HPC MPI 2.

Wojciech Raczuk 459487

Optymalizacje

Napisałem implementacje optymalizacji:

-edge classification

-pruning (push / pull)

-hybrydyzacja

-redukcja powtórzeń (w stylu SPFA)

```
it_cnt++;
for (int u : active) {

    #ifdef REMOVE_REPETITIONS
    if (when_relaxed[u - low] == it_cnt)
        continue;
    when_relaxed[u - low] = it_cnt;
    #endif // REMOVE_REPETITIONS
```

Optymalizacje są niezależne (o ile to możliwe) i włącza się je flagami.

Optymalizacje

Oprócz tego, miejscami używam parallel forów z OpenMP, do równiejszego podziału pracy.

Ze względu na ogromny rozmiar danych, zaimplementowałem szybkie wczytywanie, co znacząco poprawia stałą, zmniejszając czas wczytywania 2,5x.

```
    long_edges.resize(high - low + 1);
    #pragma omp parallel for
    for (int i = 0; i < high - low + 1; i++) {
        auto it = partition(edges[i].begin(), edges[i].end(), [](pair<int, l
        long_edges[i].reserve(edges[i].end() - it);
        for_each(it, edges[i].end(), [&](pair<int, long long> a){ long_edges
        edges[i].resize(it - edges[i].begin());
    }
```

```
unsigned long long read_ulong() {
    unsigned long long num = 0;
    int c = getchar_unlocked();
    if (c == EOF) {
        return (unsigned)-1;
    }

    while (c < '0' || c > '9') {
        c = getchar_unlocked();
    }

    while (c >= '0' && c <= '9') {
        num = num * 10 + (c - '0');
        c = getchar_unlocked();
    }

    return num;
}
```

Pruning

Moją uwagę zwrócił wzór: $\deg(v) \times \frac{d(v)-(k+1)\Delta}{w_{\max}}$.

Więcej sensu ma dla mnie MINowanie tego ułamka z 1.

Komunikację w pullach wykonuje przez RMA.

```
for (auto i : it->second) {
    for (const auto& e : long_edges[i - low]) {
        if (k * DELTA + e.second < dist[i - low]) {
            batch[pos] = {.edge_len = e.second, .to = i};
            MPI_Get(&batch[pos].dist, 1, MPI_LONG_LONG, whose
            if (++pos == BATCH_SIZE)
                free_batch();
        }
    }
}
```

Pomiary

Pomiarów dokonywałem dla grafu mającego 10^6 wierzchołków i $16 * 10^6$ za każdy wątek (należy zwrócić uwagę, że choć każdy wątek dostanie tyle samo wierzchołków, to mogą różnić się liczbą krawędzi). Inspirując się pracą, wybrałem deltę równą w_max / 5 i współczynnik do hybrydyzacji 0.4.

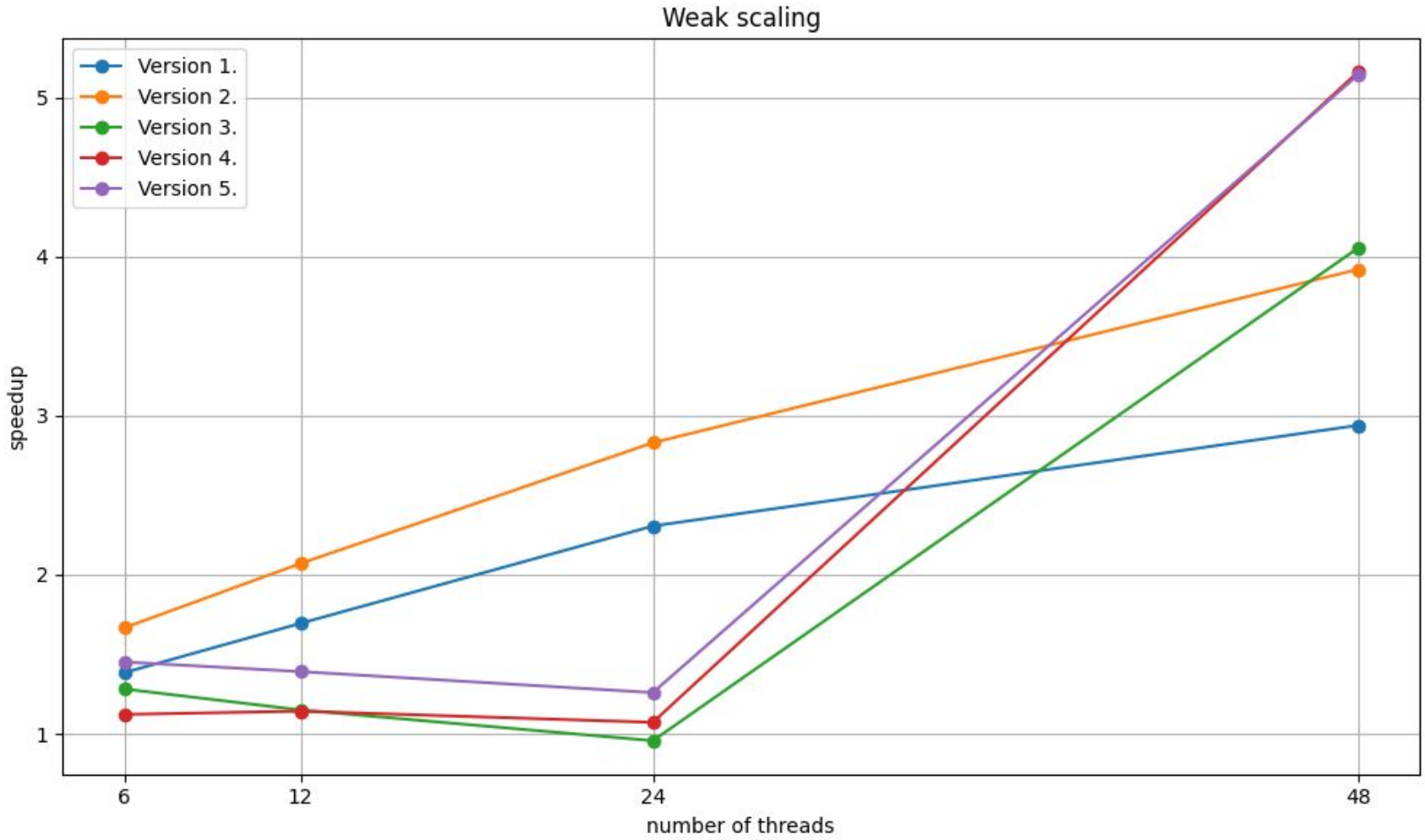
Graf budowałem skryptem z https://github.com/farkhor/PaRMAT i parametrami A=0.5, B=C=0.1 (RMAT-2).

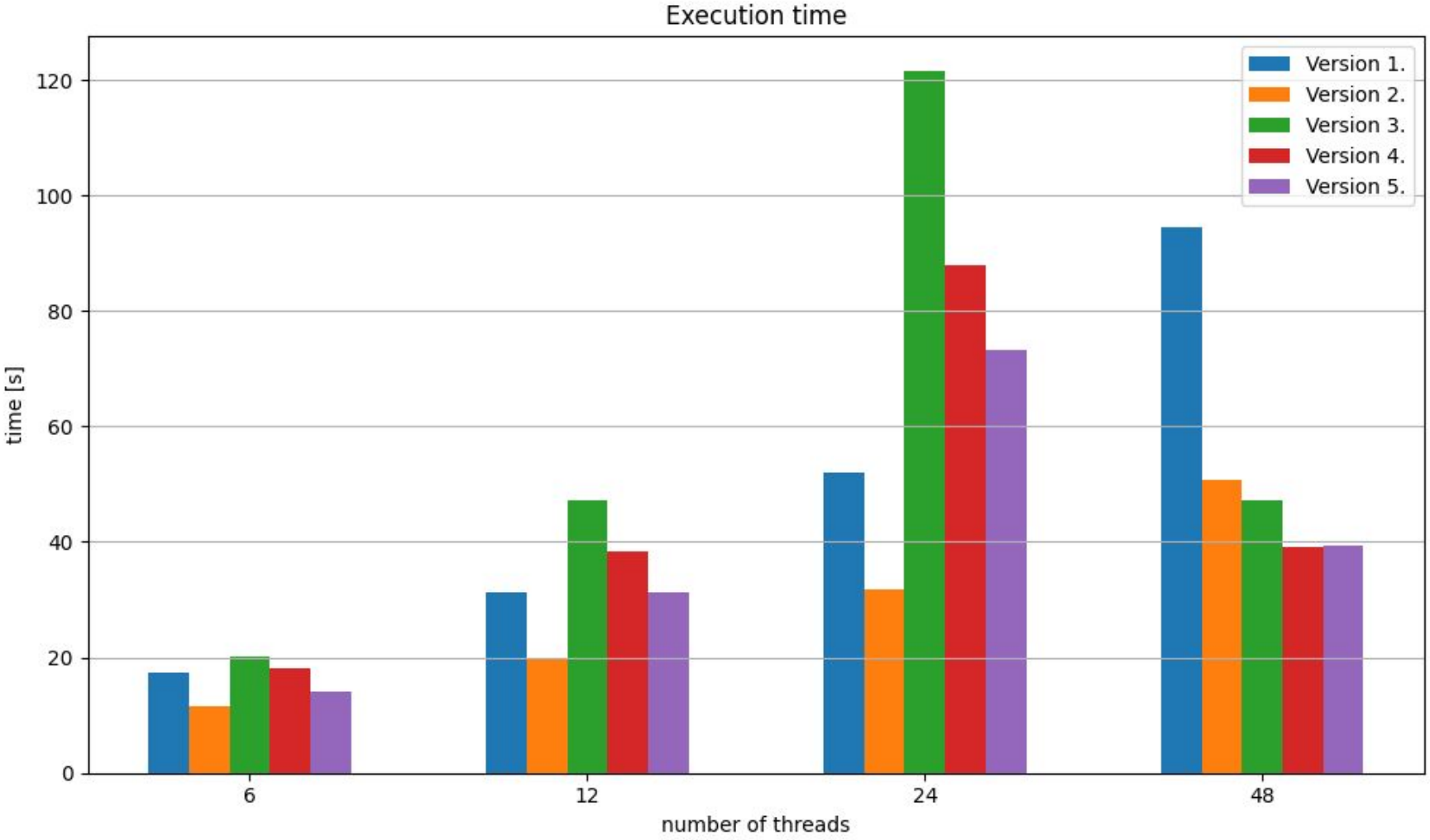
Pomiary

Zmierzyłem speedup dla kolejnych 5 wersji:

1. naiwna
2. naiwna + pozbywanie się powtórzeń
3. edge classification + pozbywanie się powtórzeń
4. pruning + pozbywanie się powtórzeń
5. pruning + hybrydyzacja + pozbywanie się powtórzeń

Speedup mierzyłem licząc sssp dla tego samego grafu na 1 wątku oraz wielu wątkach, a następnie dzieląc te wartości.


Weak scaling
Version 1.
Version 2.
Version 3.
Version 4.
Version 5.
speedup
1
2
3
4
5
6
12
24
48
number of threads


Execution time
Version 1.
Version 2.
Version 3.
Version 4.
Version 5.
time [s]
0
20
40
60
80
100
120
6
12
24
48
number of threads

Wnioski

Rozwiązanie się skaluje, choć zaobserwowany współczynnik jest niższy niż liczba wątków. Prawdopodobnie jest to spowodowane nierównym podziałem pracy - przede wszystkim przy wczytywaniu danych. Najszybsza implementacja na 48 wątkach wykonuje się 38 sekund, z czego 26 to wczytywanie grafu.

Rzeczywisty czas wykonania dla implementacji naiwnej rośnie ~liniowo - ponieważ zależy to od maksymalnej pracy pojedynczego wątku, ma to sens. Zastanawiające jest zachowanie pozostałych implementacji - np. dlaczego czasy są dużo większe na 24 wątkach niż 48. Moim przypuszczeniem jest, że te 24 wątki nie były na 1 nodzie. Wtedy w wersji naiwnej, ponieważ relaksacje krótkich i długich krawędzi robimy razem, to sumarycznie jest dużo mniej różnych transferów, które pomiędzy nodami są bardziej kosztowne.