FFT - Fast Fourier Transform

Użycie

W katalogu projektu:

```
$ source source /opt/nfs/config/source_mpich430.sh
$ source /opt/nfs/config/source_cuda121.sh

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

Specyfikacja pliku wejściowego / liczby procesów (2^k):

```
$ make run NUM_PROCESSES=16 INPUT=example2
```

Generacja nowego pliku wejściowego:

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

Uwaga - do nazwy pliku zostanie dodany przedrostek `generate\_`, 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
```

Wizualizacja działania programu przy pomocy `MPE` i `jumpshot`:

```
$ make compile_mpe
# Opcje jak dla ‘make run‘:
$ make run_mpe
$ make convert
$ make jumpshot
```

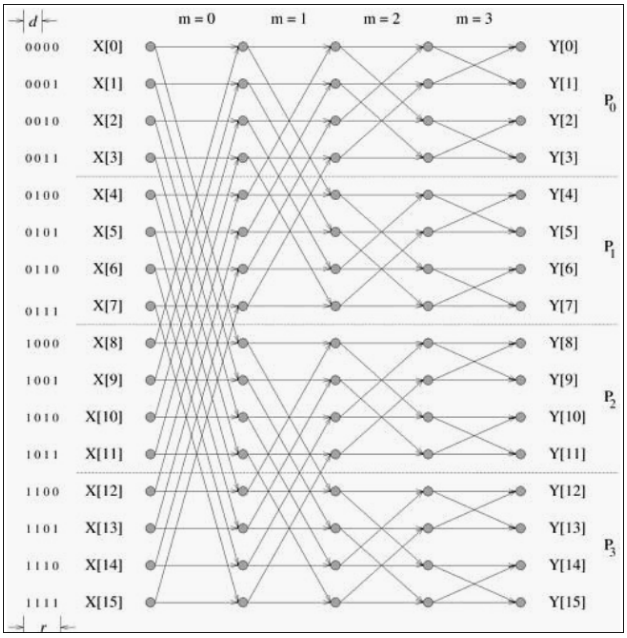
Czyszczenie katalogu do stanu pierwotnego:

```
$ make clean
```

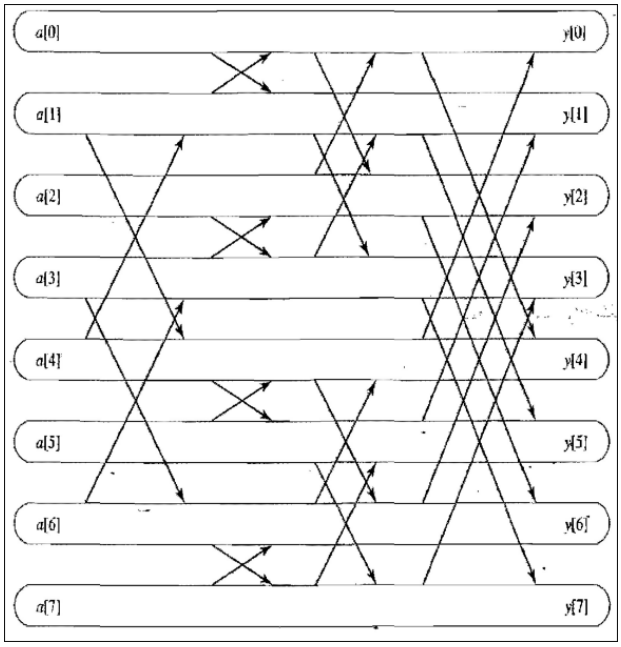
Przy dłuższym testowaniu warto co jakiś czas usuwać plik `nodes` powstający przy uruchamianiu którejkolwiek wersji (automatyczne jest tylko tworzenie, gdy go nie ma).

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. Natomiast na figurze 2 znajduje się bardziej praktyczny schemat z przepływem danych i dyrektywami `MPI`.



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



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

Figura 1: 2 Figures side by side

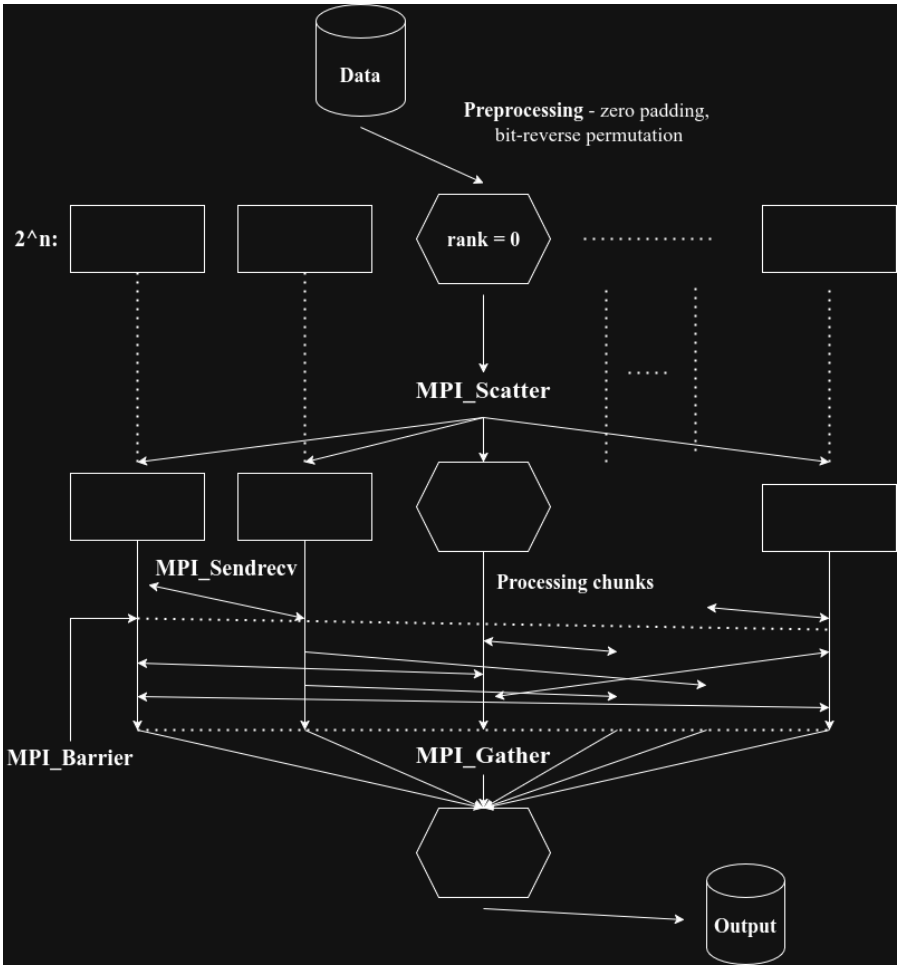


Figura 2: Przybliżony schemat blokowy programu

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*. W tym momencie wektor próbek jest równo rozdzielany pomiędzy dostępne procesy (zakładamy liczbę procesów również będącą potęga 2) - `MPI_Scatter`. Każdy proces wykonuje 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 nie jest dostępna dla procesu to wysyła dwustronne zapytanie `MPI_Sendrecv` do procesu, który ją posiada (zapytanie jest dwustronne, bo w tej implementacji drugi proces będzie również potrzebował wartości od pierwszego - *binary-exchange algorithm*). Program jest blokowany - `MPI_Barrier` - po każdej iteracji, aby uniknąć niepożądanego zachowaniu przy wymianie wiadomości. Gdy w każdym procesie iteracja dobiegnie końca, wynikowe fragmenty nowego wektora są łączone w całość dzięki dyrektywie `MPI_Gather` i wynikowy wektor zostaje zapisywany w postaci par części rzeczywistych i urojonych wyniku.