Game of Life. In [Adamatzky, 2010], pages 235-255.

Radenski, A. (2012). Distributed simulated annealing with MapReduce. *LNCS*, 7248:466-476.

Radenski, A. and Ehwerhemuepha, L. (2013). Speeding-up codon analysis on the cloud with local MapReduce aggregation. *Information Sciences* (submitted).

Radenski, A. and Norris, B. (2013) Distributed large-scale Laplace relaxation on the cloud with MapReduce. Technical Report, MCS Division, Argonne National laboratory, IL.

Sethia, P. and Karlapalem. K. (2011) A multi-agent simulation framework on small Hadoop cluster. *Eng. Appl. Artif. Intell.* 24:1120-1127.

Shinnar, A., Cunningham, D., Herta, B., and Saraswat, V. (2012) M3R: Increased performance for in-memory Hadoop jobs. *VLDB Endowment*, 5:1736-1747.

Smith, L. and Bull, M. (2001). Development of mixed mode MPI/OpenMP applications. *Scientific Programming*, 9:83-98.

Talbot, J., Yoo, R., and Kozyrakis, C. (2011). Phoenix++: Modular MapReduce for shared-memory systems. In *2nd International Workshop on MapReduce and Its Applications*, pages 9-16, ACM, New York.

Tisue, S. and Wilensky, U. (2004). Netlogo: A simple environment for modeling complexity. In Minai A. and Bar-Yam Y., editors, *International Conference on Complex Systems*, pages 16-21. Westview Press, Cambridge, MA.

Von Neumann, J. and Burks, A. (1966). Theory of Self-Reproducing Automata. University of Illinois, Urbana-Champaign.

White, T. (2012). *Hadoop: The Definitive Guide*. O'Reilly, Sebastopol, CA.

Wick, M.. (2005). Teaching design patterns in CS1: A closed laboratory sequence based on the game of life. *SIGCSE Bull.* 37:487-491.

Wuensche, A. (2011). *Exploring Discrete Dynamics; The DDLab Manual*. Luniver Press, Frome, UK.

Xia, H., Dail, H., Casanova, H., and Chien, A. (2004). The microgrid: Using online simulation to predict application performance in diverse grid network environments. In *Challenges of Large Applications in Distributed Environments*, pages 52-61. IEEE Computer Society, Los Alamitos, CA.

Zhang, Y., Gao, Q., Gao, L., and Wang, C. (2012). iMapReduce: A distributed computing framework for iterative computation. *Journal of Grid Computing*, 10:47-68.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/ecal2013/25/284/1901671/978-0-262-31709-2-ch043.pdf by guest on 10 December 2024