FFT - Fast Fourier Transform

Użycie

W katalogu projektu:

```
$ source source /opt/nfs/config/source_bupc_2022.10.sh

# Kompilacja i uruchamianie:
$ make compile-bupc
$ make run
```

Specyfikacja pliku wejściowego / liczby node'ów / liczby wątków:

```
$ make run PTHREADS=4 NODES=4 INPUT=example2
```

gdzie `PTHREADS` odpowiada liczbie wątków na node (domyślnie i zalecanie 4), a `NODES` liczbie wykorzystanych node'ów. Całkowitą liczbą wątków będzie ich iloczyn.
Uwaga: obie liczby **muszą** być potęgami 2.
Generacja nowego pliku wejściowego:

```
$ python3 signalGenerate.py liczba_pkt nazwa_pliku
```

Uwaga: do nazwy pliku zostanie dodany przedrostek `generated_`, aby automatyczne czyszczenie je usuwało, a plików przykładowych już nie.

Porównanie wykresów dla danych i dla otrzymanych wyników:

```
$ python3 compare.py nazwa_pliku_expected fft_output
```

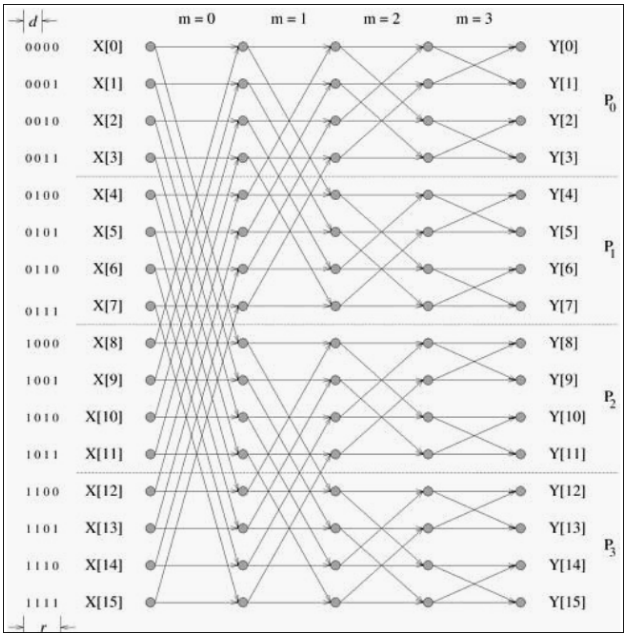
Uwaga: dla danych z `example` i `example2` porównywanie zajmuje paredziesiąt sekund. Dla danych `hard_example` nie polecamy uruchamiać, służą tylko jako performance benchmark.
Czyszczenie katalogu do stanu pierwotnego:

```
$ make clean
```

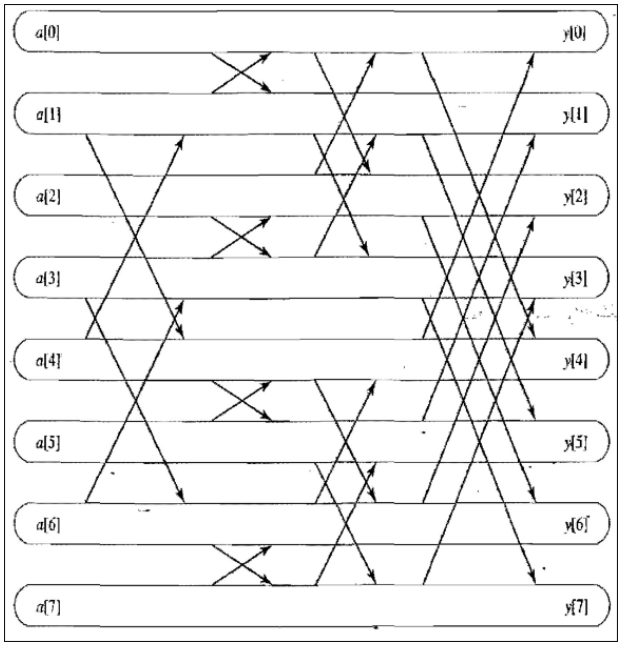
Przy dłuższym testowaniu warto co jakiś czas usuwać plik `nodes` powstający przy uruchamianiu którejkolwiek wersji (automatyczne jest tylko tworzenie, gdy go nie ma).

Działanie i budowa programu

Zaimplementowaliśmy wersję *radix-2 Cooley–Tukey FFT*, dodatkowo z wykorzystaniem *bit-reversal permutation* dla zoptymalizowania użycia pamięci. Na figurze 1 umieszczono teoretyczne diagramy tego algorytmu zaczerpnięte z literatury (na części *a)* bez *bit-reversal*, a na części *b)* z. Zdecydowaliśmy się na wybór technologii *Berkeley UPC* jako implementacja *PGAS*, ponieważ pozwala ona na implementację algorytmu w praktycznie identyczny sposób jak zwyczajne wersje wielowątkowe dostępne w literaturze. Na figurze 2 znajduje się schemat z działaniem programu w wersji `MPI`, a na figurze **??** schemat poglądowy z samą częścią algorytmiczną wersji `UPC`. Jak widać jest on praktycznie identyczny jak schematy poglądowe z literatury.



(a) standardowe *FFT* (Grama, Gupta)



(b) *bit-reversed FFT* (Quinn)

Figura 1: 2 Figures side by side

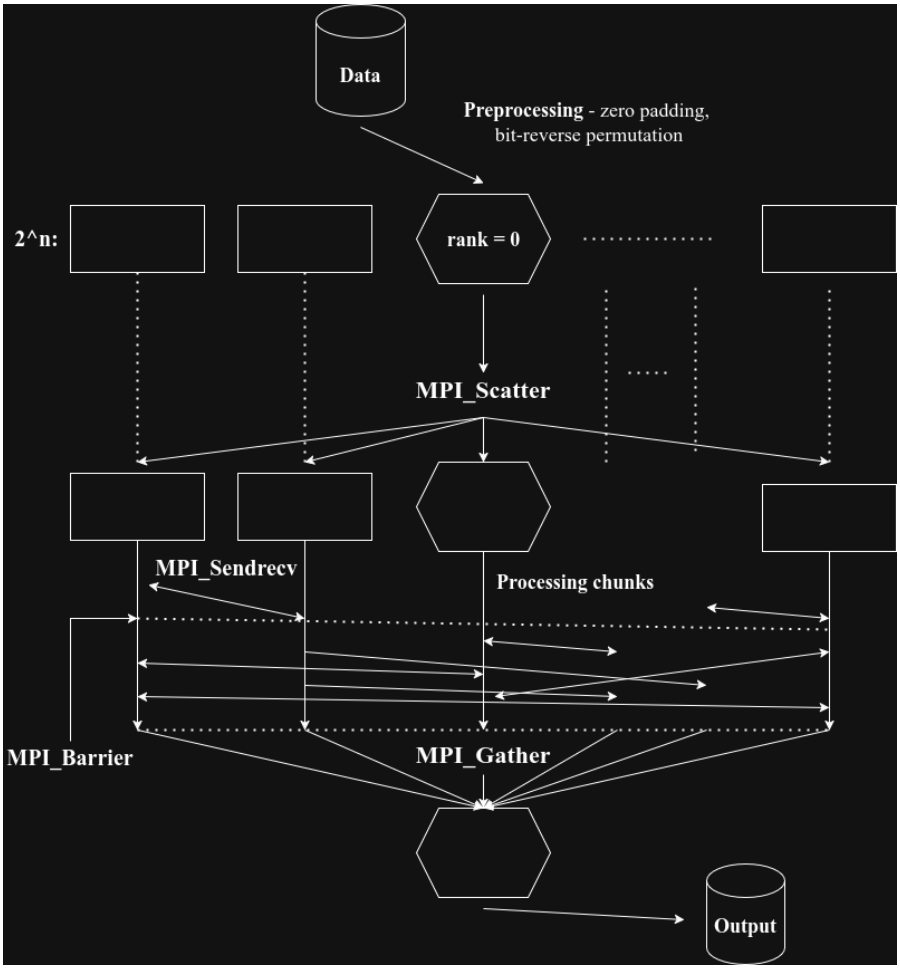


Figura 2: Przybliżony schemat blokowy programu MPI

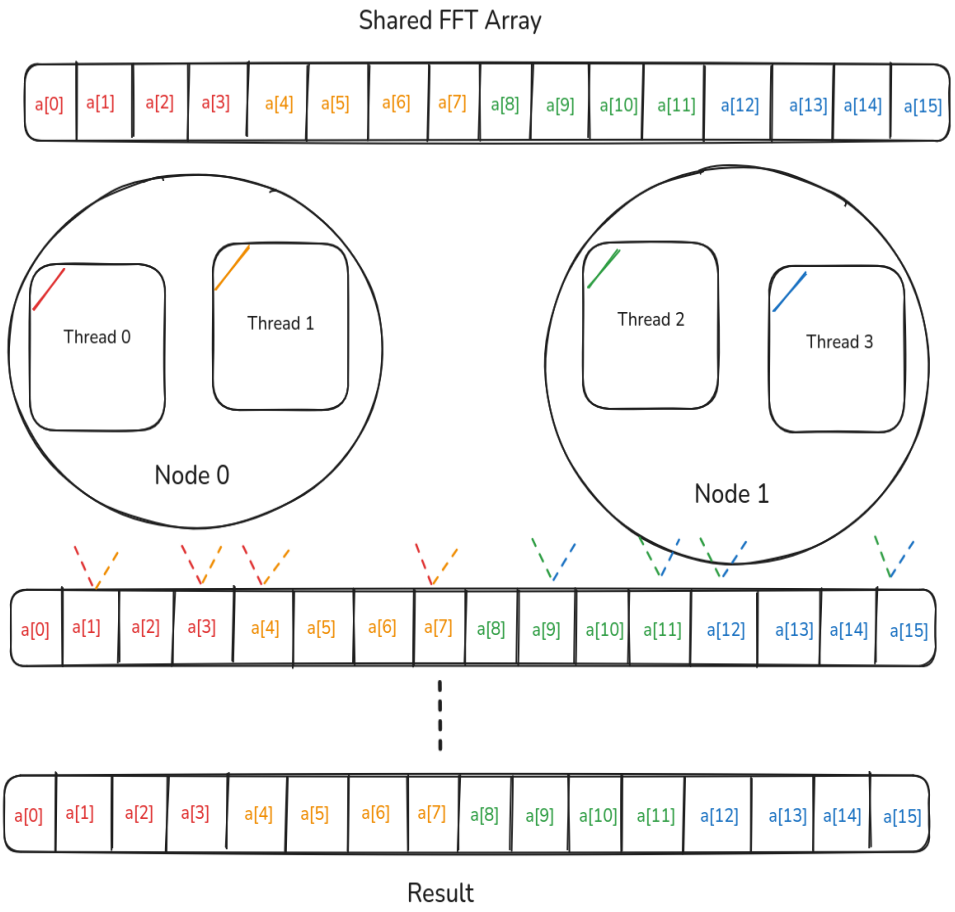


Figura 3: Przybliżony schemat blokowy programu UPC

Krótki opis programu

Pogram na wejściu przyjmuje wygenerowany plik zawierający liczbę próbek n, częstotliwość próbkowania i n próbek z wygenerowanago sygnału jednowymiarowego. Następnie, proces-klient ($rank = 0$) dokonuje paddingu próbek zerami, tak żeby nowe n było najbliższą, większą lub równą potegą 2 - algorytm `FFT` jest algorytmem typu dziel i zwyciężaj, który w każdym kroku dzieli problem na 2 dokładnie dwa razy mniejsze, więc dodawanie niekosztowynych danych (zera nie wpływają na wynik, poza jego potencjalną reprezentacją) jest warte łatwiejszych obliczeń na potęgach 2. Następnie, wykonywana jest permutacja *bit-reversal* wektora próbek w celu umożliwienia algorytmowi działania *in-place*.Dane przechowywane są w dzielonej tablicy, gdzie każdy wątek jest odpowiedzialny za obliczanie i zapis zmiennych do swojego fragmentu. Wątki wykonują swoją część algorytmu `FFT` w wersji iteracyjnej (a zatem *bottom-up*), sumując odpowiednie pary wartości (początkowo próbek) zgodnie ze standardowym algorytmem `FFT`. Jeśli jakaś wartość pochodzi z części tablicy nienależącej do wątku, to w przeciwieństwie do wersji `MPI`, wątek nie zwraca na to uwagi, ponieważ wykonuje na niej tylko odczyt - wątek odpowiedzialny za tę zmienną analogicznie odczyta zmienną pierwszego wątku, ale każdy z nich zapisuje tylko swoje - *binary exchange algorithm*. Program jest blokowany - `upc_barrier;` - po każdej iteracji, aby uniknąć wypadku gdzie czytana jest stara wartość tablicy. Gdy w każdym procesie iteracja dobiegnie końca, wątek 0 kopiuję wartości z tablicy wspólnej do lokalnej, a wynikowy wektor zostaje zapisywany w postaci par części rzeczywistych i urojonych wyniku.