Sprawozdanie z MapReduce

Patryk Wojtyczek

Konfiguracja klastrów

Przeprowadziłem testy w następujących ustawieniach: Oba testowane klasty korzystały z maszyn tego samego typu co pozwoli na łatwiejszą interpretację skalowalności systemu.

instance type	core nodes
m4.xlarge	2
m4.xlarge	4

Oprócz testowania wydajności klastra przetestowałem też sekwencyjny program.

Sekwencyjny program:

```
use std::collections::HashMap;
use std::io;
use std::io::{BufRead, Write};

fn main() {
    let stdin = io::stdin();
    let mut map = HashMap::new();
    let mut lines = stdin.lock().lines();

    while let Some(line) = lines.next() {
        let value = line.unwrap();

        // trim whitespaces
        let trimmed = value.trim();

        // split on spaces to find words
        for word in trimmed.split(" ") {
            *map.entry(word.to_string()).or_insert(0) += 1;
        }
    }

    let stdout = io::stdout();
    let mut handle = stdout.lock();

    for (key, value) in map.into_iter() {
        let _ = handle.write(key.as_bytes());
        let _ = handle.write(b" : ");
        let _ = handle.write(value.to_string().as_bytes());
        let _ = handle.write(b"\n");
    }
}
```

Mapper:

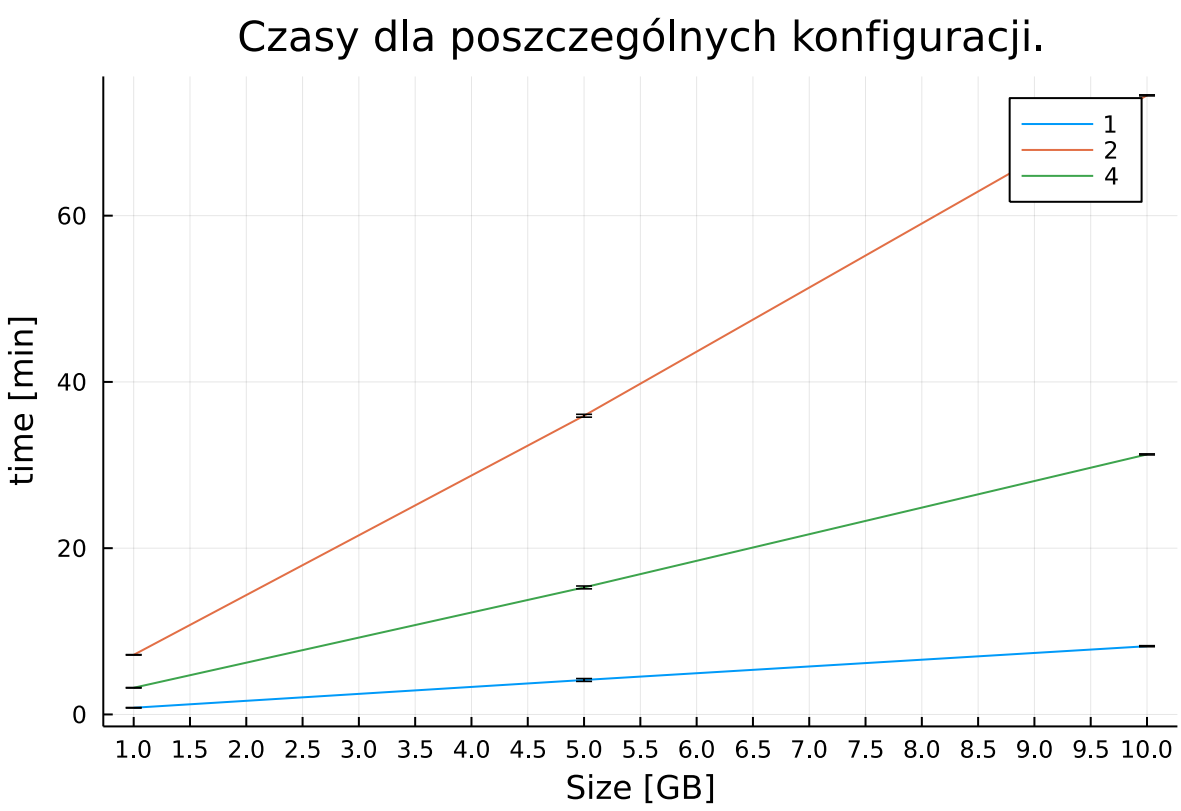
```
extern crate efflux;
extern crate regex;

use efflux::prelude::{Context, Mapper};
use regex::Regex;

fn main() {
    // simply run the mapping phase with our mapper
    efflux::run_mapper(WordcountMapper::new());
}

/// Simple struct to represent a word counter mapper.
///
/// Contains several internal patterns to use when processing
/// the input text, to avoid re-compilation of Regex.
struct WordcountMapper {
    multi_spaces: Regex,
    punc_matcher: Regex,
}

impl WordcountMapper {
    /// Creates a new `WordcountMapper` with pre-compiled `Regex`.
    pub fn new() -> Self {
        Self {
            // detects multiple spaces in a row
            multi_spaces: Regex::new(r"\s{2,}").unwrap(),
            // detects punctuation followed by a space, or trailing
            punc_matcher: Regex::new(r"[[:punct:]](\s|$)").unwrap(),
        }
    }
}

// Mapping stage implementation.
impl Mapper for WordcountMapper {
    /// Mapping implementation for the word counter example.
    ///
    /// The input value is split into words using the internal patterns,
    /// and each word is then written to the context.
    fn map(&mut self, _key: usize, value: &[u8], ctx: &mut Context) {
        // skip empty
        if value.is_empty() {
            return;
        }

        // parse into a string using the input bytes
        let value = std::str::from_utf8(value).unwrap();

        // trim whitespaces
        let value = &value.trim();

        // remove all punctuation breaks (e.g. ". ")
        let value = self.punc_matcher.replace_all(&value, "$1");

        // compress all sequential spaces into a single space
        let value = self.multi_spaces.replace_all(&value, " ");

        // split on spaces to find words
        for word in value.split(" ") {
            // write each word
            ctx.write_fmt(word, 1);
        }
    }
}
```

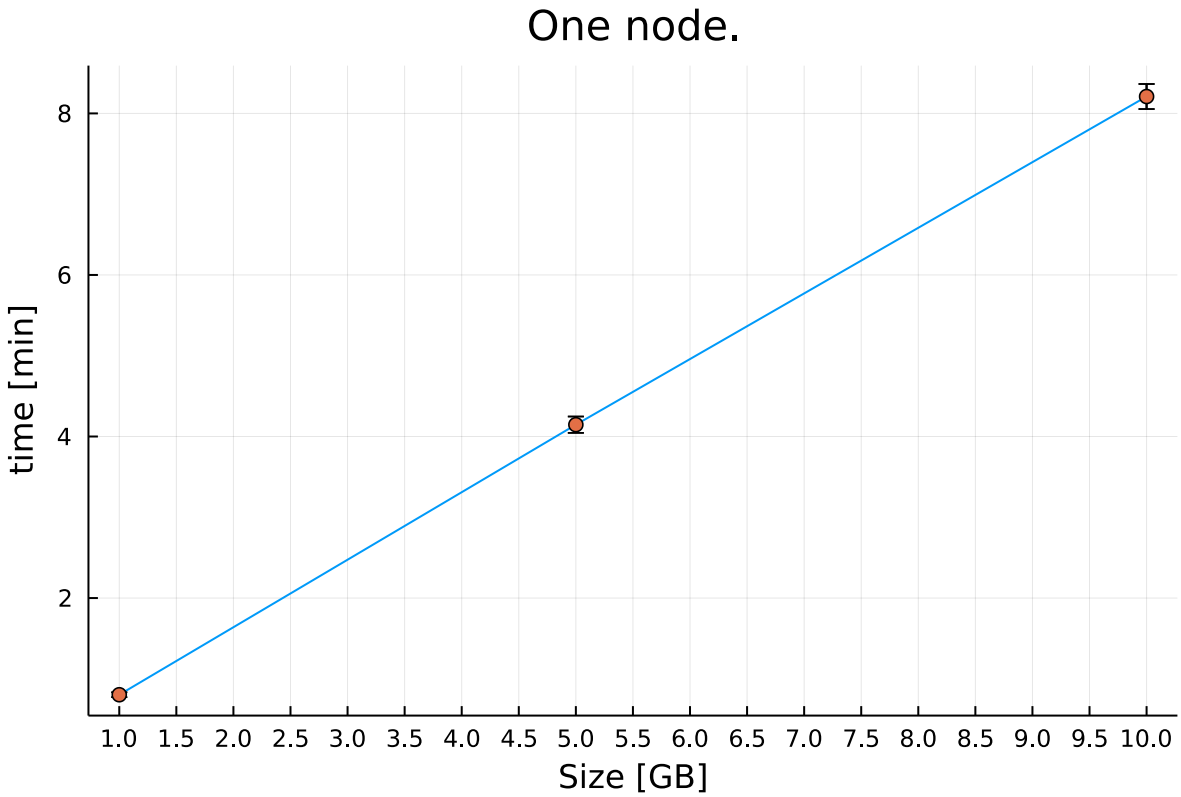
Reducer:

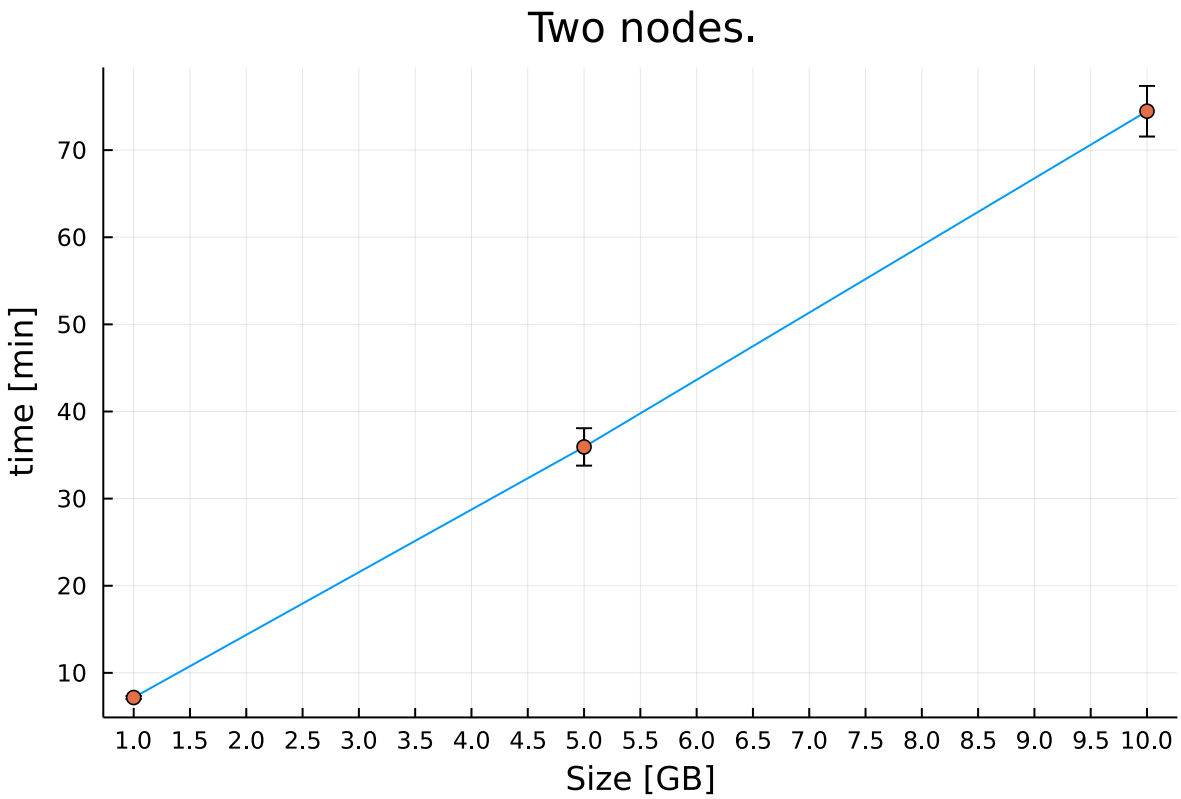
```
extern crate efflux;

use efflux::prelude::{Context, Reducer};

fn main() {
    // simply run the reduction phase with our reducer
    efflux::run_reducer(WordcountReducer);
}

/// Simple struct to represent a word counter reducer.
struct WordcountReducer;

// Reducing stage implementation.
impl Reducer for WordcountReducer {
    fn reduce(&mut self, key: &[u8], values: &[&[u8]], ctx: &mut Context) {
        // base counter
        let mut count = 0;

        for value in values {
            // parse each value sum them all to obtain total appearances
            let value = std::str::from_utf8(value);
            if value.is_err() {
                continue;
            }

            let value = value.unwrap().parse::<usize>();
            if value.is_err() {
                continue;
            }

            count += value.unwrap();
        }

        // write the word and the total count as bytes
        ctx.write(key, count.to_string().as_bytes());
    }
}
```

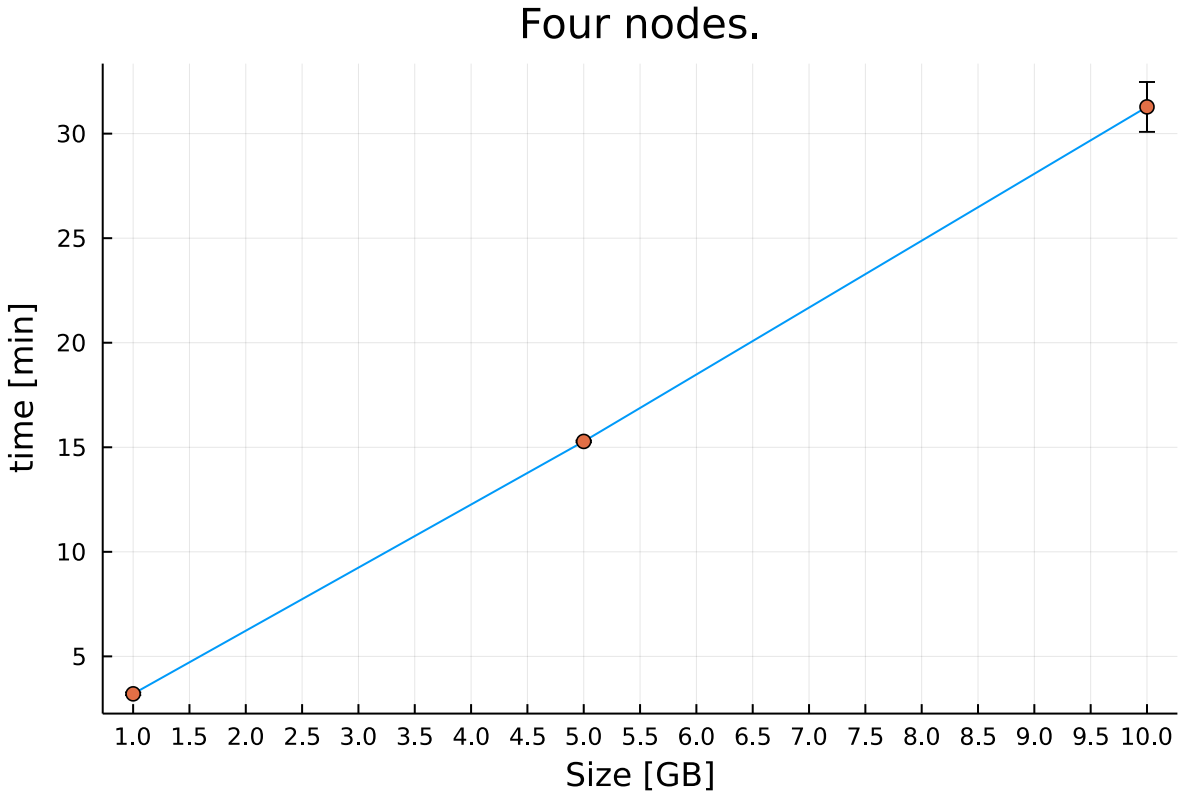
Uzyskane czasy

```
parsetime (generic function with 1 method)
```

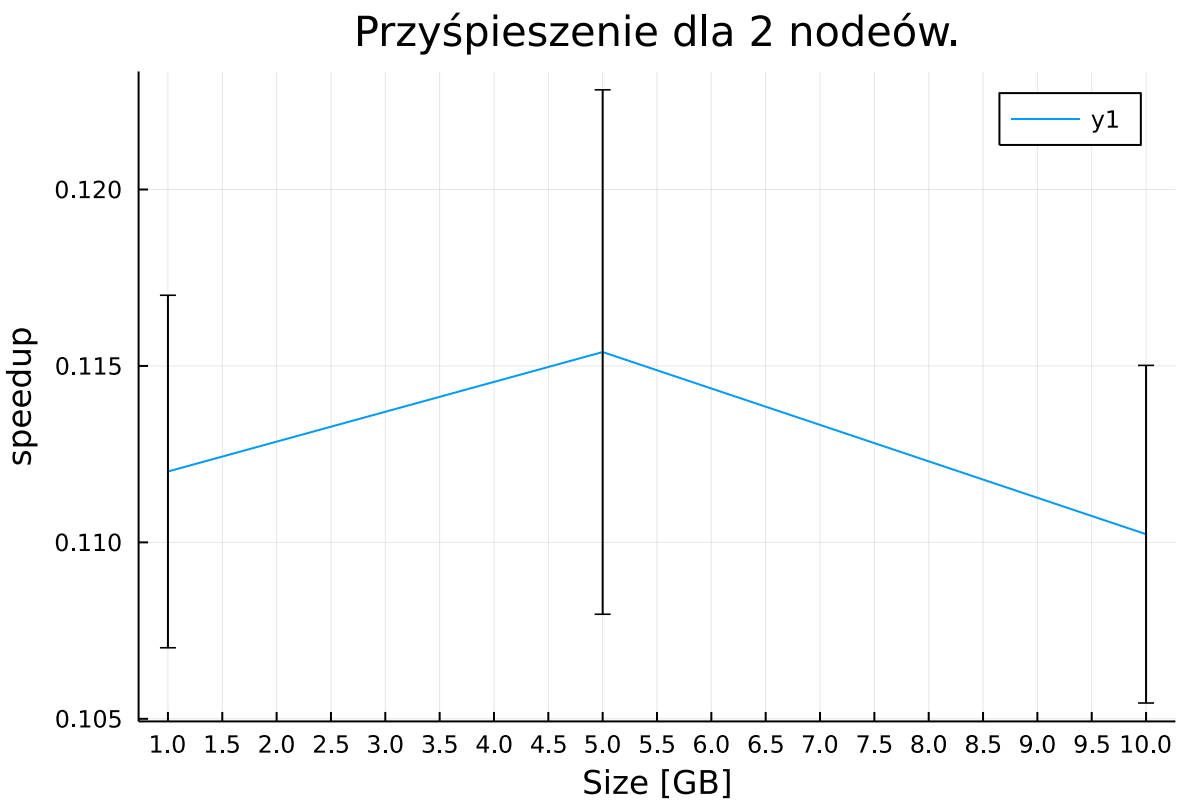

Czasy dla poszczególnych konfiguracji.
1
2
4
time [min]
0
20
40
60
1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
Size [GB]

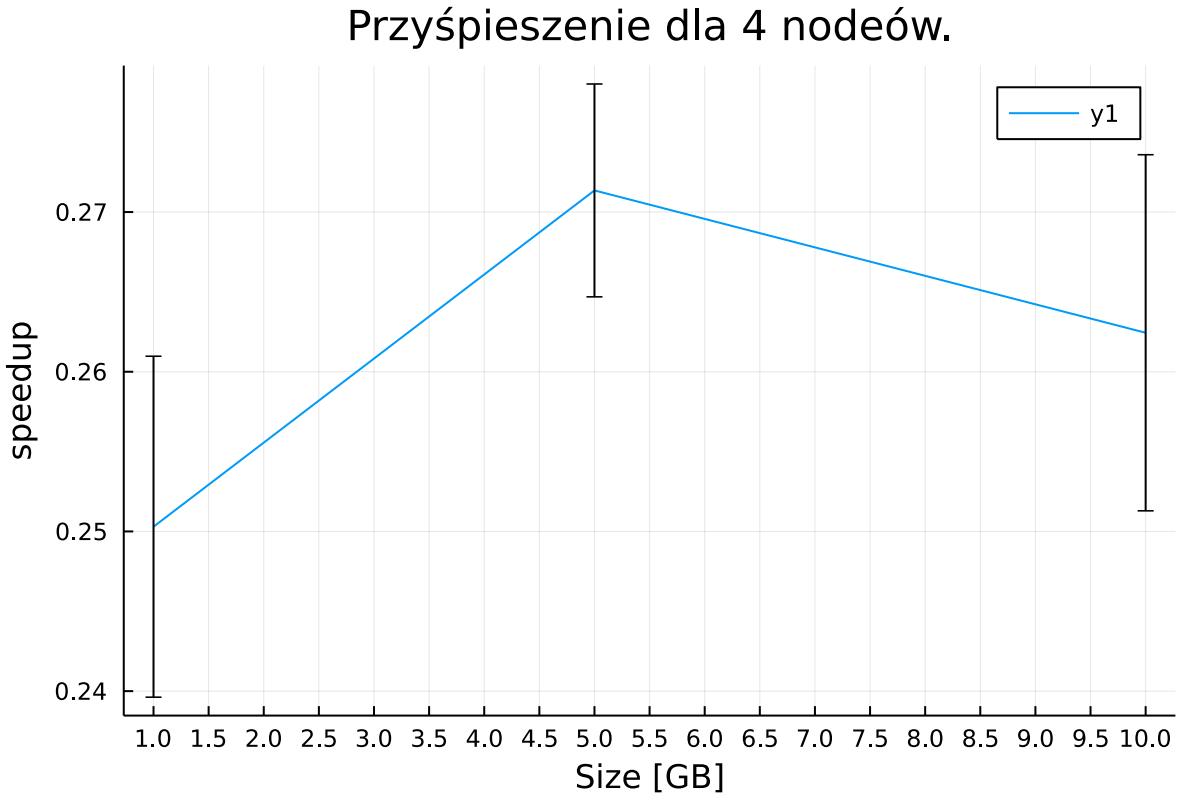

One node.
time [min]
2
4
6
8
1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
Size [GB]





Wykresy przyśpieszenia


Przyśpieszenie dla 2 nodeów.
y1
speedup
0.120
0.115
0.110
0.105
1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
Size [GB]


Przyśpieszenie dla 4 nodeów.
y1
speedup
0.27
0.26
0.25
0.24
1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
Size [GB]

Podsumowanie

Widzimy, że sekwencyjna wersja programu okazła się być najszybsza. Nie jest to zaskakujące ze względu na ilość operacji IO w przypadku map-reducea. Widzimy również iż mimo, że czas sekwencyjnego programu jest lepszy to map-reduce całkiem nieźle się skaluje wraz z dodawaniem kolejnych maszyn.

Dla rozmiaru danych testowanych tutaj 5Gb, 10Gb synchroniczna wersja programu osiągała znacznie lepsze wyniki ale dla 100x większych danych konieczne byłoby wykorzystanie systemu takiego jak map-reduce.

Wygląda na to, że COST w naszym przypadku to 16 nodeów (zakładając brak dużego wzrostu czasu wykonywania ze względu na IO).