Team 2
Huffman encoder

Valon Berisha, Stefan Helfert, Jacob Schlesinger, Maximilian Körner

huffman encoding in general

what it is:

- *entropy encoding algorithm*
- *lossless data compression*
- *variable code length*

how it works:

- analyse data based on word size
- count each word
- build a tree based on frequency of each word
- generate new huffman codes
- replace old words with new huffman codes

example

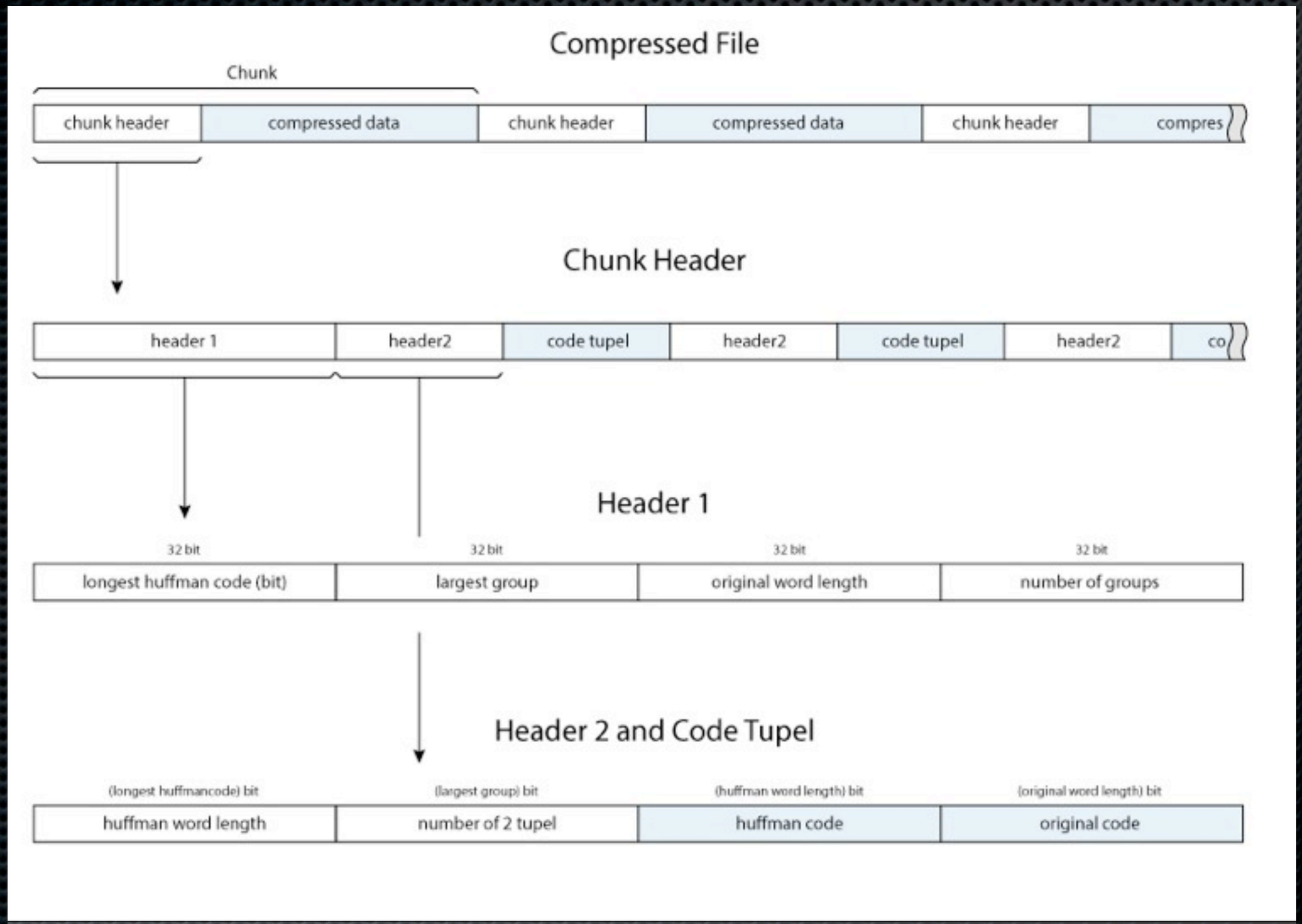
this is a test

	a	e	h	i	s	t	space
freq	1	1	1	2	3	3	3
code	11110	11111	1110	110	1	10	00

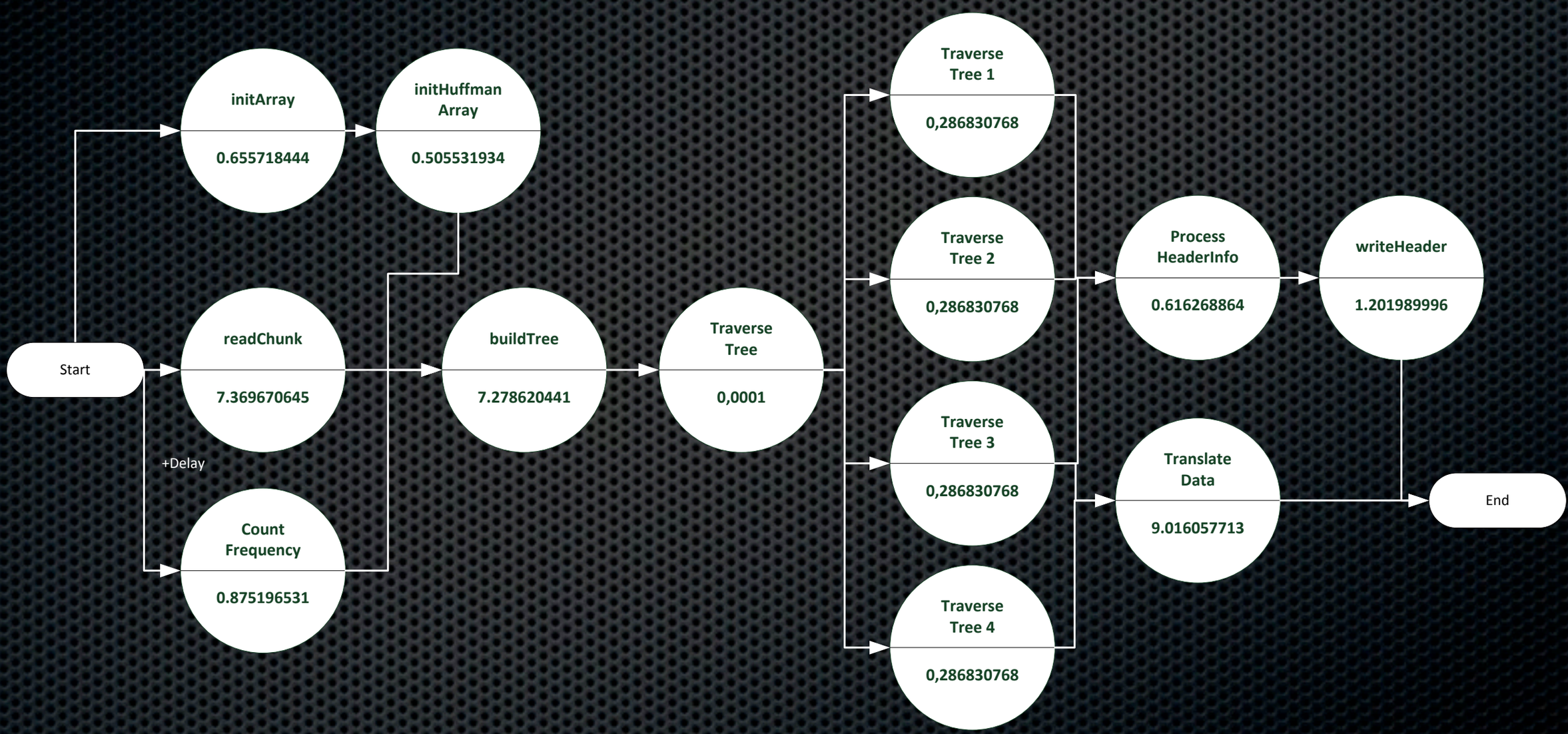
new encoded huffman text

10111101100100110010011111000101111110110

header

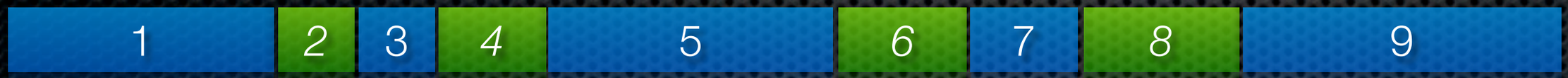

Compressed File
Chunk
chunk header
compressed data
chunk header
compressed data
chunk header
compres
Chunk Header
header 1
header2
code tupel
header2
code tupel
header2
co
Header 1
32 bit
32 bit
32 bit
32 bit
longest huffman code (bit)
largest group
original word length
number of groups
Header 2 and Code Tupel
(longest huffmancode) bit
(largest group) bit
(huffman word length) bit
(original word length) bit
huffman word length
number of 2 tupel
huffman code
original code

taskgraph

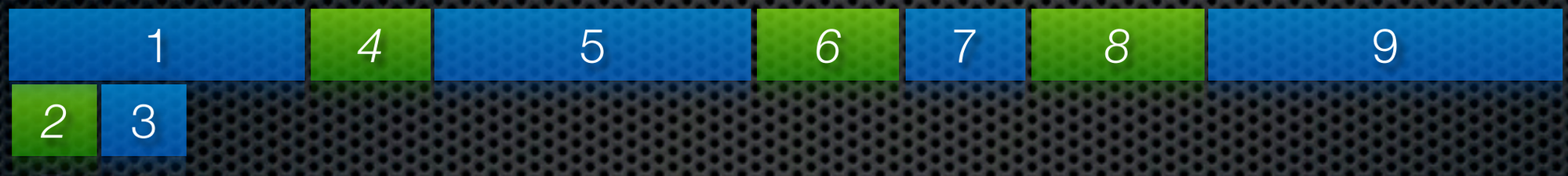

initArray
0.655718444
initHuffman Array
0.505531934
readChunk
7.369670645
buildTree
7.278620441
Traverse Tree
0,0001
Start
+Delay
Count Frequency
0.875196531
Traverse Tree 1
0,286830768
Traverse Tree 2
0,286830768
Traverse Tree 3
0,286830768
Traverse Tree 4
0,286830768
Process HeaderInfo
0.616268864
writeHeader
1.201989996
Translate Data
9.016057713
End

schedules

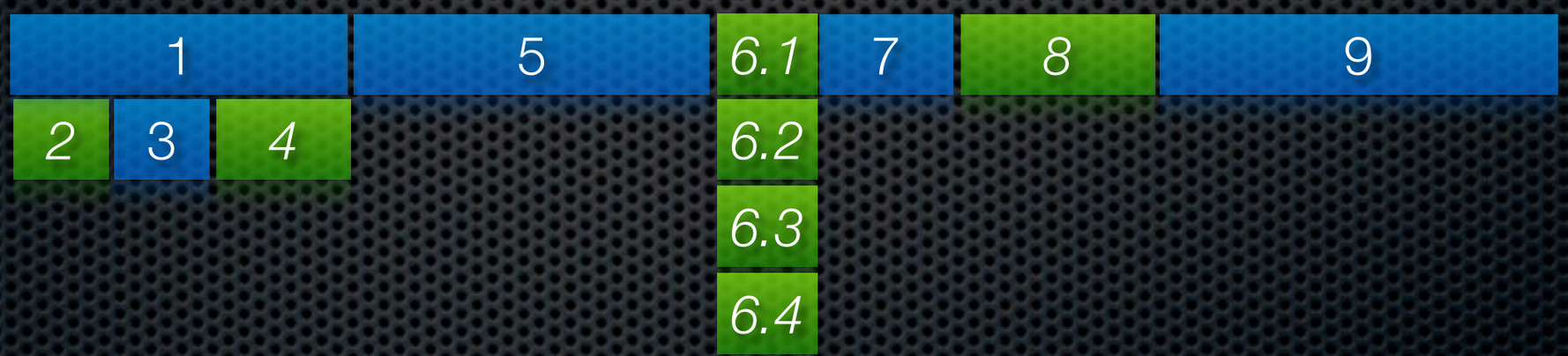
sequential - no deadline



parallel - long deadline



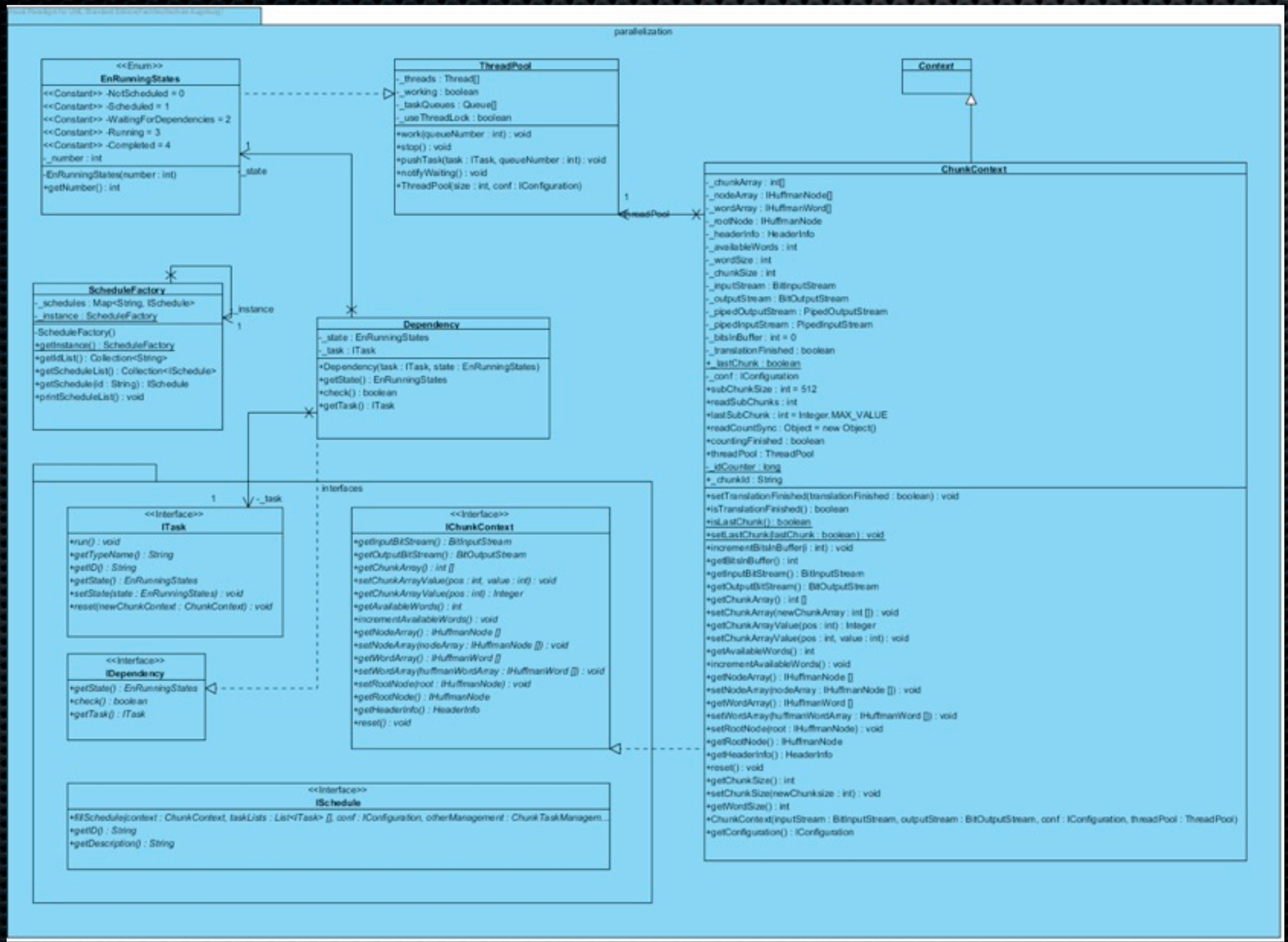
parallel - short deadline



theorethical speedup: 10,3%

1. read chunk
2. init node array
3. init word array
4. count frequencies
5. build tree
6. traverse tree
7. process header info
8. write header
9. translate data

class diagram parallelization



process of measuring

- central logging of each task
- generation of schedule diagrams via visualizer on logging information
- time measurement of tasks or parts of code
- comparison of different schedules
- code validation via profiler

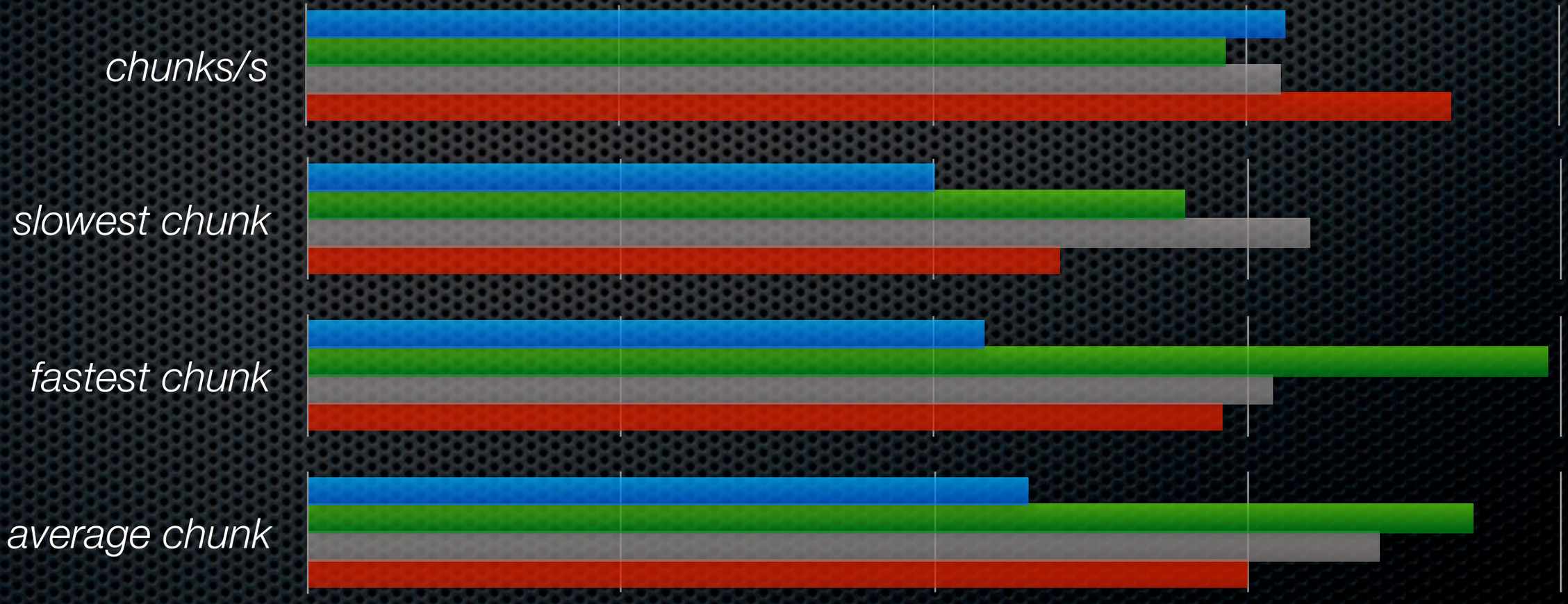
implemented schedules

- sequential - 1 thread, split in tasks (contains overhead)
- sequentialSingleTask - 1 thread, 1 task, no overhead
- basicParallel - implementation of „short deadline“
- mirrored - basicParallel with task interleaving
- improved basic - uses gaps for sequential execution

measured chunk times

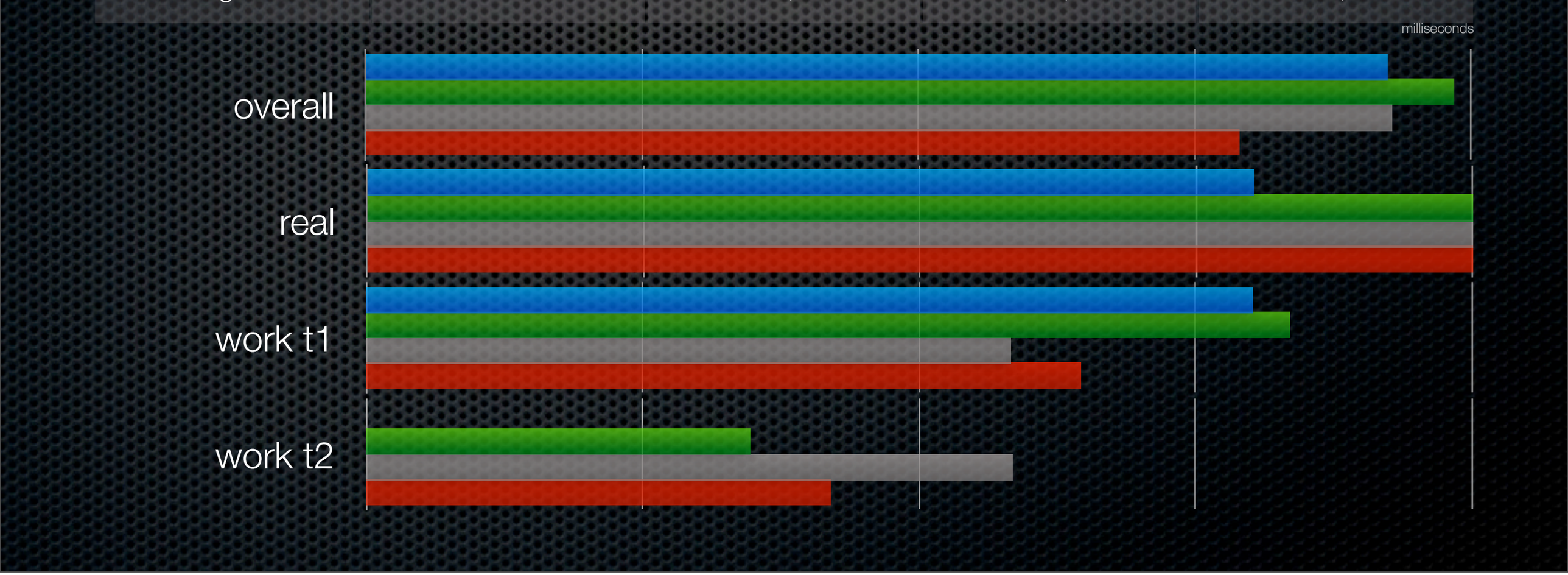
	sequential	basic parallel	mirrored	improved
number of chunks	954	954	954	954
chunks/s	859	807	855	1005
slowest chunk	5,58	7,34	8,8	6,81
fastest chunk	0,54	0,99	0,77	0,73
average chunk	1,15	1,86	1,71	1,50

milliseconds



measured thread times

	sequential	basic parallel	mirrored	improved
overall running time	1109,59	1181,50	1114,69	948,48
real working time	962,46	1420,30	1402,53	1280,59
working time t1	962,46	1003,03	700,39	776,33
working time t2	-	417,27	702,14	504,26

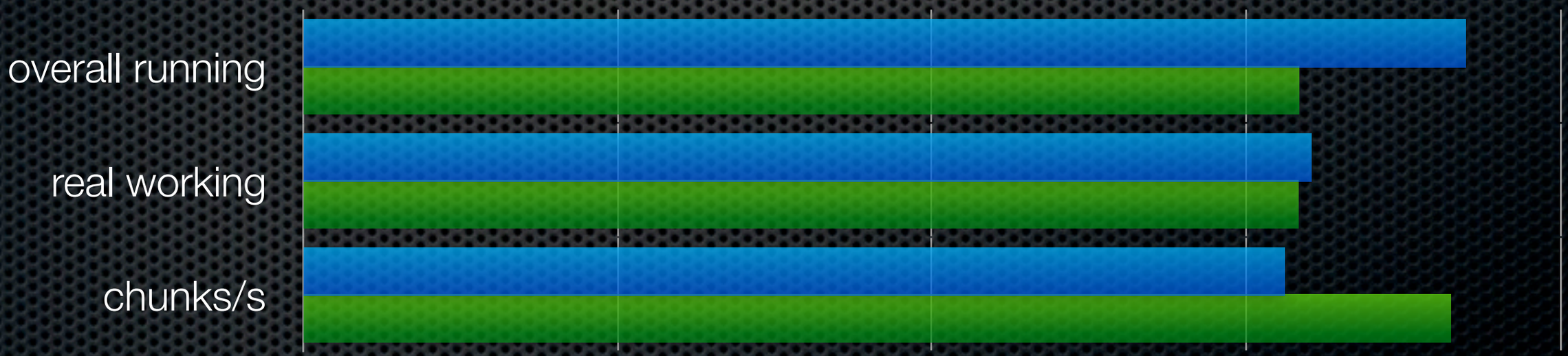


idle percentages

	sequential	basic parallel	mirrored	improved
processor idle (%)	13,26	39,89	37,09	32,49
t1 idle (%)	13,26	15,10	37,17	18,15
t2 idle (%)	-	64,68	37,01	46,84

overhead estimation

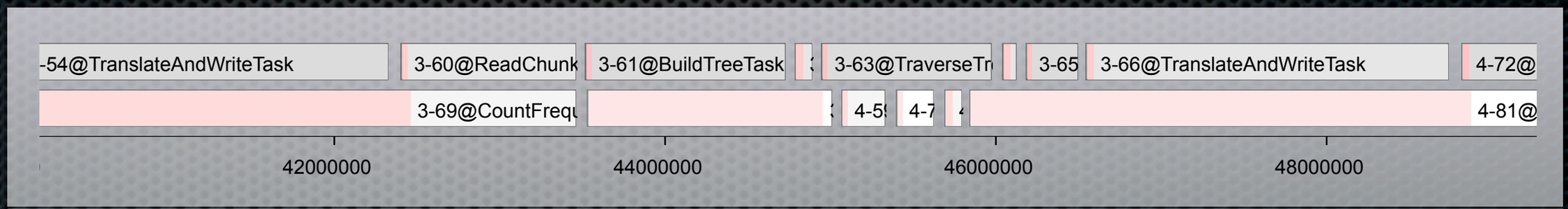
	overall running	real working	chunks/s
sequential (multiple tasks)	1109,59	962,46	859
sequential (single task)	950,36	950,01	1004



overhead approximately: 16%

visualizer

- reads logfile - no performance reduction during runtime
- generates SVG image to display task lifetime
- supports visualisation of several parallel threads



conclusion

- our approach only makes sense for problems/ algorithms with a theoretical speedup of at least >25%
- we could probably reduce the overhead on realtime systems
- usage of affinity locks makes no sense on non realtime systems

thanks for your attention

any questions?

links

- huffman algorithm: http://en.wikipedia.org/wiki/Huffman_coding
- project git: https://github.com/team2Huffman/huffman_encoding