Parallel Automata Minimization

Course in *Programming on Parallel Architectures*

Roberto Borelli
borelli.roberto@spes.uniud.it
Stefano Rocco
rocco.stefano.1@spes.uniud.it

University of Udine

September 2024

Abstract

The minimization problem of an automaton is central in automata theory and has various practical implications. In this work, we aim to develop a parallel version of the well-known Moore's algorithm, which classically runs in $O(n^2)$ time. We will review the fundamental concepts and problems in the field, analyze the serial algorithm by examining its code, theoretical properties, and time complexity. Using the OpenMP programming model, we will develop six different parallel versions of the algorithm. The first four, more efficient versions, are based on dividing the main loop into parallel tasks. The fifth version addresses the issue of merging multiple iterations of the refinement loop. The sixth and most scalable version will attempt an approach based on the parallelization of RadixSort and, ultimately, CountingSort, which will then be further developed in CUDA. We will divide CountingSort into three phases, proposing various implementation solutions for each phase using this programming model. We will test and compare the OpenMP and CUDA implementations on a significant set of instances.

Contents

Introduction

Deterministic Automata

- A **D**eterministic **F**inite state **A**utomaton is a 5-tuple $(Q, \Sigma, q_0, \delta, F)$ where
  - Q is a **finite** set of n states
  - Σ is an alphabet of m symbols
  - $q_0 \in Q$ is the initial state
  - $\delta : Q \times \Sigma \to Q$ is the transition function
  - $F \subseteq Q$ is the set of final states
- We say a (finite) word $w = w_1 \dots w_j \in \Sigma^*$ is **accepted** by $\mathcal{A}$ if $\delta(\dots\delta(\delta(q_0, w_1), w_2)\dots, w_j) = \hat{\delta}(q_0, w) \in F$
- The **language accepted** by $\mathcal{A}$ is the set $\mathcal{L}(\mathcal{A}) = \left\{ w \in \Sigma^* : \hat{\delta}(q_0, w) \in F \right\} \subseteq \Sigma^*$

When the context is clear, we write δ instead of $\hat{\delta}$

Accessible States

- A state p is **accessible** if it can be reached from the initial state, formally, $\exists w \in \Sigma^* \ (\delta(q_0, w) = p)$
- A state p is **co-accessible** if it can reach a final state, formally, $\exists w \in \Sigma^* \ (\delta(p, w) \in F)$
- An automaton is accessible (resp. co-accessible) if all of its states are accessible (resp. co-accessible)
- Notice that states which are not accessible or not co-accessible can be safely removed without affecting the recognized language

Myhill-Nerode Equivalence

- The Myhill-Nerode theorem implies that for each DFA $\mathcal{A}$, belonging to the class $\{\mathcal{B} : \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})\}$ there is a unique automation with the minimum number of states (modulo isomorphism)
- Let $\mathcal{A}$ be a DFA. For each pair of states $p, q \in Q$ we define the equivalence relation $\sim \subseteq Q \times Q$ as

$$p \sim q \iff \forall w \in \Sigma^*(\delta(p, w) \in F \iff \delta(q, w) \in F)$$

- If p and q belong to the same class, we say they are **indistinguishable**

The Automata Minimization Problem

Problem (DFA minimization)

Given an automaton $\mathcal{A}$, find $\mathcal{A}'$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ such that $|Q'|$ is minimum

Approaches

- Bottom Up: starting from $\{F, Q - F\}$ and refining partitions
  - Moore's algorithm runs in $\mathcal{O}(n^2)$
  - Hopcroft's algorithm runs in $\mathcal{O}(n \log n)$
- Top Down: starting from $\{\{q\} : q \in Q\}$ and merging classes
  - the best known algorithm is quadratic

More on DFA minimization

Open problem:

- It is still unknown whether or not the problem admits a linear time solution

Practical applications:

- Lexical analysis
- Hardware optimization
- Comparing languages: the minimum DFA is a fingerprint of a given language
- Checking if a language is star-free

Motivations

In the following sections, we show parallelizations of Moore's algorithm

- The serial version is easy to understand and implement
- It appears to be faster than Hopcroft's algorithm in practice (same as quicksort vs mergesort)
- Hopcroft's algorithm is less straightforward and needs specific data structures to reach the desired complexity; it also appears to be less suitable for parallelization

Moore

Theory

We give an incremental way to compute $\sim$. For each $i \in \mathbb{N}$, we define

$$p \sim_i q \iff \forall w \in \Sigma^{\leq i}(\delta(p, w) \in F \iff \delta(q, w) \in F)$$

Useful properties

1. $\sim_0$ distinguishes final states from non-final ones
2. $\sim_{i+1}$ refines $\sim_i$
3. $p \sim_{i+1} q \iff p \sim_i q \wedge \forall a \in \Sigma(\delta(p, a) \sim_i \delta(q, a))$
4. There exists $k \leq |Q|$ s.t. $\sim_k = \sim_{k+1} = \sim_{k+2} = \ \ldots \ = \sim$
5. $\sim_{i+1} \neq \sim_i \implies |Q \backslash \sim_{i+1}| > |Q \backslash \sim_i|$

The minimum DFA is the quotient auotomaton $\mathcal{A} \backslash_\sim$

$$\mathcal{A} \backslash_\sim = (Q \backslash \sim, \Sigma, [q_0], \delta^*, \{[f] : f \in F\}) \text{ where } \delta^*([q], a) = [\delta(q, a)]$$

Pseudo-Code: Main Procedure

```
1  DFA minimize(DFA A=(Q, Σ, q0, δ, F))
2    n ← |Q|, n_class ← 0, n_class′ ← 2
3    alloc int[n] tilde
4
5    forall (q ∈ Q) tilde[i] ← ((q ∈ F) ? 1 : 2)
6    while(n_class != n_class′)
7      n_class ← n_class′
8      n_class′ ← refine(tilde)
9
10   alloc DFA A′=(Q′, Σ′, q0′, δ′, F′) : Σ′ = Σ, q0′ = tilde[q0]
11   n′ ← n_class, alloc int[n] Q′{1,...,n′}
12   forall(p ∈ Q′)
13     q ← pick any q ∈ Q : tilde[q] == p
14     if (q ∈ F) F′ ← F′ ∪ {p}
15     forall (a ∈ Σ) δ′(p,a) ← tilde[δ(q,a)]
16
17   free tilde
18   return A′
```

Pseudo-Code: Refining Classes

```
1  int refine(int[] tilde : ref)
2    alloc string[n] sign
3
4    forall(q ∈ Q)
5      sign[q] = (tilde[q],tilde[δ(q,s1)],...,tilde[δ(q,sm)])
6    π ← radixSort(sign)
7
8    class ← 1
9    tilde[π[1]] ← class
10   for(i ← 2 upto n)
11     if (sign[π[i]] != sign[π[i-1]]) class++
12     tilde[π[i]] ← class
13
14   free sign, π
15   return class
```

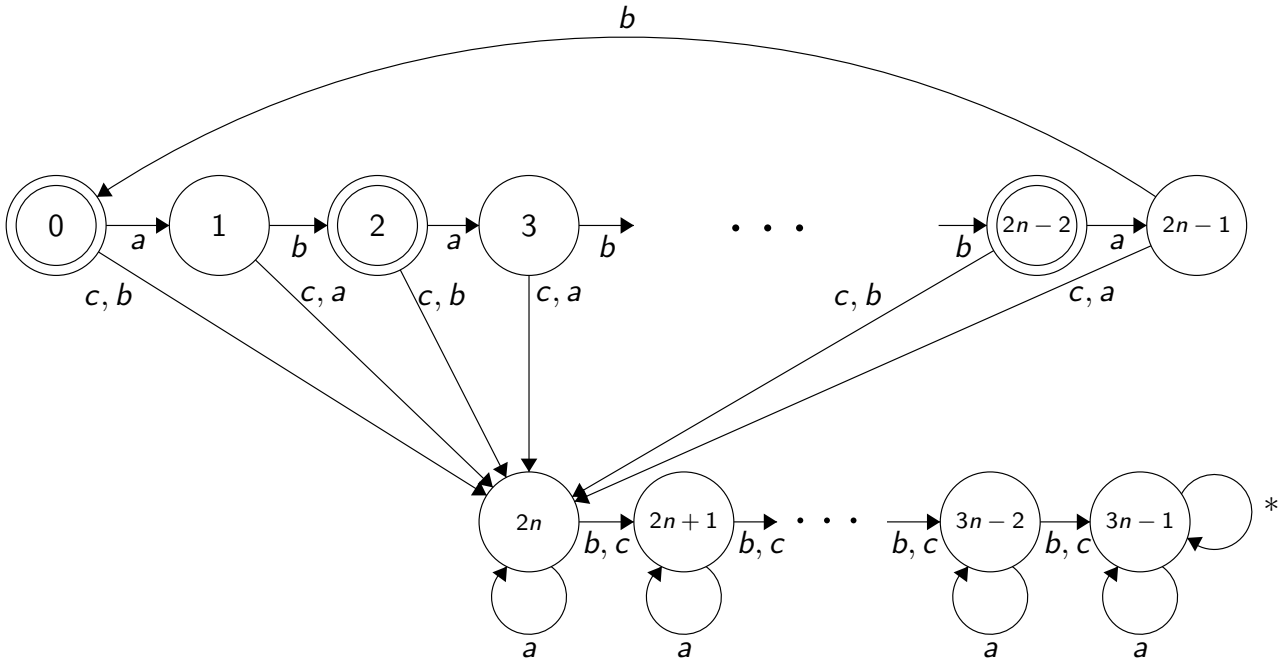
Pseudo-Code: Sorting

```
1  int[] radixSort(string[n] sign)
2    alloc int[n] π{1,...,n}
3    for(j ← 1 upto m+1) countingSort(sign, π, j)
4    return π
5
6  void countingSort(string[n] sign, int[n] π: ref, int j)
7    k ← n_classes // max value we can find in sign
8    alloc int[n] out, int[k] count
9
10   for(i ← 1 upto n)
11       index ← sign[π[i]][j]
12       count[index]++
13   for(i ← 2 upto k)
14       count[i] ← count[i] + count[i - 1]
15   for(i ← n downto 1)
16       index ← sign[π[i]][j]
17       outIndex ← --count[index]
18       out[outIndex] ← π[i]
19   copyArray(out, π)
```

Complexity Analysis: An Overview

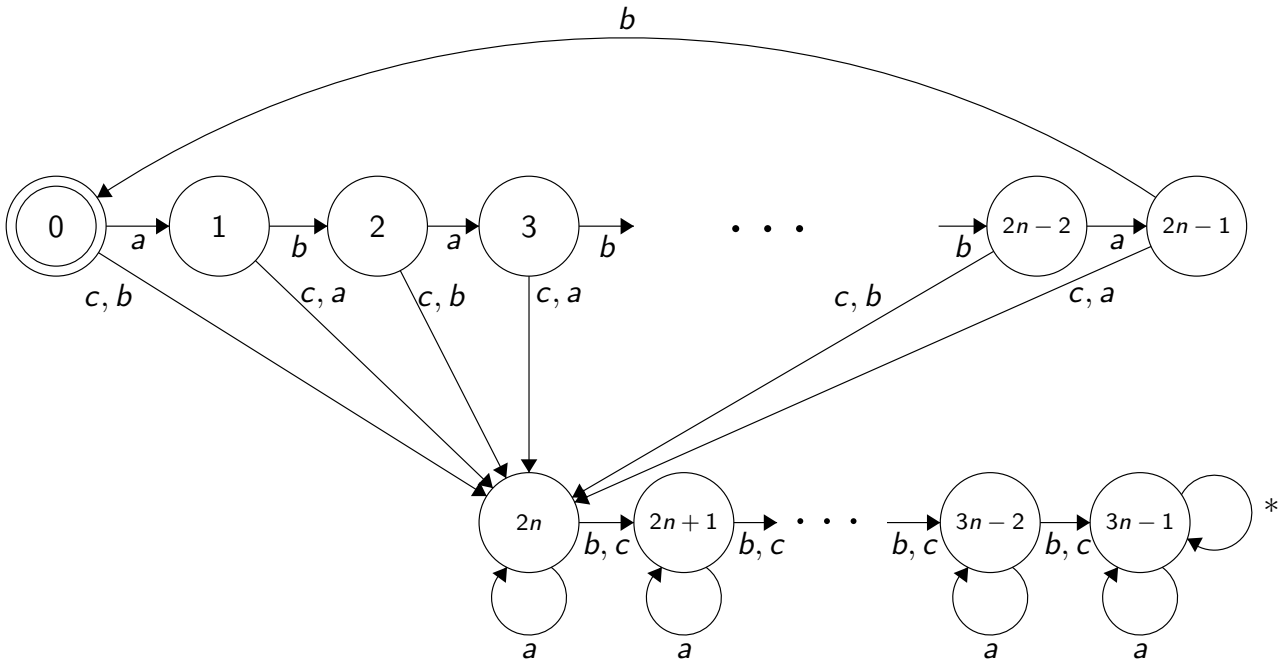
- Moore's Algorithm takes in input complete accessible DFAs
- Best case: $\mathcal{O}(mn)$ "linear"
- Worst case: $\mathcal{O}(mn^2)$ "quadratic"
- Average case: $\mathcal{O}(mn \log n)$ when F is chosen uniformly at random. see Appendix A
- If $m \geq 3$, the probability that a random accessible automaton is minimal tends to 1 as the number of states tends to infinity
- Moore's algorithm on any accessible automaton takes the same number of iterations as on the corresponding minimal automaton
- Moore's algorithm on a minimal automaton of size n has complexity $\Omega(\frac{m}{\log m} n \log \log n)$ for an alphabet of size $m \geq 2$

Complexity Analysis: Best Case Instances



- DFAs $\{\mathcal{A}_n : n \geq 1\}$ accept the language $(ab)^*$
- $3n$ states, n of them are final
- $\mathcal{A}_1$ is the minimum DFA for $(ab)^*$
- Given $\mathcal{A}_n$, $\mathcal{A}_1$ is always found in 2 iterations

Complexity Analysis: Worst Case Instances



- DFAs $\{\mathcal{B}_n : n \geq 1\}$ accept the family of languages $\{((ab)^n)^* : n \geq 1\}$
- $3n$ states, only state 0 is final
- States $2n + 1, \ldots, 3n - 1$ are not co-accessible and removing them leads to the the minimum DFA for $((ab)^n)^*$
- Given $\mathcal{B}_n$, the solution is always found in $2n$ iterations

OpenMP

Evaluation: Instances

- I take in consideration the following instances

ist1B	$\mathcal{A}_{2000000,30}$	The automaton $\mathcal{A}_{2000000}$ with $m = 30$ slide 15
ist2B	$\mathcal{B}_{5000,20}$	The automaton $\mathcal{B}_{5000}$ with $m = 20$ slide 16
ist3B	$\mathcal{C}_{2000000,30}$	A random accessible automaton with $n = 2000000$ and $m = 30$
ist1S	$\mathcal{A}_{20000000,2}$	The automaton $\mathcal{A}_{20000000}$ with $m = 2$ slide 15
ist2S	$\mathcal{B}_{15000,2}$	The automaton $\mathcal{B}_{15000}$ with $m = 2$ slide 16
ist3S	$\mathcal{C}_{20000000,2}$	A random accessible automaton with $n = 20000000$ and $m = 2$

- I run tests on 11th Gen Intel(R) Core(TM) i7-1165G7 at 2.80GHz
- I call *naive*, the first serial program
- If P is an OpenMP program, I call $P_p(n)$ the program P executed with n threads in **p**arallel and I call P_s the program P compiled without the `-fopenmp` option thus executed **s**erially

Evaluation: Statistics

program		ist1B				ist2B				ist3B			
		t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)
0	naive	27.2	/	/	/	91.3	/	/	/	39.6	/	/	/

- Let i be the index of an OpenMP program, 0 is the index of the naive one. For each test instance I show 4 statistics:
  - t Average time (in seconds) taken by program i over 4 executions
  - 0 Percentage of time saved by using program i instead of program 0
  - i_s Percentage of time saved by using program i_p instead of program i_s
  - $i–1_p$ Percentage of time saved by using program i_p instead of program $i-1_p$

1. Using for parallelism

First idea

Do in advance all the counting and compute all the prefix-sums using the `parallel for` directive.

- `count` is now a $m + 1 \times n$ matrix
- For each $1 \leq i \leq n$ and $1 \leq j \leq m + 1$, `count[j][i]` must be atomically incremented
- In the `refine` function I create signatures, initialize and fill `count` and compute prefix sums
- In the `radixSort` function I use precomputed values of `count`

1. Using for parallelism

In function `refine` we have

```
1  #pragma omp parallel num_threads(N)
2    #pragma omp for schedule(static) nowait
3    for(i ← 1 upto n)
4      index ← (sign[i][1] ← tilde[i])
5      #pragma omp atomic update
6      count[1][index]++
7
8    #pragma omp for schedule(static) collapse(2)
9    for(i ← 1 upto n)
10     for(j ← 2 upto m+1)
11       index ← (sign[i][j] ← tilde[ δ(i,j-1) ])
12       #pragma omp atomic update
13       count[j][index]++
14
15   #pragma omp for schedule(static) nowait
16   for(j ← 1 upto m+1)
17     for(i ← 2 upto k)
18       count[j][i] ← count[j][i] + count[j][i - 1]
```

1. Using for parallelism

`radixSort` becomes something like

```
1  alloc int[n] out
2  in ← π
3  for(j ← 1 upto m+1)
4    for(i ← n downto 1)
5      index ← sign[in[i]][j]
6      outIndex ← --count[j][index]
7      out[outIndex] ← in[i]
8    swap(in, out)
```

1. Using for parallelism

program		ist1B				ist2B				ist3B			
		t (s)	0 (%)	i_s (%)	$i - 1_p$ (%)	t (s)	0 (%)	i_s (%)	$i - 1_p$ (%)	t (s)	0 (%)	i_s (%)	$i - 1_p$ (%)
0	naive	27.2	/	/	/	91.3	/	/	/	39.6	/	/	/
1	for$_p$(4)	16.1	41%	14%	/	79.9	12%	-20%	/	34.7	12%	16%	/
	for$_s$	18.8	31%	/	/	66.4	27%	/	/	41.5	-5%	/	/

Problem

Most of the time is still spent on the sorting process which has not been parallelized.

2. Using task parallelism

Second idea

For each digit j we define four tasks, the fourth one being the sorting over digit j. Use the `parallel task` directive.

Let j be a digit, I define tasks

- A_j initialize `count[j]`, create `sign[i][j]` for each state i
- B_j fill `count[j][l]` for each index l
- C_j compute the prefix sum on the array `count[j]`
- D_j use the array `count[j]` to perform a `countingSort` over digit j

2. Using task parallelism

- Tasks are defined in a `for` loop inside a `single` region in the `refine` function and start to execute in parallel after the implicit barrier of the `single` region
- The dependencies are specified through arrays of integers and are as follows:
  - B_j depends on A_j
  - C_j depends on B_j
  - D_j depends on C_j and on D_{j-1}

Problem

The prefix-sum task is too light compared to the others.

Second idea (refined)

Merge the prefix sum task C_j with the counting task B_j.

2. Using task parallelism

In the `refine` function we have

```
1  #pragma omp single
2    for(j ← 1 upto m+1)
3      #pragma omp task firstprivate(j) depend(out:depSig[j])
4        for(i ← 1 upto n)
5          sign[i][j] ← tilde[ δ(i,j-1) ]
6      #pragma omp task firstprivate(j) depend(in:depSig[j])
           depend(out:depSum[j])
7        for(i ← 1 upto n)
8          count[j][ sign[i][j] ]++
9        for(i ← 2 upto k)
10         count[j][i] ← count[j][i] + count[j][i - 1]
11     #pragma omp task firstprivate(j) depend(in:depSum[j])
           depend(in:depSort[j-1]) depend(out:depSort[j])
12       for(i ← n downto 1)
13         index ← sign[in[i]][j]
14         outIndex ← --count[j][index]
15         out[outIndex] ← in[i]
16       swap(in, out)
```

2. Using task parallelism

program		ist1B				ist2B				ist3B			
		t (s)	0 (%)	i_s (%)	$i - 1_p$ (%)	t (s)	0 (%)	i_s (%)	$i - 1_p$ (%)	t (s)	0 (%)	i_s (%)	$i - 1_p$ (%)
0	naive	27.2	/	/	/	91.3	/	/	/	39.6	/	/	/
1	$\text{for}_p(4)$	16.1	41%	14%	/	79.9	12%	-20%	/	34.7	12%	16%	/
	for_s	18.8	31%	/	/	66.4	27%	/	/	41.5	-5%	/	/
2	$\text{task}_p(4)$	19.3	29%	33%	-19%	72.0	21%	8%	10%	38.3	3%	18%	-10%
	task_s	29.0	-7%	/	/	78.0	15%	/	/	46.7	-18%	/	/

Achievements

- The sorting task is done in parallel with other tasks
- There's no need for `atomic` primitives

Problem

Most of the time is spent on synchronizing threads and scheduling tasks

3. Using sections

Third idea

Define an optimal task schedule and use the `parallel sections` directive instead of `parallel task`.

- We already know the shape of the optimal schedule:

time instant	**1**	**2**	**3**	**4**			**m**	**m+1**	**m+2**	**m+3**
create `sign`	A_1	A_2	A_3	A_4		A_{m-1}	A_m	A_{m+1}		
count and p-sum		B_1	B_2	B_3	B_4		b_{m-1}	B_m	B_{m+1}	
sort			C_1	C_2	C_3	C_4		C_{m-1}	C_m	C_{m+1}

- With 3 threads we use just $m+3$ time units compared to $3(m+1)$ time units used by a serial execution

3. Using sections

We replace the previous code with:

```
1  #pragma omp single     // { A_1 task }
2  #pragma omp sections
3    #pragma omp section // { A_2 task }
4    #pragma omp section // { B_1 task }
5
6  for(j ← 3 upto m+1)
7    #pragma omp sections
8      #pragma omp section  // { A_j task   }
9      #pragma omp section  // { B_j-1 task }
10     #pragma omp section  // { C_j-2 task }
11
12 #pragma omp sections
13   #pragma omp section // { B_m+1 task }
14   #pragma omp section // { C_m task   }
15 #pragma omp single     // { C_m+1 task }
```

3. Using sections

program		ist1B				ist2B				ist3B			
		t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)
0	naive	27.2	/	/	/	91.3	/	/	/	39.6	/	/	/
1	$for_p(4)$	16.1	41%	14%	/	79.9	12%	-20%	/	34.7	12%	16%	/
	for_s	18.8	31%	/	/	66.4	27%	/	/	41.5	-5%	/	/
2	$task_p(4)$	19.3	29%	33%	-19%	72.0	21%	8%	10%	38.3	3%	18%	-10%
	$task_s$	29.0	-7%	/	/	78.0	15%	/	/	46.7	-18%	/	/
3	$section_p(3)$	6.8	**75%**	47%	64%	47.7	**48%**	28%	34%	10.7	**73%**	53%	72%
	$section_s$	12.9	52%	/	/	65.8	28%	/	/	22.9	42%	/	/

Problem

The sorting task takes twice the time of the other tasks.

4. Refining sections

Fourth idea

Split the j-th sorting task.

- We want to create two `sections` that can be parallel executed, aimed at sorting two non-overlapping regions of the array
- The schedule becomes:

time instant	**1**	**2**	**3**	**4**				**m**	**m+1**	**m+2**	**m+3**
create `sign`	A_1	A_2	A_3	A_4			A_{m-1}	A_m	A_{m+1}		
count and p-sum		B_1	B_2	B_3	B_4			b_{m-1}	B_m	B_{m+1}	
sort$_1$			C_1	C_2	C_3	C_4			C_{m-1}	C_m	C_{m+1}
sort$_2$			D_1	D_2	D_3	D_4			D_{m-1}	D_m	D_{m+1}

4. Refining sections

There are two sections performing the j-th sorting task. The first only considers numbers below a certain threshold, and the second one does the opposite.

```
1  #pragma omp section
2    for(i ← n downto 1)
3      index ← sign[in[i]][j]
4      if(index <= th[j]) continue
5      outIndex ← --count[j][index]
6      out[outIndex] ← in[i]
7  #pragma omp section
8    for(i ← n downto 1)
9      index ← sign[in[i]][j]
10     if(index > th[j]) continue
11     outIndex ← --count[j][index]
12     out[outIndex] ← in[i]
```

4. Refining sections

- After computing the prefix-sum over digit j, the sequence `count[j][1]...count[j][k]` is non-decreasing
- `count[j][i] = v` means there are v values smaller or equal than i and $n - v$ values grater than i
- For each digit j we want to find an index `th[j]` such that `count[j][th[j]]` and `n-count[j][th[j]]` are balanced
- For sake of simplicity, we can choose `th[j]` to be the smallest index such that `count[j][th[j]]` is greater than $n/2$

```
1  th[j] ← rand(1, k-1)
2  for(i ← 2 upto k-1)
3    if(count[j][i] >= (n/2))
4      th[j] ← i
5      break
```

4. Refining sections

program		ist1B				ist2B				ist3B			
		t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)
0	naive	27.2	/	/	/	91.3	/	/	/	39.6	/	/	/
1	$\text{for}_p(4)$	16.1	41%	14%	/	79.9	12%	-20%	/	34.7	12%	16%	/
	for_s	18.8	31%	/	/	66.4	27%	/	/	41.5	-5%	/	/
2	$\text{task}_p(4)$	19.3	29%	33%	-19%	72.0	21%	8%	10%	38.3	3%	18%	-10%
	task_s	29.0	-7%	/	/	78.0	15%	/	/	46.7	-18%	/	/
3	$\text{section}_p(3)$	6.8	**75%**	47%	64%	47.7	**48%**	28%	34%	10.7	**73%**	53%	72%
	section_s	12.9	52%	/	/	65.8	28%	/	/	22.9	42%	/	/
4	$\text{splitsort}_p(4)$	7.6	72%	49%	-11%	51.0	44%	29%	-7%	10.7	73%	48%	1%
	splitsort_s	14.8	45%	/	/	71.7	21%	/	/	20.8	47%	/	/

Problems

- The splitting isn't always optimal (see ist2B)
- Doesn't scale with the number of threads

5. Loop Unfolding

Fifth Idea

In the special case $m = 2$, consider two refining iterations in a single step

- I give an optimized way to compute $\sim_{i+2}$ starting from $\sim_i$
- From property 3 (slide 10), we have

$$\begin{aligned} p \sim_{i+2} q &\iff p \sim_{i+1} q \wedge (\forall a \in \Sigma\ \delta(p,a) \sim_{i+1} \delta(q,a)) \\ &\iff p \sim_i q \wedge (\forall a \in \Sigma\ \delta(p,a) \sim_i \delta(q,a)) \\ &\wedge (\forall a \in \Sigma\ \delta(p,a) \sim_i \delta(q,a) \wedge (\forall b \in \Sigma\ \delta(p,ab) \sim_i \delta(q,ab))) \\ &\iff p \sim_i q \wedge (\forall a \in \Sigma\ \delta(p,a) \sim_i \delta(q,a)) \\ &\wedge (\forall a,b \in \Sigma\ \delta(p,ab) \sim_i \delta(q,ab)) \end{aligned}$$

- In this way signatures would require $1 + m + m^2$ digits. It may seem that the m terms are redundant

5. Loop Unfolding

Lemma (Unfolding Property)

For any $i > 0$, we have

$$p \sim_{i+2} q \iff p \sim_i q \wedge (\forall a, b \in \Sigma\ \delta(p, ab) \sim_i \delta(q, ab))$$

- Intuitively, the first term guarantees properties for words of length $0, \ldots, i$ and the universal term guarantees properties for words of length $2, \ldots, i+2$, thus we require $i > 0$
- $\sim_2$ cannot be directly calculated with the above formula
- Signatures can use just $1 + m^2$ digits
- See Appendix B for a generalization

5. Loop Unfolding

- If $m = 2$ and we want to calculate $\sim_2$ and $\sim_3$, the schedule requires 10 steps

			$\sim_2$					$\sim_3$		
time instant	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
create `sign`	A_1	A_2	A_3			A_1	A_2	A_3		
count and p-sum		B_1	B_2	B_3			B_1	B_2	B_3	
sort			C_1	C_2	C_3			C_1	C_2	C_3

- Instead, using the previous lemma and a signature of $m^2 + 1 = 5$ digits, we can use just 7 steps to calculate $\sim_3$ directly from $\sim_1$

time instant	**1**	**2**	**3**	**4**	**5**	**6**	**7**
create `sign`	A_1	A_2	A_3	A_4	A_5		
count and p-sum		B_1	B_2	B_3	B_4	B_5	
sort			C_1	C_2	C_3	C_4	C_5

- In terms of steps, this is an improvement even on the serial execution (18 steps vs 15)
- Doesn't work with $m > 2$

5. Loop Unfolding

	program	ist1S				ist2S				ist3S			
		t (s)	0 (%)	i_s (%)	$i - 1_p$ (%)	t (s)	0 (%)	i_s (%)	$i - 1_p$ (%)	t (s)	0 (%)	i_s (%)	$i - 1_p$ (%)
0	naive	19.7	/	/	/	158.74	/	/	/	109.9	/	/	/
1	$for_p(4)$	15.9	19%	-1%	/	138.36	13%	-36%	/	72.6	34%	25%	/
	for_s	15.8	20%	/	/	101.38	36%	/	/	96.7	12%	/	/
2	$task_p(4)$	14.9	24%	5%	7%	125.96	21%	-18%	9%	72.4	34%	19%	0%
	$task_s$	15.7	20%	/	/	106.92	33%	/	/	89.7	18%	/	/
3	$section_p(3)$	9.0	**54%**	31%	39%	88.24	44%	9%	30%	47.2	57%	28%	35%
	$section_s$	13.2	33%	/	/	97.36	39%	/	/	65.6	40%	/	/
4	$splitsort_p(4)$	10.2	48%	27%	-13%	86.13	46%	19%	2%	40.8	**63%**	14%	14%
	$splitsort_s$	14.0	29%	/	/	106.87	33%	/	/	65.5	40%	/	/
5	$unfold_p(4)$	11.7	40%	30%	-15%	63.92	**60%**	23%	26%	48.5	56%	37%	-19%
	$unfold_s$	16.9	14%	/	/	82.56	48%	/	/	76.8	30%	/	/

Achievement

Performance improvements in the case the number of iterations is high.

Problem

There is an additional overhead on instances with constant number of iterations (see instances ist1S and ist3S).

6. Parallel RadixSort

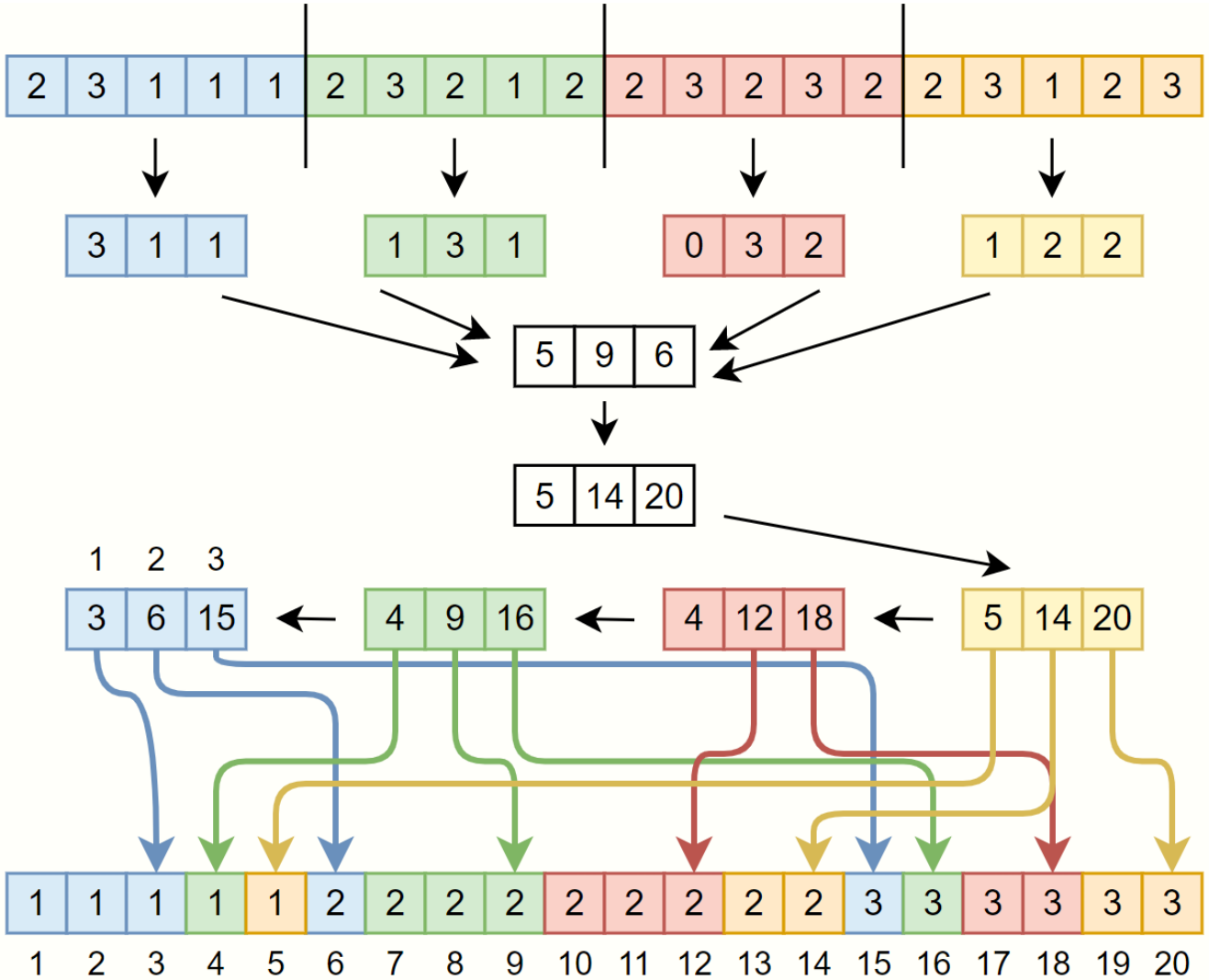
Sixth Idea

Try to parallelize `radixSort`.

- I parallelize `radixSort` by parallelizing `countingSort`
- The literature is full of results, here I try the very simplest approach
- Suppose to have N processors, the steps are:
  1. Each processor i, fills $\texttt{count}_\texttt{i}$
  2. Sum local counters and obtain `globalCount`
  3. Compute the prefix-sum over `globalCount`
  4. Redistribute information to local counters
  5. Each processor i, uses $\texttt{count}_\texttt{i}$ to sort the array

6. Parallel RadixSort

The following figure illustrates explains `parallelCountingSort`:



6. Parallel RadixSort

- Each step can be implemented with a `parallel for` directive
- Let us focus on steps 1 and 5

```
1  chunkSize ← n/N
2  i ← omp_get_thread_num()
3  // step 1
4  #pragma omp for schedule(static, chunkSize)
5    for (j ← 1 upto n)
6      count_i[arr[j]]++
7  ...
8  // step 5
9  #pragma omp for schedule(static, chunkSize)
10   for (j ← 1 upto n)
11     ind ← ...
12     el ← arr[ind]
13     outIndex ← --count_i[ind]
14     out[outIndex]← el
```

6. Parallel RadixSort

Recall: on the `parallel for` directive (OpenMP 5.0)

Suppose that N threads encounter a `parallel for` region, with option `schedule(static, chunkSize)`. If the `for` consists of n iterations:

1. The iterations are divided into contiguous non-empty subsets, called chunks
2. Each chunk has dimension `chunckSize`
3. Each thread executes its assigned chunk(s)
4. Different worksharing-loop regions with the same schedule and iteration count, even if they occur in the same parallel region, can distribute iterations among threads **differently**

6. Parallel RadixSort

- Rule 4 of previous slide is a problem
- We want to ensure that each thread i receives the same chunk C_i in step 5 as it did in step 1
- Fortunately, with `schedule(static, chunkSize)` the same assignment of logical iteration numbers to threads will be used in two worksharing-loop regions if:
  1. Both worksharing-loop regions have the same number of iterations
  2. Both regions have the same value of `chunkSize` specified
  3. Both regions bind to the same parallel region
  4. Neither loop is associated with a SIMD construct
- The correctness of our implementation is guaranteed

6. Parallel RadixSort

Achievement

The approach scales with the number of processors (N).

Problem

There is an additional overhead due to the new procedure, so with 4 threads it is not the most efficient approach.

Results and Discussion

program		ist1B				ist2B				ist3B			
		t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)
0	naive	27.2	/	/	/	91.3	/	/	/	39.6	/	/	/
1	$\text{for}_p(4)$	16.1	41%	14%	/	79.9	12%	-20%	/	34.7	12%	16%	/
	for_s	18.8	31%	/	/	66.4	27%	/	/	41.5	-5%	/	/
2	$\text{task}_p(4)$	19.3	29%	33%	-19%	72.0	21%	8%	10%	38.3	3%	18%	-10%
	task_s	29.0	-7%	/	/	78.0	15%	/	/	46.7	-18%	/	/
3	$\text{section}_p(3)$	6.8	**75%**	47%	64%	47.7	**48%**	28%	34%	10.7	**73%**	53%	72%
	section_s	12.9	52%	/	/	65.8	28%	/	/	22.9	42%	/	/
4	$\text{splitsort}_p(4)$	7.6	72%	49%	-11%	51.0	44%	29%	-7%	10.7	73%	48%	1%
	splitsort_s	14.8	45%	/	/	71.7	21%	/	/	20.8	47%	/	/
5	$\text{unfold}_p(4)$	8.3	69%	45%	-9%	50.6	45%	30%	1%	10.4	74%	47%	3%
	unfold_s	15.2	44%	/	/	71.8	21%	/	/	19.8	50%	/	/
6	$\text{psort}_p(4)$	15.3	44%	44%	-85%	88.3	3%	2%	-75%	31.1	22%	55%	-198%
	psort_s	27.3	0%	/	/	90.5	1%	/	/	69.5	-75%	/	/

Table 1: Time analysis on *Big* instances.

Results and Discussion

program		ist1S				ist2S				ist3S			
		t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)	t (s)	0 (%)	i_s (%)	$i-1_p$ (%)
0	naive	19.7	/	/	/	158.74	/	/	/	109.9	/	/	/
1	$\text{for}_p(4)$	15.9	19%	-1%	/	138.36	13%	-36%	/	72.6	34%	25%	/
	for_s	15.8	20%	/	/	101.38	36%	/	/	96.7	12%	/	/
2	$\text{task}_p(4)$	14.9	24%	5%	7%	125.96	21%	-18%	9%	72.4	34%	19%	0%
	task_s	15.7	20%	/	/	106.92	33%	/	/	89.7	18%	/	/
3	$\text{section}_p(3)$	9.0	**54%**	31%	39%	88.24	44%	9%	30%	47.2	57%	28%	35%
	section_s	13.2	33%	/	/	97.36	39%	/	/	65.6	40%	/	/
4	$\text{splitsort}_p(4)$	10.2	48%	27%	-13%	86.13	46%	19%	2%	40.8	**63%**	14%	14%
	splitsort_s	14.0	29%	/	/	106.87	33%	/	/	65.5	40%	/	/
5	$\text{unfold}_p(4)$	11.7	40%	30%	-15%	63.92	**60%**	23%	26%	48.5	56%	37%	-19%
	unfold_s	16.9	14%	/	/	82.56	48%	/	/	76.8	30%	/	/
6	$\text{psort}_p(4)$	11.0	44%	38%	6%	104.43	34%	21%	-63%	63.4	42%	44%	-31%
	psort_s	17.9	9%	/	/	132.17	17%	/	/	112.5	-2%	/	/

Table 2: Time analysis on *Small* instances.

Results and Discussion

1. Poor performance due to the serial sorting part
2. Poor performance due to synchronization
3. Good performance; the pipeline only uses 3 threads
4. The pipeline now uses 4 threads but the splitting isn't always optimal and may cause an additional overhead
5. Good performance in theory but not much in practice; It causes an overhead on instances requiring a small number of iterations, hence poor performance in the best and average case but good performance in the worst case
6. The parallel `radixSort` procedure causes an additional overhead but it scales with the number of processors

Results and Discussion

Which program is better?

With more than 16 processors use program **6**. With 3 processors use program **3**. With 4-16 processors use program **4**. In the special case $m = 2$, if the number of iteration is suspected not to be constant w.r.t n, use program **5**.

- I didn't come up with a uniform strategy, instead I used different heuristics addressing different cases
- I was unable to parallelize the computation of new classes (lines 8-12 of the serial `refine` function slide 12)
- The sorting algorithm used was `radixSort` (as in the serial code) but other algorithms may be more suitable for an OpenMP implementation

CUDA

CUDA

This part is presented on the blackboard.

Conclusions

Final Results

		ist1B	ist2B	ist3B	ist1S	ist2S	ist3S
serial program	time (s)	27.2	91.3	39.6	19.7	158.74	109.9
best OpenMP result	program	3	3	3	3	5	4
	time (s)	6.8	47.7	10.7	9.0	63.9	40.8
	saving (%)	75 %	48 %	73 %	54 %	60 %	63 %
best CUDA result	program	7Vb	7Va	7Vc	7Ob	7Oa	7Oc
	time (s)	2.6	241.10	18.69	/	341.31	36.0
	saving (%)	90 %	- 164 %	52 %	/	- 115 %	67 %

Table 3: Best programs and architectural comparison.

	ist1B	ist2B	ist3B	ist1S	ist2S	ist3S
program	7Vb	7Va	7Vc	7Ob	7Oa	7Oc
time (s)	0.3	41.51	1.87	0.51	28.88	3.63
saving (%)	99 %	55 %	95 %	97 %	81 %	97 %

Table 4: CUDA performance on a more powerful device.

Future works

Regarding the OpenMP implementation:

- Test the scalability of program 6
- Test different sorting algorithms
- Test OpenMP programs that will be executed on the device

Regarding the CUDA implementation:

- Use the "top-down" decomposition to split each instance of CountingSort into two instances and execute them on two devices
- Compare the presented method to standard libraries

Regarding theoretical results:

- Investigate the computation of $\sim_i$ using less *digits* as possible

Bibliography I

[BDN07] F. Bassino, J. David, and C. Nicaud. "REGAL: A Library to Randomly and Exhaustively Generate Automata". In: July 2007. ISBN: 978-3-540-76335-2. DOI: 10.1007/978-3-540-76336-9_28.

[BDN09] F. Bassino, J. David, and C. Nicaud. "On the average complexity of Moore's state minimization algorithm". In: *arXiv preprint arXiv:0902.1048* (2009).

[BDN12] F. Bassino, J. David, and C. Nicaud. "Average case analysis of Moore's state minimization algorithm". In: *Algorithmica* 63.1-2 (2012), pp. 509–531.

Bibliography II

[Hop71] J. Hopcroft. "An n log n algorithm for minimizing states in a finite automaton". In: *Theory of machines and computations*. Elsevier, 1971, pp. 189–196.

[Nic14] C. Nicaud. "Random Deterministic Automata". In: *Mathematical Foundations of Computer Science 2014*. Ed. by E. Csuhaj-Varjú, M. Dietzfelbinger, and Z. Ésik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 5–23. ISBN: 978-3-662-44522-8.

[Wat93] B. W. Watson. "A taxonomy of finite automata minimization algorithms". In: (1993).

Appendix

A. Moore's Algorithm: Average Case Analysis

- A **Transition Structure** T is an automaton where the set of final states F has not been fixed
- Let $\mathcal{T}_n$ and $\mathcal{U}_n$ be the set of transition structures with n states and the set of automata with n states
- $|\mathcal{T}_n| = n \cdot n^{nm}$
- $|\mathcal{U}_n| = |\mathcal{T}_n| \cdot 2^n = n \cdot n^{nm} \cdot 2^n$
- It can be shown that with high probability, a randomly chosen element of $\mathcal{T}_n$ is not accessible

A. Moore's Algorithm: Average Case Analysis

- Given a transition structure $T \in \mathcal{T}_n$, $\mathcal{F}_l(p, q, p', q')$ is the set of sets of final states F such that in automaton (T, F), p leads to p' through a word w of length l, q leads to q' through w, $p \sim_{l-1} q$, $p' \not\sim_0 q'$ (hence $p \not\sim_l q$)
- It can be shown that $|\mathcal{F}_l(p, q, p', q')| \leq 2^{n-l}$
- Let $\mathcal{F}^{\geq l} = \bigcup_{p \neq q, p' \neq q'} \mathcal{F}_l(p, q, p', q')$
- A set $F \in \mathcal{F}^{\geq l}$ witnesses that Moore's algorithm takes more than l iterations on automaton (T, F)
- $|\mathcal{F}^{\geq l}| \leq \sum |\mathcal{F}_l(p, q, p', q')| \leq n^4 \cdot 2^{n-l}$
- Let $\mathit{Moore}(T, F)$ be the number of iterations taken by Moore's algorithm on automaton (T, F)

A. Moore's Algorithm: Average Case Analysis

Theorem

Given $T \in \mathcal{T}_n$, the average number of iterations taken by Moore's algorithm on (T, F) where F is chosen at random is $\log n$, thus the expected running time is $mn \log n$

For fixed l, if the set F of final states is chosen at random, the average number of iterations taken by Moore's algorithm is

$$\begin{aligned}
&\frac{1}{2^n} \sum_{F \subset \{1,\dots,n\}} Moore(T, F) \\
&= \frac{1}{2^n} \sum_{F \in \mathcal{F}^{<l}} Moore(T, F) + \frac{1}{2^n} \sum_{F \in \mathcal{F}^{\geq l}} Moore(T, F) \\
&\leq \frac{2^n \cdot l}{2^n} + \frac{n^4 \cdot 2^{n-l} \cdot n}{2^n} = l + n^5 \cdot 2^{-l}
\end{aligned}$$

Choosing $l = \lceil 5 \log n \rceil$ we obtain the desired result.

B. Loop Unfolding

The presented lemma admits a simple generalization:

Lemma (Generalized Unfolding Property)

For any $k > 2$, for any $i \geq k - 1$

$$p \sim_{i+k} q \iff p \sim_i q \wedge (\forall a_1, \ldots, a_k \in \Sigma \; \delta(p, a_1 \ldots a_k) \sim_i \delta(q, a_1 \ldots a_k))$$