Histogram Multiprocessing (MPI) - Paper

Problem Description

The task was to analyse the given text documents and to display the count of all characters and all words in the given text afterwards. Counting the characters is straightforward and only requires counting of each character appearing in the text. The task to count the words is more complicated, as we must first define what a "word" is. The man page[1] of the Linux command "wc" considers a word to be a string of characters with a length greater than zero delimited by white space. White space is defined by the function isspace()[2] of the ISO C standard. However, we created our own definition of a word. We decided to count both characters and words case-sensitive, which means that e.g., 'a' and 'A' are counted separately, and 'House' and 'house' are counted separately.

Definition of a word

Based on the given example texts, we tried to come up with a fitting definition of a word. We decided to include the characters between 'A' and 'Z' and between 'a' and 'z' to count as part of a word. We also included all characters, which require more than 1 byte in UTF-8 (which have a decimal value larger than 127 in UTF-8). We did this to include characters with diacritics such as 'á' or 'é', which appear for example in the given text "king arthur.txt". When a not included character is encountered, we end the word and start a new one. The table in Figure 1 shows the included ASCII characters.

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL	32	space	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n

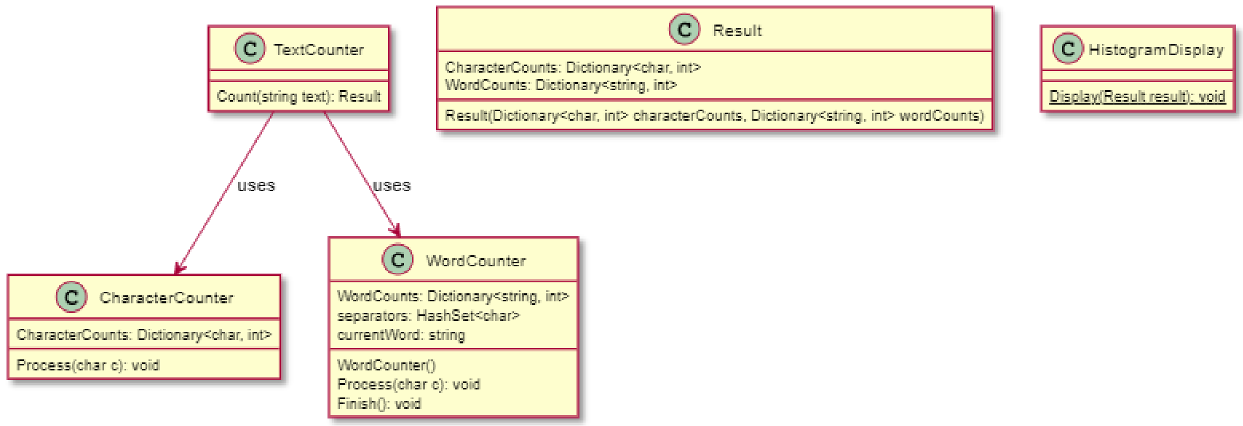
[1] https://man7.org/linux/man-pages/man1/wc.1p.html
[2] https://pubs.opengroup.org/onlinepubs/009604499/functions/isspace.html

15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Figure 1: ASCII table: The marked characters and all characters not in this table were defined as valid parts of a word.

Sequential Solution

We started with a sequential solution using only one process (and thread) as a proof of concept. The program takes the file to analyse as a single command line argument. At the start, the whole file is read into the memory as a string. Afterwards the program loops over all characters and counts the characters and words as described in the section Problem Description. At the end, the character counts and word counts are displayed in the console ordered by their character/string value. Figure 1, Figure 2 and Figure 3 show the design of the sequential solution in UML diagrams.



TextCounter
Count(string text): Result
Result
CharacterCounts: Dictionary<char, int>
WordCounts: Dictionary<string, int>
Result(Dictionary<char, int> characterCounts, Dictionary<string, int> wordCounts)
HistogramDisplay
Display(Result result): void
uses
uses
CharacterCounter
CharacterCounts: Dictionary<char, int>
Process(char c): void
WordCounter
WordCounts: Dictionary<string, int>
separators: HashSet<char>
currentWord: string
WordCounter()
Process(char c): void
Finish(): void

Figure 2: Sequential Solution Class Diagram

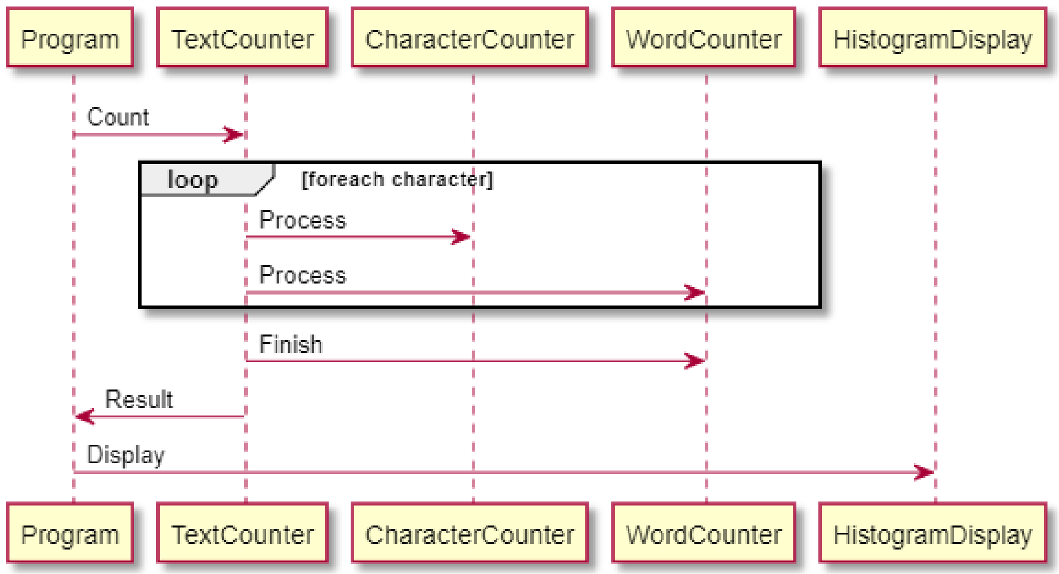


Figure 3: Sequential Solution Sequence Diagram

Parallel Solution (Multiprocessing)

Based on the sequential solution, we tried to come up with a parallel solution. The problem can be solved with a “single program and multiple data” (SPMD) approach, as we can split the text into independent parts and use the same program for every part. The Master (or Controller) reads the whole file from the disk and tries to split it into equal sized parts. It is important that every part has nearly the same size, so that every process has the same work to do. The text is not split in the middle of a word and only done when the position of the split is a whitespace character. Every part is then sent to a Worker via an MPI_Scatter. The Master and the Workers count the occurrences of all characters and words in their respective text part. After the counting is finished, the results are propagated back to the Master via an MPI_Gather. At the end the Master process sums up the character and word counts and displays the results in the console. Figure 4 and Figure 5 show the implementation design for the parallel solution.

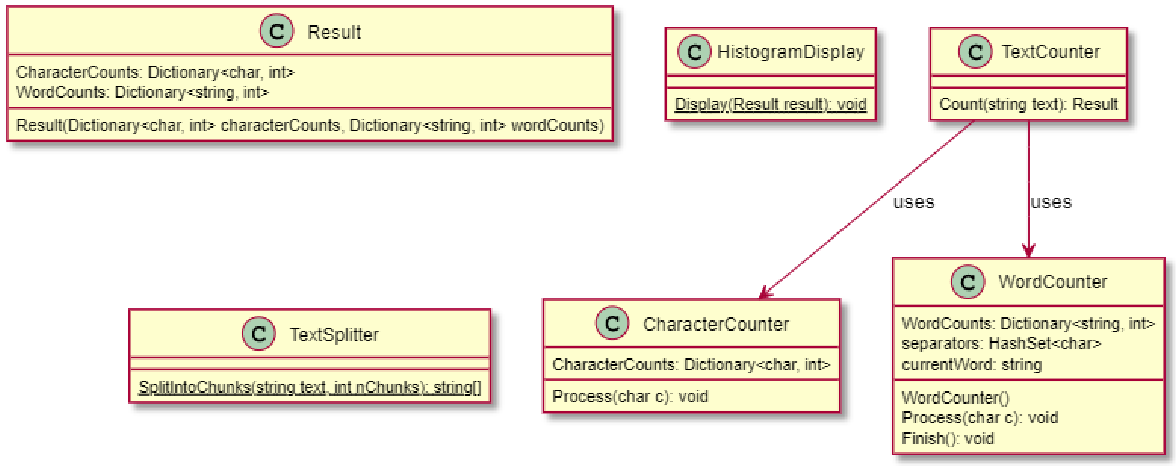

C Result
CharacterCounts: Dictionary<char, int>
WordCounts: Dictionary<string, int>
Result(Dictionary<char, int> characterCounts, Dictionary<string, int> wordCounts)
C HistogramDisplay
Display(Result result): void
C TextCounter
Count(string text): Result
uses
uses
C TextSplitter
SplitIntoChunks(string text, int nChunks): string[]
C CharacterCounter
CharacterCounts: Dictionary<char, int>
Process(char c): void
C WordCounter
WordCounts: Dictionary<string, int>
separators: HashSet<char>
currentWord: string
WordCounter()
Process(char c): void
Finish(): void

Figure 4: MPI Solution Class Diagram

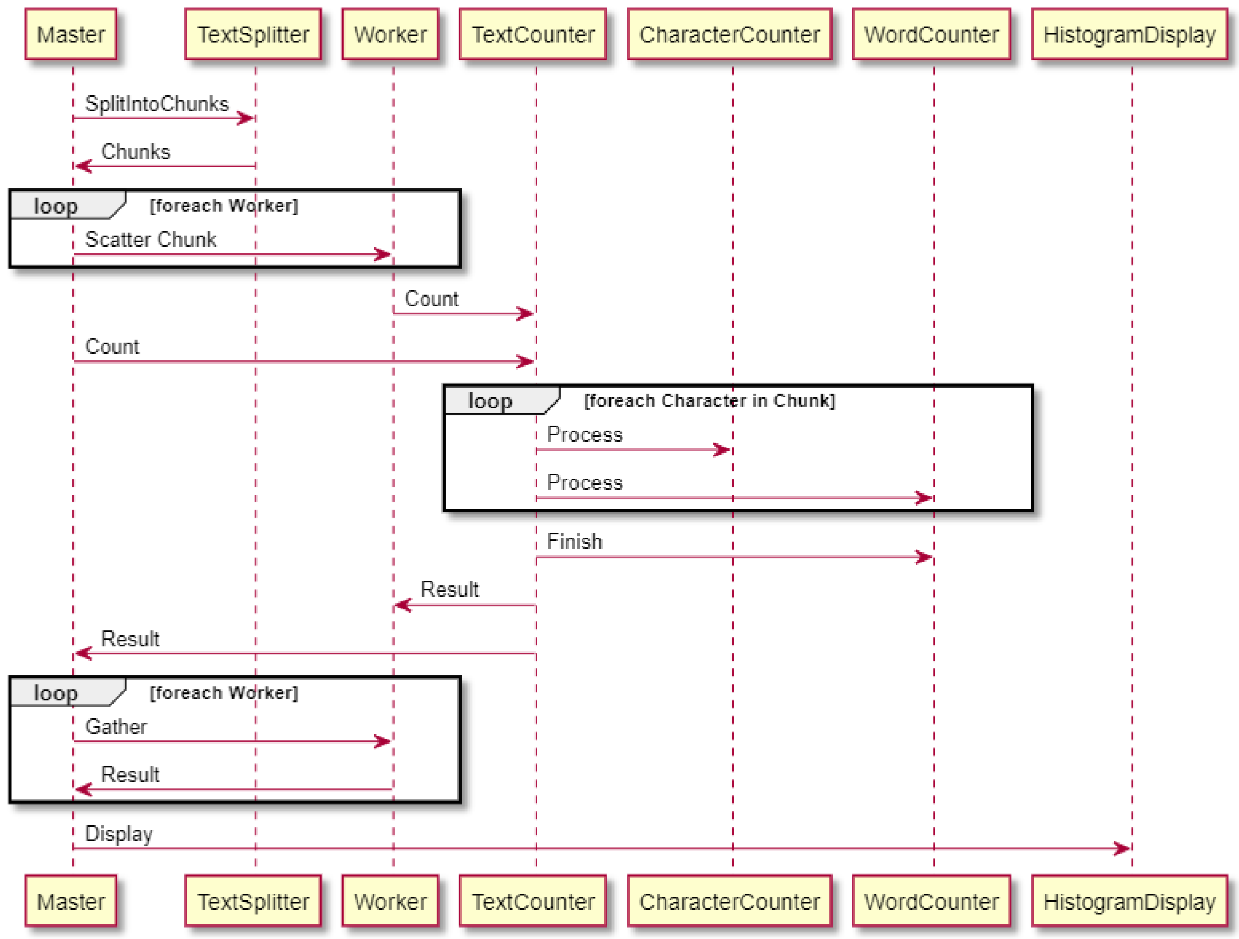


Figure 5: MPI Solution Sequence Diagram

Performance

All Tests were conducted with the following hardware:

- CPU: AMD Ryzen 5 4500U with Radeon Graphics, 2375 Mhz, 6 Core(s), 6 Logical Processor(s)
- RAM: 16.0 GB (2x8 Dual Channel)
- OS: Microsoft Windows 11 Pro
- C# Version: C# 10.0

Every datapoint in the following diagrams is the average of 5 measurements. So for every process count there were 5 measurements. Figure 6 shows the average time it took to analyze the example text “lesmiserables.txt” with the sequential solution and with the parallel solution with different amounts of processes. Figure 7 shows the average time it took for the MPI routines with the different amounts of processes.

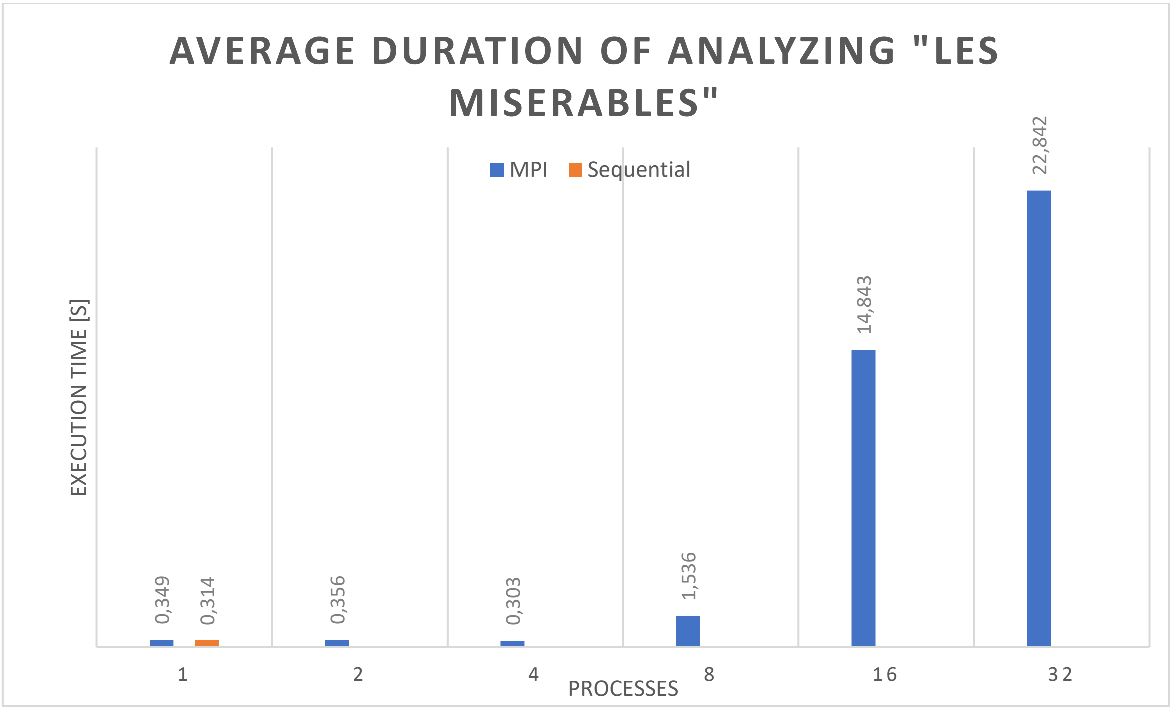


Figure 6: Duration for analysing "lesmiserables.txt"

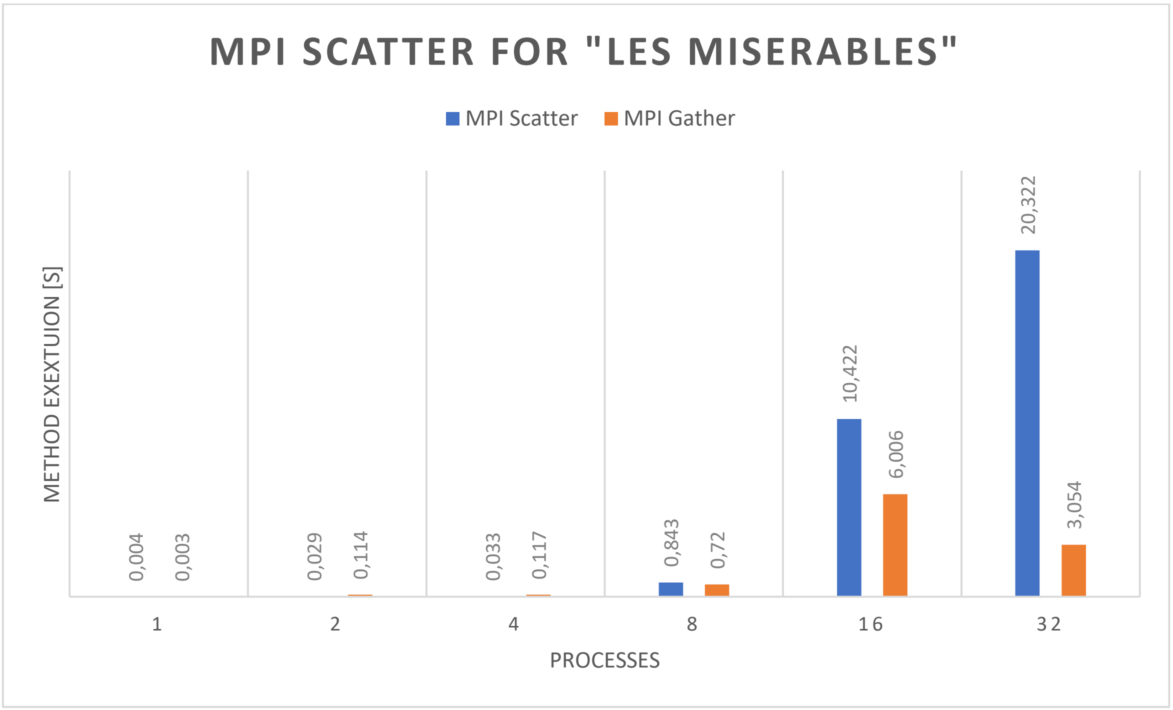


Figure 7: Duration for scatter and gather "lesmiserables.txt"

We also conducted tests with the example text “lesmiserables.txt” replicated 10 times, to see if the parallel solution can perform better with more work to do. Figure 8 shows the average time it took to analyze the example text “lesmiserables.txt” replicated 10 times with the sequential solution and the parallel solution with different amounts of processes. Figure 9 shows the time it took for the MPI routines for the different amounts of processes.

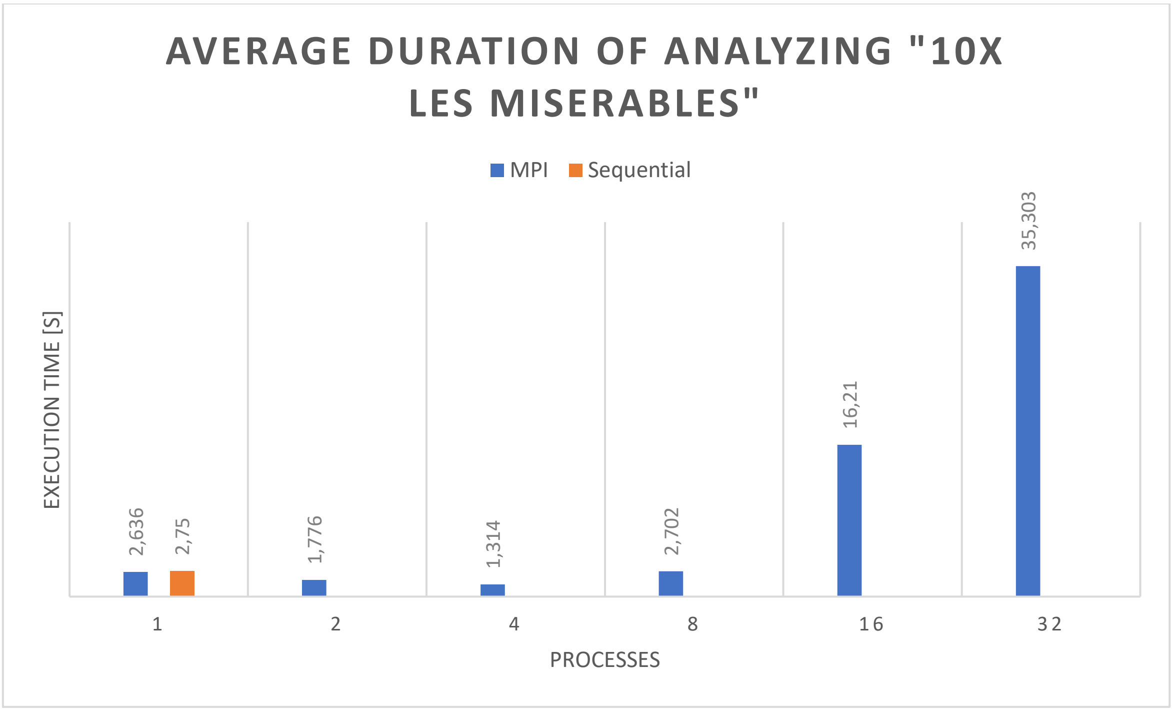


Figure 8: Duration for analysing “lesmiserables.txt” replicated 10 times

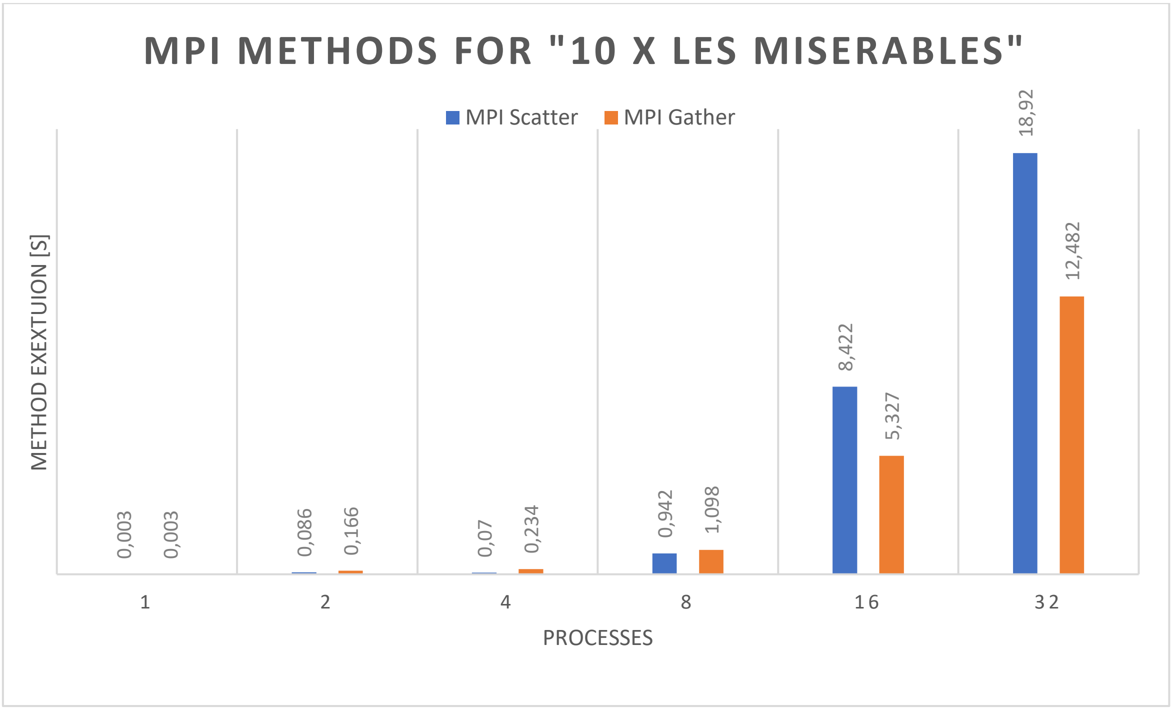


Figure 9: Duration for scatter and gather “lesmiserables.txt” replicated 10 times

Conclusion

The performance tests with the example “lesmiserables.txt” showed that the parallel solution does not provide a real performance boost for such small text sizes. With an increasing amount of processes, the overhead for the communication increases and slows down the whole processing. Starting with 8 processes, the program was mostly occupied with communication between the processes. With less than 8 processes, the parallel solution has an equal performance as the sequential solution.

The tests with the larger text showed that the processing time can benefit from the parallel solution, when there is more work to do. The analysis of the example “lesmiserable.txt” replicated 10 times with 4 processes, was on average 2x faster ($\frac{2.636\ \text{s}}{1.314\ \text{s}} = 200.6\%$), than the sequential solution.

As illustrated by the diagrams, the two MPI methods MPI_Scatter and MPI_Gather account for 70-95% of the runtime for runs with equal to or more than 8 processes.

Given that the communication overhead of MPI appears to contribute significantly to the execution time, a purely multi-threaded, rather than multi-process implementation should be the most advantageous, since threads can access the same memory without the communication overhead of MPI.