Most Common Words – A cP Systems Solution

Radu Nicolescu

Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand
r.nicolescu@auckland.ac.nz

Abstract. Finding the most common words in a text file is a famous "programming pearl", originally posed by Jon Bentley (1984). Several interesting solutions have been proposed by Knuth (an exquisite model of literate programming, 1986), McIlroy (an engineering example of combining a timeless set of tools, 1986), Hanson (an alternate efficient solution, 1987). Here we propose a concise efficient solution based on the fast parallel and associative capabilities of cP systems. We also check their parallel sorting capabilities and propose a dynamic version of the classical pigeonhole algorithm.

Keywords: Literate programming, most common words, membrane computing, P systems, cP systems, associative data structures, inter- and intra-cell parallelism, Prolog terms and unification, parallel sorting, pigeonhole algorithm.

1 Introduction and Background

cP systems share the fundamental features of the traditional cell-like (tree-based) and tissue (graph-based) P systems: top-cells are organised in graph/digraph networks, top-cells contain nested (and labelled) sub-cells, the evolution is governed by multiset rewriting rules, possibly running in maximal parallel modes.

Although not strictly necessary – but also shared with other versions of the traditional P systems – our typical rulesets are state based and run in a weak priority mode.

There are two main innovations in cP systems. First, unlike in traditional cell-like P systems, sub-cells do NOT have own rules. Basically, the sub-cells are just nested passive repositories of other sub-cells or atomic symbols; therefore, they can also be viewed as nested complex objects (or terms).

This seems a severe limitation. However, it is more than compensated by the provision of higher level rules, which extend the classical multiset rewriting rules with concepts borrowed from logic programming, namely Prolog unification. In other words, cP systems may be seen as adapting the classical Prolog unification from structured terms to multisets – which again is a novel feature.

However, unlike traditional Prolog, where rules are applied in a backward-chaining mode, with possible backtracks, cP rules work in a forward mode, like all known P system rules. This may perhaps allow better parallelism capabilities

than the past and actual parallel versions of Prolog – but this topic will not be further followed here.

The net result is a powerful system which can crisply and efficiently solve complex problems, with small fixed-size alphabets and small, fast fixed-size rulesets. In particular, cP systems enable a reasonably straightforward creation and manipulation of high-level data structures typical of high-level languages, such as: numbers, relations (graphs), associative arrays, lists, trees, strings.

In this sense, cP systems have been successfully used to develop parallel and distributed models in a large variety of domains, such as distributed algorithms, graph theory, image processing, NP complete problems.

In this paper, we further assess the "computer science" capabilities of our cP systems by solving a version of a famous *programming pearl*, initially posed by Jon Bentley (1984): printing the most common words in a text file, more precisely (but still a bit vague) [1]:

> Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency.

Additionally, the integer N is typically used for the number of words, d is the number of distinct words, and f is the highest frequency count. Of course, one typically assumes that $N > d > k$ and $N - d + 1 \geq f \geq N/d$, but some solutions are optimised for the more special case $N \gg d \gg k$.

Several interesting solutions have been proposed by Knuth in 1986 – an exquisite model of literate programming [1], McIlroy in 1986 – an engineering example of combining a timeless set of tools [1], Hanson in 1987 – an alternate efficient solution [12]. All these three solutions can be considered as great literate programming sample models, if we take "literal programming" in a generic sense – not just Knuth's WEB/TANGLE implementation [2].

Here we propose a concise efficient solution, following Hanson's revised formulation [12] of the original problem specification, which clarifies the slight ambiguity of the original:

> Given a text file and an integer k, you are to print the words (and their frequencies of occurrence) whose frequencies of occurrence are among the k largest in order of decreasing frequency.

A tiny but artificial example may clarify these specifications. Assume that $k = 2$ and the input text is:

```
ccc aa aa aa ccc bb d aa d
```

Note that, here, $N = 9, d = 4, f = 4$. Bentley's original formulation, used by Knuth and McIlroy [1], essentially requires – a bit ambiguously – one of the following two outputs:

```
4 aa
2 ccc
```

or

```
4 aa
2 d
```

In contrast, Hanson's revised formulation [12], requires the following output – which is unambiguous, if the order of word sublists is not relevant (i.e. ccc d ≡ d ccc):

```
4 aa
2 ccc d
```

Schematically, all these there solutions follow *four main phases*: (I) reading and splitting the text file into words (parsing it); (II) computing the word frequencies; (III) sorting according to frequencies; and (IV) printing the required output.

Knuth and Hanson provide large *monolithic* solutions, which include all four phases. Moreover, they combine phases I and II, by using associative data structures: Knuth uses a custom hash-trie and Hanson a custom hashtable with splay (move to front) lists. For phase III, both authors try to use efficient sorting methods. Knuth uses a fast sorting method, assuming that $N \gg d \gg k$ and that most frequent words tend to appear from the beginning of the text – however, as McIlroy points out, this does not always hold. Hanson offers a more universal fast sorting method based on the *pigeonhole algorithm*, with f holes.

McIlroy's solution is a textbook example for the *separation-of-concerns* principle, via a pipeline of staple general-purpose utilities initially developed for UNIX. Each of the four phases is implemented via just one or two commands. Together, phases II and III take exactly three lines in the pipe [1]:

```
(3) sort |
(4) uniq -c |
(5) sort -rn |
```

Line (3) sorts the N input words (lexicographically). Line (4) counts then discards the duplicates, keeping d unique exemplars and their frequency counts (as count/word pairs). Line (5) sorts d count/word pairs, in reverse count order (numerically).

Intentionally not given here are pipe lines (1), (2) and (6), which deal with phases I and IV. Reading, splitting into words and printing can be defined in a seemingly endless multiplicity of ways, which may not be worth discussing here. In particular, the concept of "word" itself may be highly interpretable: does it include ASCII letters, UNICODE letters, digits, punctuation signs, does it have a length limit, etc. Here, we will stay away from this discussion.

McIlroy's solution is also reasonably fast – not as fast as the other two – but it is extremely crisp and clear, and can be flexibly adapted to other input and output formats. Such a solution can be developed and deployed in just a few minutes – this sounds amazing, but does not account for the many man-months

required to develop and tune the used building blocks (UNIX tools). McIlroy also notes that his solution could be sped up by replacing the more costlier lines (3) and (4) by a hypothetical tool based on associative arrays – in fact, this would bring his solution closer to Hanson's solution for phase II.

Our cP solution – which uses *one single top-level cell with data-only subcells* – follows the spirit of McIlroy's and Hanson's solutions. It is based on associative data types and a sorting idea close to Hanson's pigeonhole algorithm. It also uses a small fixed number of rules – close to McIlroy's pipeline size – but, in contrast, it is built from scratch (not on higher building block as the UNIX commands).

We offer two alternate solutions: (i) a solution which solves Hanson's version of the problem – where the result is a *sorted sequence of word multisets*; and (ii) a solution which solves the original problem, as posed by Bentley and used by Knuth and McIlroy – where the result is a *sorted sequence of words*.

In this process, we propose and use a *dynamic pigeonhole algorithm*, adaptable to other platforms with strong associative capabilities, where – metaphorically - pigeonholes are only opened one at at time, instantly attracting objects with matching keys.

In our case, we must first adapt the above problem formulation to typical P systems, where cells contain multisets of symbols, not ordered structures. What is a sorted multiset? Ordered structures must be constructed in terms of multisets – in cP systems, we can create the required high-level structures by deep nesting of complex symbols (subcells).

As above mentioned, we chose to skip over the reading phase (I) and we assume that all words are "magically" present at start-time in our single cell. Our focus is on phases II and III, where all operations are clearly defined and can be efficiently performed by cP systems.

Finally – as used in our first solution (i) – we simulate the printing phase IV, by *sequentially sending out the required results, in order, over a designated line*. Alternatively – as used in our second solution (ii) – we actually *build an ordered list containing the required results*.

For completeness, Section 2 introduces a few high-level data structures in cP systems and Appendix A offers a more complete definition of the cP systems – both these sections incrementally update the results and definitions given in our earlier paper [7]. The remaining sections discuss our solution.

2 Data structures in cP systems

We assume that the reader is familiar with the membrane extensions collectively known as *complex symbols*, proposed by Nicolescu et al. [8, 9, 6]. However, to ensure some degree of self-containment, our revised extensions, (still) called cP systems, are reproduced in Appendix A.

In this section we sketch the design of high-level data structures, similar to the data structures used in high-level pseudocode or high-level languages: numbers, relations, functions, associative arrays, lists, trees, strings, together with alternative more readable notations.

Natural numbers. Natural numbers can be represented via *multisets* containing repeated occurrences of the *same* atom. For example, considering that *1* represents an ad-hoc unary digit, the following complex symbols can be used to describe the contents of a virtual integer *variable* a: $a() = a(\lambda)$ — the value of a is 0; $a(1^3)$ — the value of a is 3. For concise expressions, we may alias these number representations by their corresponding numbers, e.g. $a() \equiv a(0), b(1^3) \equiv b(3)$. Nicolescu et al. [8, 9] show how the basic arithmetic operations can be efficiently modelled by P systems with complex symbols.

Here follows a list of simple arithmetic expressions, assignments and comparisons:

$$
\begin{array}{l}
x = 0 \;\equiv\; x(\lambda) \\
x = 1 \;\equiv\; x(1) \\
x = 2 \;\equiv\; x(11) \\
x = n \;\equiv\; x(1^n) \\
x \leftarrow y + z \;\equiv\; y(Y)\; z(Z) \;\rightarrow\; x(YZ) \;\; \textit{destructive add} \\
x \leftarrow y + z \;\equiv\; \rightarrow\; x(YZ) \;\mid\; y(Y)\; z(Z) \;\; \textit{preserving add} \\
x = y \;\equiv\; x(X)\; y(X) \\
x \leq y \;\equiv\; x(X)\; y(XY) \\
x < y \;\equiv\; x(X)\; y(X1Y)
\end{array}
$$

Relations and functions. Consider the *binary relation* r, defined by: $r = \{(a, b), (b, c), (a, d), (d, c)\}$ (which has a diamond-shaped graph). Using complex symbols, relation r can be represented as a *multiset* with four r items, $\{r(\kappa(a)\ \upsilon(b)), r(\kappa(b)\ \upsilon(c)), r(\kappa(a)\ \upsilon(d)), r(\kappa(d)\ \upsilon(c))\}$, where ad-hoc atoms κ and υ introduce *domain* and *codomain* values (respectively). We may also alias the items of this multiset by a more expressive notation such as: $\{(a \overset{r}{\rightleftarrows} b), (b \overset{r}{\rightleftarrows} c), (a \overset{r}{\rightleftarrows} d), (d \overset{r}{\rightleftarrows} c)\}$.

If the relation is a *functional relation*, then we can emphasise this by using another operator, such as "mapsto". For example, the functional relation $f = \{(a, b), (b, c), (d, c)\}$ can be represented by multiset $\{f(\kappa(a)\ \upsilon(b)), f(\kappa(b)\ \upsilon(c)), f(\kappa(d)\ \upsilon(c))\}$ or by the more suggestive notation: $\{(a \overset{f}{\mapsto} b), (b \overset{f}{\mapsto} c), (d \overset{f}{\mapsto} c)\}$. To highlight the actual mapping value, instead of $a \overset{f}{\mapsto} b$, we may also use the succinct abbreviation $f[a] = b$.

In this context, the $\rightleftarrows$ and $\mapsto$ operators are considered to have a high associative priority, so the enclosing parentheses are mostly used for increasing the readability.

Associative arrays. Consider the *associative array* x, with the following key-value mappings (i.e. functional relation): $\{1 \mapsto a; 1^3 \mapsto c; 1^7 \mapsto g\}$. Using complex symbols, array x can be represented as a multiset with three items, $\{x(\kappa(1)\,\upsilon(a)), x(\kappa(1^3)\,\upsilon(c)), x(\kappa(1^7)\,\upsilon(g))\}$, where ad-hoc atoms κ and υ introduce keys and values (respectively). We may also alias the items of this multiset by the more expressive notation $\{1 \overset{x}{\mapsto} a,\ 1^3 \overset{x}{\mapsto} c,\ 1^7 \overset{x}{\mapsto} g\}$.

Lists. Consider the *list* y, containing the following sequence of values: $[u; v; w]$. List y can be represented as the complex symbol $y(\,\gamma(u\ \gamma(v\ \gamma(w\ \gamma()))))$, where

the ad-hoc atom γ represents the list constructor *cons* and $\gamma()$ the empty list. We may also alias this list by the more expressive equivalent notation $y(u \,|\, v \,|\, w)$ – or by $y(u \,|\, y')$, $y'(v \,|\, w)$ – where operator | separates the head and the tail of the list. The notation $z(|)$ is shorthand for $z(\gamma())$ and indicates an empty list, z.

Trees. Consider the *binary tree* z, described by the structured expression $(a,(b),(c,(d),(e)))$, i.e. z points to a root node which has: (i) the value a; (ii) a left node with value b; and (iii) a right node with value c, left leaf d, and right leaf e. Tree z can be represented as the complex symbol $z(a\ \phi(b)\ \psi(c\ \phi(d)\ \psi(e)))$, where ad-hoc atoms ϕ, ψ introduce left subtrees, right subtrees (respectively).

Strings. Consider the *string* $s =$ "*abc*", where a, b, and c are atoms. Obviously, string s can interpreted as the list $s = [a; b; c]$, i.e. string s can be represented as the complex symbol $s(\,\gamma(a\ \gamma(b\ \gamma(c\ \gamma()))))$, etc.

3 The parallel cP algorithm – solution (i)

3.1 Initial state

We need *one single cell* with one designated output line. Required data structures are built as complex symbols (data-only subcells), using the interpretations and notations defined in Section 2. In particular, the N input words are strings built via functor w; these complex symbols are already extant when the systems starts. Figure 1 illustrates the initial cell contents for the sample given in Section 1.

"*ccc*" "*aa*" "*aa*" "*aa*" "*ccc*" "*bb*" "*d*" "*aa*" "*d*"

(a) High-level strings.

$w(c\,w(c\,w(c\,w())))$ $w(a\,w(a\,w()))$ $w(a\,w(a\,w()))$ $w(a\,w(a\,w()))$
$w(c\,w(c\,w(c\,w())))$ $w(b\,w(b\,w()))$ $w(d\,w())$ $w(a\,w(a\,w()))$ $w(d\,w())$

(b) Underlying complex symbols.

Fig. 1: Sample initial word multiset.

3.2 Phase II

Using an associative relation, α, each word is tagged with an initial "frequency" count of 1 and then we *merge all word duplicates* and *sum* their associated counts. In the end, we get d words, each one with its actual frequency count.

Figure 2 shows the three rules for phase II. This ruleset starts in state S_0. Rule (0) establishes relation α between extant strings given by $w(X)$ and the initial frequency count 1; it runs in `max` mode, so it completes its job in 1 cP step.

Rule (1) repeatedly merges word duplicates and sums their associated counts; it runs in `max` mode, so it completes its job in $\log(d)$ cP steps – this rule is non-deterministic but confluent.

After rule (1) completes, rule (2) moves to the final state of this ruleset, S_2. Table 1 illustrates the evolution of the cell contents for our initial sample.

$$
\begin{array}{llll}
S_0 & w(W) & \rightarrow_{\texttt{max}} & S_1\ \alpha(w(W)\, f(1)) \qquad (0) \\
S_1 & \alpha(w(W)\, f(F))\ \ \alpha(w(W)\, f(F')) & \rightarrow_{\texttt{max}} & S_1\ \alpha(w(W)\, f(FF')) \qquad (1) \\
S_1 & & \rightarrow_{\texttt{min}} & S_2 \qquad (2)
\end{array}
$$

Fig. 2: Ruleset for phase II.

Apply	State	Cell contents
(0)	S_0	“*ccc*” “*aa*” “*aa*” “*aa*” “*ccc*” “*bb*” “*d*” “*aa*” “*d*”
(1)	S_1	α(“*ccc*” $f(1)$) α(“*aa*” $f(1)$) α(“*aa*” $f(1)$) α(“*aa*” $f(1)$) ...
(1)	S_1	α(“*ccc*” $f(2)$) α(“*aa*” $f(2)$) α(“*aa*” $f(2)$) α(“*bb*” $f(1)$) α(“*d*” $f(2)$)
(2)	S_1	α(“*ccc*” $f(2)$) α(“*aa*” $f(4)$) α(“*bb*” $f(1)$) α(“*d*” $f(2)$)
–	S_2	α(“*ccc*” $f(2)$) α(“*aa*” $f(4)$) α(“*bb*” $f(1)$) α(“*d*” $f(2)$)

Table 1: Phase II evolution of the sample word multiset.

3.3 Phase III

We create maximal word multisets by merging all words sharing the same *frequency counts*.

Figure 3 shows the two rules for phase III. This ruleset starts in state S_2, the final state for phase II (3.2). Rule (3) merges word multisets sharing the same frequency counts; it runs in `max` mode, so it completes its job in $\log(f)$ cP steps – this rule is non-deterministic but confluent.

After rule (3) completes, rule (4) moves to the final state of this ruleset, S_3. Table 2 illustrates the evolution of the cell contents for the initial sample.

$$S_2 \quad \alpha(W\, f(F)) \quad \alpha(W'\, f(F)) \rightarrow_{\texttt{max}} \quad S_2 \; \alpha(W\, W'\, f(F)) \quad (3)$$
$$S_2 \qquad\qquad\qquad\qquad \rightarrow_{\texttt{min}} \quad S_3 \qquad\qquad\qquad\quad (4)$$

Fig. 3: Ruleset for phase III.

Apply	State	Cell contents
(3)	S_2	$\alpha(\text{“}ccc\text{”}\ f(2))\ \ \alpha(\text{“}aa\text{”}\ f(4))\ \ \alpha(\text{“}bb\text{”}\ f(1))\ \ \alpha(\text{“}d\text{”}\ f(2))$
(4)	S_2	$\alpha(\text{“}ccc\text{”}\ \text{“}d\text{”} f(2))\ \ \alpha(\text{“}aa\text{”}\ f(4))\ \ \alpha(\text{“}bb\text{”}\ f(1))$
–	S_3	$\alpha(\text{“}ccc\text{”}\ \text{“}d\text{”} f(2))\ \ \alpha(\text{“}aa\text{”}\ f(4))\ \ \alpha(\text{“}bb\text{”}\ f(1))$

Table 2: Phase III evolution of the sample word multiset.

3.4 Phase IV

We send out all existing word multisets, sequentially, in decreasing order of their *frequency counts*. We propose and use a *dynamic* version of the classical *pigeonhole algorithm* (adaptable to other platforms with strong associative capabilities), where – metaphorically - pigeonholes are only opened one at at time, instantly attracting objects with matching keys.

First, we determine the highest frequency count. Next, we repeatedly output the word multiset having the current highest frequency count – if any – and then decrement this count, until we reach 0. This current highest frequency count is the "enabled pigeonhole" which "attracts" the word multiset having the same frequency count. For simplicity, we do not consider the parameter k, but it is straightforward to include it in this ruleset.

Figure 4 shows the rules for phase IV. This ruleset starts in state S_3, the final state for phase III (3.3). Rule (5) extracts frequency counts; it runs in `max` mode, so it completes its job in 1 cP steps.

Rule (6) determines the highest frequency count by taking pairwise maximums (note that all extant frequency counts are different); it runs in `max` mode, so it completes its job in $\log(f)$ cP steps – this rule is non-deterministic but confluent.

After rule (6) completes, rule (7) moves to the next state of this ruleset, S_5. Rule (8) outputs the word multiset having the current (highest) non-zero frequency count – if any – and then decrements this count; rule (9) just decrements this count, if there is no matching word multiset; this pair of rules complete their job in $\log(f)$ cP steps.

After all the word multisets are sent out, the cell remains idle in the final state, S_5 – alternatively, one more rule could clear the remaining $f(0)$ counter and transit to another state (e.g. S_6). Table 3 illustrates the evolution of the

cell contents for the initial sample. Essentially, in this scenario we output the sequence [(“aa”, 4); (“ccc” “d”, 2); (“bb”, 1)],

$$
\begin{array}{llll}
S_3 & \alpha(W\, f(F)) & \to_{\texttt{max}} & S_4\ \alpha(W\, f(F))\ f(F) \quad (5)\\
S_4 & f(F)\ \ f(F1F') & \to_{\texttt{max}} & S_4\ f(F1F') \quad (6)\\
S_4 & & \to_{\texttt{min}} & S_5 \quad (7)\\
S_5 & \alpha(W\, f(F1))\ \ f(F1) & \to_{\texttt{min}} & S_5\ \alpha(W\, f(F1))\downarrow\ f(F) \quad (8)\\
S_5 & f(F1) & \to_{\texttt{min}} & S_5\ f(F) \quad (9)
\end{array}
$$

Fig. 4: Ruleset for phase IV.

Apply	State	Cell contents
(5)	S_3	$\alpha(\text{“}ccc\text{”}\ \text{“}d\text{”}\, f(2))\ \ \alpha(\text{“}aa\text{”}\ f(4))\ \ \alpha(\text{“}bb\text{”}\ f(1))$
(6)	S_4	$\alpha(\text{“}ccc\text{”}\ \text{“}d\text{”}\, f(2))\ \ \alpha(\text{“}aa\text{”}\ f(4))\ \ \alpha(\text{“}bb\text{”}\ f(1))\ \ f(2)\ \ f(4)\ \ f(1)$
(6)	S_4	$\alpha(\text{“}ccc\text{”}\ \text{“}d\text{”}\, f(2))\ \ \alpha(\text{“}aa\text{”}\ f(4))\ \ \alpha(\text{“}bb\text{”}\ f(1))\ \ f(4)\ \ f(1)$
(7)	S_4	$\alpha(\text{“}ccc\text{”}\ \text{“}d\text{”}\, f(2))\ \ \alpha(\text{“}aa\text{”}\ f(4))\ \ \alpha(\text{“}bb\text{”}\ f(1))\ \ f(4)$
(8)	S_5	$\alpha(\text{“}ccc\text{”}\ \text{“}d\text{”}\, f(2))\ \ \alpha(\text{“}\mathbf{aa}\text{”}\ \mathbf{f(4)})\ \ \alpha(\text{“}bb\text{”}\ f(1))\ \ f(4)$
(9)	S_5	$\alpha(\text{“}ccc\text{”}\ \text{“}d\text{”}\, f(2))\ \ \alpha(\text{“}bb\text{”}\ f(1))\ \ f(3)$
(8)	S_5	$\alpha(\text{“}\mathbf{ccc}\text{”}\ \text{“}\mathbf{d}\text{”}\mathbf{f(2)})\ \ \alpha(\text{“}bb\text{”}\ f(1))\ \ f(2)$
(8)	S_5	$\alpha(\text{“}\mathbf{bb}\text{”}\ \mathbf{f(1)})\ \ f(1)$
–	S_5	$f(0)$

Table 3: Phase IV evolution of the sample word multiset – each time it is applied, the highlighted rule (8) outputs one word multiset and its associated frequency count.

4 The parallel cP algorithm – alternate solution (ii)

Here we sketch an alternate implementation, which actually builds a *sorted list of words*, ordered on their frequency counts. This solution could be applied to get a sorted list of word multisets, but here we use it to get a *sorted list of words*,

i.e. a result closer to the original problem formulation posed by Bentley and used by Knuth and McIlroy [1].

Conceptually, we start from the interim results of phase II of solution (i) (3.2), but this time we give a complete solution (not explicitly split into phases).

We create a list of words, sorted in decreasing order of their *frequency counts*. As in the earlier phase II (3.2) each word is tagged with an initial "frequency" count of 1 and then we *merge all word duplicates* and *sum* their associated counts. In the end, we get d words, each one with its actual frequency count.

Then, as in the earlier phase IV (3.4), we use a *dynamic* version of the classical *pigeonhole algorithm*, but this time we stack the "attracted" words in a result list (instead of sending them out).

First, we "enable a pigeonhole" for frequency 1 and create an empty result list. Next, we repeatedly stack all words having the current pigeonhole frequency count – if any – and then increment this count, until we exhaust all extant words. For simplicity, we again do not consider the parameter k, but it is straightforward to include it in this ruleset.

Figure 5 shows all rules for this alternate solution. Rules (0) and (1) are exactly as in the earlier phase II. Rule (2) is modified: to "enable a pigeonhole" for frequency 1 and to create an empty result list, ρ.

Rule (3) repeatedly stacks onto ρ all words having the current frequency count – if any; the standalone f acts as a promoter. Rule (4) increments this frequency count, if there are no (more) matching words for this count, but there are still other words to process; any extant $\alpha(...)$ acts as a promoter. The rules pair (3) and (4) complete their job in $\log(f)$ cP steps.

After all the words are stacked, the cell remains idle in the final state, S_2. The evolution is non-deterministic, which exactly corresponds to the slight vagueness of the original problem formulation. Table 4 illustrates a possible evolution of the cell contents for the initial sample. Essentially, in this scenario we obtain the list [("aa", 4); ("d" 2); ("ccc" 2); ("bb", 1)], but we could have also obtained the list [("aa", 4); ("ccc" 2); ("d" 2); ("bb", 1)].

5 Reflections and open problems

Both our solutions seem to have an optimal *runtime complexity*, or close to it, essentially $\mathcal{O}(\log(d)+\log(f))$ cP steps, which, in the worst case, is $\mathcal{O}(\log(N))$, but typically is much smaller. This optimality is not proven, but seems a believable hypothesis.

Also, our solutions seem to have a very decent *static complexity*, comparable to the the best known solution in this regard, proposed by McIlroy: 10 or 5 rules – in our two solutions – vs. 4 lines – the combination of 4 powerful UNIX commands in McIlroy's excellent solution. Moreover, in contrast to this, our solutions are build from "scratch" (including the associative sorting!), not on other complex utilities. Also, as presented, McIlroy's solution runs in $\mathcal{O}(N\log(N))$ steps (because of the initial sorting), which makes it slower than ours. In all fairness, McIlroy mentions potential speed-ups, but these do not seem yet available.

$$
\begin{array}{llll}
S_0 & w(W) & \to_{\texttt{max}} & S_1\ \alpha(w(W)\, f(1)) \quad (0)\\
S_1 & \alpha(w(W)\, f(F))\ \ \alpha(w(W)\, f(F')) & \to_{\texttt{max}} & S_1\ \alpha(w(W)\, f(FF')) \quad (1)\\
S_1 & & \to_{\texttt{min}} & S_2\ f(1)\ \rho() \quad (2)\\
S_2 & \alpha(w(W)\, f(F))\ \ \rho(R) & \to_{\texttt{max}} & S_2\ \rho(\alpha(w(W)\, f(F))\, \rho(R)) \quad (3)\\
 & & & \mid f(F)\\
S_2 & f(F) & \to_{\texttt{min}} & S_2\ f(F1) \quad (4)\\
 & & & \mid \alpha(\_)
\end{array}
$$

Fig. 5: Ruleset for alternate solution (ii).

Apply	State	Cell contents
(0)	S_0	“*ccc*” “*aa*” “*aa*” “*aa*” “*ccc*” “*bb*” “*d*” “*aa*” “*d*”
(1)	S_1	α(“*ccc*” $f(1)$) α(“*aa*” $f(1)$) α(“*aa*” $f(1)$) α(“*aa*” $f(1)$) ...
(1)	S_1	α(“*ccc*” $f(2)$) α(“*aa*” $f(2)$) α(“*aa*” $f(2)$) α(“*bb*” $f(1)$) α(“*d*” $f(2)$)
(2)	S_1	α(“*ccc*” $f(2)$) α(“*aa*” $f(4)$) α(“*bb*” $f(1)$) α(“*d*” $f(2)$)
(3)	S_2	$f(1)$ α(“*ccc*” $f(2)$) α(“*aa*” $f(4)$) α(“*bb*” $f(1)$) α(“*d*” $f(2)$) $\rho()$
(4)	S_2	$f(1)$ α(“*ccc*” $f(2)$) α(“*aa*” $f(4)$) α(“*d*” $f(2)$) $\rho(\alpha$(“*bb*” $f(1)$) $\rho())$
(3)	S_2	$f(2)$ α(“*ccc*” $f(2)$) α(“*aa*” $f(4)$) α(“*d*” $f(2)$) $\rho(\alpha$(“*bb*” $f(1)$) $\rho())$
(3)	S_2	$f(2)$ α(“*aa*” $f(4)$) α(“*d*” $f(2)$) $\rho(\alpha$(“*ccc*” $f(2)$) $\rho(\alpha$(“*bb*” $f(1)$) $\rho()))$
(4)	S_2	$f(2)$ α(“*aa*” $f(4)$) $\rho(\alpha$(“*d*” $f(2)$) $\rho(\alpha$(“*ccc*” $f(2)$) $\rho(\alpha$(“*bb*” $f(1)$) $\rho())))$
(4)	S_2	$f(3)$ α(“*aa*” $f(4)$) $\rho(\alpha$(“*d*” $f(2)$) $\rho(\alpha$(“*ccc*” $f(2)$) $\rho(\alpha$(“*bb*” $f(1)$) $\rho())))$
(3)	S_2	$f(4)$ α(“*aa*” $f(4)$) $\rho(\alpha$(“*d*” $f(2)$) $\rho(\alpha$(“*ccc*” $f(2)$) $\rho(\alpha$(“*bb*” $f(1)$) $\rho())))$
–	S_2	$f(4)$ $\rho(\alpha$(“*aa*” $f(4)$) $\rho(\alpha$(“*d*” $f(2)$) $\rho(\alpha$(“*ccc*” $f(2)$) $\rho(\alpha$(“*bb*” $f(1)$) $\rho()))))$

Table 4: Alternate solution (ii): possible evolution of the sample word multiset. Here the final result is the sorted list $[\alpha(\text{“}aa\text{”}\ f(4)); \alpha(\text{“}d\text{”}\ f(2)); \alpha(\text{“}ccc\text{”}\ f(2)); \alpha(\text{“}bb\text{”}\ f(1))]$.

In fact, these comparisons may be misleading, as our solution runs on a highly parallel engine – cP systems – while the other solutions are purely sequential. It may be interesting to evaluate other parallel solutions to this problem, including other P systems solutions, but we are not aware of any.

As earlier mentioned, cP systems rules generalise the traditional P systems rules by powerful Prolog-like unifications, but the classical Prolog unification algorithms do *not* work on multisets. More work is needed to design efficient unification algorithms which work on multisets and and scale out well on parallel architectures.

It is also interesting to note that our solutions seem to struggle a bit when they are constrained to run in a purely sequential mode, as in phase IV of solution (i), but feel more comfortable when they can unleash the parallel associative potential of cP systems, as in solution (ii).

To the best of our knowledge, this paper proposes a novel sorting algorithm, with a remarkable crisp expression: a dynamic version of the classical pigeonhole algorithm, apparently suitable for any platform with strong associative features (such as many or most versions of P systems).

Finally, as an open problem, it might be worthwhile to invest more effort into developing a real literate model for P systems and to develop a set of tools corresponding to Knuth's WEB toolset – perhaps P-WEB or cP-WEB?

References

1. Bentley, J., Knuth, D., McIlroy, D.: Programming pearls: A literate program. Commun. ACM 29(6), 471–483 (Jun 1986), http://doi.acm.org/10.1145/5948.315654
2. Knuth, D.E.: Literate programming. Comput. J. 27(2), 97–111 (May 1984), http://dx.doi.org/10.1093/comjnl/27.2.97
3. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
4. Nicolescu, R.: Parallel and distributed algorithms in P systems. In: Gheorghe, M., Păun, G., Rozenberg, G., Salomaa, A., Verlan, S. (eds.) Membrane Computing, CMC 2011, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7184, pp. 35–50. Springer Berlin / Heidelberg (2012)
5. Nicolescu, R.: Parallel thinning with complex objects and actors. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8961, pp. 330–354. Springer (2015)
6. Nicolescu, R.: Structured grid algorithms modelled with complex objects. In: Rozenberg, G., Salomaa, A., Sempere, J.M., Zandron, C. (eds.) 16th Conference on Membrane Computing (CMC16), Revised Selected Papers, Lecture Notes in Computer Science, vol. 9504, pp. 321–337. Springer (2015)
7. Nicolescu, R.: Revising the membrane computing model for byzantine agreement. In: Leporati, A., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) 17th International Conference on Membrane Computing (CMC17), Revised Selected Papers, Lecture Notes in Computer Science, vol. 10105, pp. 317–339. Springer (2016)
8. Nicolescu, R., Ipate, F., Wu, H.: Programming P systems with complex objects. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Y., Rozenberg, G., Salomaa, A. (eds.) 14th Conference on Membrane Computing, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8340, pp. 280–300. Springer (2013)

9. Nicolescu, R., Wu, H.: Complex objects for complex applications. Romanian Journal of Information Science and Technology 17(1), 46–62 (2014)
10. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
11. Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press (2000)
12. Van Wyk, C.J.: Literate programming. Commun. ACM 30(7), 583–599 (Jul 1987), http://doi.acm.org/10.1145/28569.315738

A Appendix
cP Systems : P Systems with Complex Symbols

We present the details of our cP framework, simplified from our earlier papers [5, 6].

A.1 Complex symbols as subcells

Complex symbols or *subcells*, play the roles of cellular micro-compartments or substructures, such as organelles, vesicles or cytoophidium assemblies ("snakes"), which are embedded in cells or travel between cells, but without having the full processing power of a complete cell. In our proposal, *subcells* represent nested labelled data compartments which have no own processing power: they are acted upon by the rules of their enclosing cells.

Our basic vocabulary consists of *atoms* and *variables*, collectively known as *simple symbols*. *Complex symbols* are similar to Prolog-like *first-order terms*, recursively built from *multisets* of atoms and variables. Together, complex symbols and simple symbols (atoms, variables) are called *symbols* and can be defined by the following formal grammar:

```
<symbol> ::= <atom> | <variable> | <term>
<term> ::= <functor> '(' <argument> ')'
<functor> ::= <atom>
<argument> ::= λ | ( <symbol> )+
```

Atoms are typically denoted by lower case letters (or, occasionally, digits), such as a, b, c, *1*. *Variables* are typically denoted by uppercase letters, such as X, Y, Z. *Functors* are term (subcell) labels; here functors can only be atoms, not variables.

For improved readability, we also consider *anonymous variables*, which are denoted by underscores ("_"). Each underscore occurrence represents a *new* unnamed variable and indicates that something, in which we are not interested, must fill that slot.

Symbols that do *not* contain variables are called *ground*, e.g.:

– Ground symbols: a, $a(\lambda)$, $a(b)$, $a(bc)$, $a(b^2c)$, $a(b(c))$, $a(bc(\lambda))$, $a(b(c)d(e))$, $a(b(c)d(e))$, $a(b(c)d(e(\lambda)))$, $a(bc^2d)$.

- Symbols which are not ground: X, $a(X)$, $a(bX)$, $a(b(X))$, $a(XY)$, $a(X^2)$, $a(XdY)$, $a(Xc())$, $a(b(X)d(e))$, $a(b(c)d(Y))$, $a(b(X^2)d(e(Xf^2)))$; also, using anonymous variables: $\_$, $a(b\_)$, $a(X\_)$, $a(b(X)d(e(\_)))$.
- This term-like construct which starts with a variable is not a symbol (this grammar defines first-order terms only): $X(aY)$.

Note that we may abbreviate the expression of complex symbols by removing inner λ's as explicit references to the empty multiset, e.g. $a(\lambda) = a()$.

In *concrete* models, *cells* may contain *ground* symbols only (no variables). Rules may however contain *any* kind of symbols, atoms, variables and terms (whether ground and not).

Unification. All symbols which appear in rules (ground or not) can be (asymmetrically) *matched* against *ground* terms, using an ad-hoc version of *pattern matching*, more precisely, a *one-way first-order syntactic unification* (one-way, because cells may not contain variables). An atom can only match another copy of itself, but a variable can match any multiset of ground terms (including λ). This may create a combinatorial *non-determinism*, when a combination of two or more variables are matched against the same multiset, in which case an arbitrary matching is chosen. For example:

- Matching $a(b(X)fY) = a(b(cd(e))f^2g)$ deterministically creates a single set of unifiers: $X, Y = cd(e), fg$.
- Matching $a(XY^2) = a(de^2f)$ deterministically creates a single set of unifiers: $X, Y = df, e$.
- Matching $a(b(X)c(1X)) = a(b(1^2)c(1^3))$ deterministically creates one single unifier: $X = 1^2$.
- Matching $a(b(X)c(1X)) = a(b(1^2)c(1^2))$ fails.
- Matching $a(XY) = a(df)$ non-deterministically creates one of the following four sets of unifiers: $X, Y = \lambda, df$; $X, Y = df, \lambda$; $X, Y = d, f$; $X, Y = f, d$.

A.2 High-level or generic rules

Typically, our rules use *states* and are applied top-down, in the so-called *weak priority* order.

Pattern matching. Rules are matched against cell contents using the above discussed *pattern matching*, which involves the rule's left-hand side, promoters and inhibitors. Moreover, the matching is *valid* only if, after substituting variables by their values, the rule's right-hand side contains ground terms only (so *no* free variables are injected in the cell or sent to its neighbours), as illustrated by the following sample scenario:

- The cell's *current content* includes the *ground term*:
  $n(a\,\phi(b\,\phi(c)\,\psi(d))\,\psi(e))$
- The following (state-less) *rewriting rule* is considered:
  $n(X\,\phi(Y\,\phi(Y_1)\,\psi(Y_2))\,\psi(Z)) \;\rightarrow\; v(X)\,n(Y\,\phi(Y_2)\,\psi(Y_1))\,v(Z)$

- Our pattern matching determines the following *unifiers*:
  $X = a$, $Y = b$, $Y_1 = c$, $Y_2 = d$, $Z = e$.
- This is a *valid* matching and, after *substitutions*, the rule's *right-hand* side gives the *new content*:
  $v(a)\ n(b\,\phi(d)\,\psi(c))\ v(e)$

Generic rules format. We consider rules of the following *generic* format (we call this format generic, because it actually defines templates involving variables):

$$\begin{aligned} &\textit{current-state}\ \ \textit{symbols}\ldots\ \ \rightarrow_{\alpha}\ \ \textit{target-state}\ \ (\textit{in-symbols})\ldots \\ &\qquad\qquad\qquad\qquad\qquad\quad (\textit{out-symbols})_{\delta}\ldots \\ &\qquad\qquad\qquad\qquad\qquad\quad \mid \textit{promoters}\ldots\ \ \neg\ \textit{inhibitors}\ldots \end{aligned}$$

Where:

- *current-state* and *target-state* are atoms or terms;
- *symbols*, *in-symbols*, *promoters* and *inhibitors* are symbols;
- *in-symbols* become available after the end of the current step only, as in traditional P systems (we can imagine that these are sent via an ad-hoc fast *loopback* channel);
- subscript $\alpha \in \{\texttt{min}, \texttt{max}\}$, indicates the application mode, as further discussed in the example below;
- *out-symbols* are sent, at the end of the step, to the cell's structural neighbours. These symbols are enclosed in round parentheses which further indicate their destinations, above abbreviated as δ. The most usual scenarios include:
  - $(a) \downarrow_i$ indicates that a is sent over outgoing arc i (unicast);
  - $(a) \downarrow_{i,j}$ indicates that a is sent over outgoing arcs i and j(multicast);
  - $(a) \downarrow_{\forall}$ indicates that a is sent over all outgoing arcs (broadcast).

  All symbols sent via one *generic rule* to the same destination form one single *message* and they travel together as one single block (even if the generic rule is applied in mode `max`).

Example. To explain our rule application mode, let us consider a cell, σ, containing three counter-like complex symbols, $c(1^2)$, $c(1^2)$, $c(1^3)$, and the two possible application modes of the following high-level "decrementing" rule:

$$(\rho_{\alpha})\ S_1\ c(1\,X) \rightarrow_{\alpha} S_2\ c(X), \text{where } \alpha \in \{\texttt{min},\texttt{max}\}.$$

The left-hand side of rule ρ_{α}, $c(1\,X)$, can be unified in three different ways, to each one of the three c symbols extant in cell σ. Conceptually, we instantiate this rule in three different ways, each one tied and applicable to a distinct symbol:

$$\begin{aligned} &(\rho_1)\ \ S_1\ c(1^2) \rightarrow S_2\ c(1), \\ &(\rho_2)\ \ S_1\ c(1^2) \rightarrow S_2\ c(1), \\ &(\rho_3)\ \ S_1\ c(1^3) \rightarrow S_2\ c(1^2). \end{aligned}$$

1. If $\alpha = \texttt{min}$, rule $\rho_{\texttt{min}}$ non-deterministically selects and applies one of these virtual rules ρ_1, ρ_2, ρ_3. Using ρ_1 or ρ_2, cell σ ends with counters $c(1)$, $c(1^2)$, $c(1^3)$. Using ρ_3, cell σ ends with counters $c(1^2)$, $c(1^2)$, $c(1^2)$.
2. If $\alpha = \texttt{max}$, rule $\rho_{\texttt{max}}$ applies in parallel all these virtual rules ρ_1, ρ_2, ρ_3. Cell σ ends with counters $c(1)$, $c(1)$, $c(1^2)$.

Special cases. Simple scenarios involving generic rules are sometimes semantically equivalent to loop-based sets of non-generic rules. For example, consider the rule

$$S_1\ a(x(I)\ y(J))\ \rightarrow_{\texttt{max}}\ S_2\ b(I)\ c(J),$$

where the cell's contents guarantee that I and J only match integers in ranges $[1, n]$ and $[1, m]$, respectively. Under these assumptions, this rule is equivalent to the following set of non-generic rules:

$$S_1\ a_{i,j}\ \rightarrow S_2\ b_i\ c_j,\ \forall i \in [1, n], j \in [1, m].$$

However, unification is a much more powerful concept, which cannot be generally reduced to simple loops.

Benefits. This type of generic rules allow (i) a reasonably fast parsing and processing of subcomponents, and (ii) algorithm descriptions with *fixed-size alphabets* and *fixed-sized rulesets*, independent of the size of the problem and number of cells in the system (often *impossible* with only atomic symbols).

Synchronous vs asynchronous. In our models, we do not make any *syntactic* difference between the synchronous and asynchronous scenarios; this is strictly a *runtime* assumption [4]. Any model is able to run on both the synchronous and asynchronous runtime "engines", albeit the results may differ. Our asynchronous model matches closely the standard definition for asynchonicity used in distributed algorithms; however, this is not needed in this paper so we don't follow this topic here.