Aleksandra Jarzyńska **136722** L8
Grzegorz Bryk **136686** L8
Termin zajęć: środa 9:45
Data: 27.04.2020 [*wymagany termin: 27.04.2020*]
WERSJA PIERWSZA
mail: aleksandra.w.jarzynska@student.put.poznan.pl
grzegorz.bryk@student.put.poznan.pl

Przetwarzanie równoległe – laboratorium

Projekt nr 1: Analiza efektywności przetwarzanie równoległego realizowanego w komputerze równoległym z procesem wielordzeniowym z pamięcią współdzieloną na podstawie problemu znajdowania liczb pierwszych w podanym jako parametr przedziale.

1. Procesor: *i7-8750H*

Liczba procesorów fizycznych	Liczba procesorów logicznych	Liczba uruchamianych wątków w systemie	Typ procesora	Wielkość i organizacja pamięci podręcznych	Wielkość i organizacja bufora translacji adresów
6	12	2 * 6	CPU / Microprocessor	9 MB Intel® Smart Cache L1 384KB L2 1.5MB L3 9MB	64-byte Prefetching

System operacyjny	IDE	Wersja VTune
Windows 10	Visual Studio 2019	2020 Update 1

Potencjalne problemy efektywnościowe:

- **false sharing** – zjawisko, które jest odpowiedzialne za niezamierzone/fałszywe współdzielenie danych. Polega na tym, że wiele procesów modyfikuje zmienne, które znajdują się na tej samej linii danych pamięci podręcznej procesora. False sharing najłatwiej zaobserwować, kiedy użyjemy zmiennych typu **volatile**. Typ ten przestrzega przed tym, że zmienna może być modyfikowana i w pewien sposób zapewnia użytkownikowi synchronizację – broni przed odczytaniem nieaktualnej wartości takiej zmiennej, co jest wymuszone poprzez aktualizację całej linii danych pamięci podręcznej w każdym procesie, który z niej korzysta – zaraz po zapisie do tej linii danych przez jakikolwiek z nich. Łatwo zauważyć, że każde żądanie dostępu będzie zajmowało znacznie więcej (o około rząd jednostki) czasu, niż w przypadku, kiedy nie używamy typu volatile, należy jednak pamiętać, że narażamy wtedy pozostałe procesy na odczytanie nieaktualnej wartości (w przypadku jeśli rzeczywiście korzystają z tej samej zmiennej). **False sharing** występuje, kiedy procesy korzystają z **tej samej linii danych** pamięci podręcznej, ale **modyfikują zmienne, które są niezależne** (zapis do zmiennej jest traktowany jako zapis do konkretnej linii danych, co w każdym procesie, który korzysta z tej linii wymusza odświeżenie).
- **Synchronizacja** – podobnie jak w opisanym wcześniej przypadku false sharingu (gdzie również mamy do czynienia z synchronizacją – dopiero po zapisie i uaktualnieniu wartości może nastąpić jej odczyt), synchronizacja jest realnym problemem efektywnościowym. Dostęp synchroniczny do niektórych części pamięci jest wielokrotnie **niezbędny** do poprawnego funkcjonowania programu, jednocześnie umieszczanie pewnych operacji w sekcji krytycznej jest bardzo **kosztowne czasowo**.

Potencjalne problemy poprawnościowe:

- **wyścig** – warunkiem wystąpienia wyścigu jest dostęp dwóch wątków do tej samej (współdzielonej – shared) zmiennej w **tym samym czasie**. Pierwszy wątek odczytuje wartość zmiennej – jednocześnie drugi wątek odczytuje taką samą wartość. Oba wątki wykonują działania na tej wartości zmiennej, więc ten, który **skończy pracę później** (więc później nadpisze jej wartość) **wygrywa wyścig**. Mamy tutaj do czynienia z działaniem wątku, które nie przynosi żadnego wymiernego efektu, a zużywa ograniczone zasoby komputera.

2. Warianty kodu.

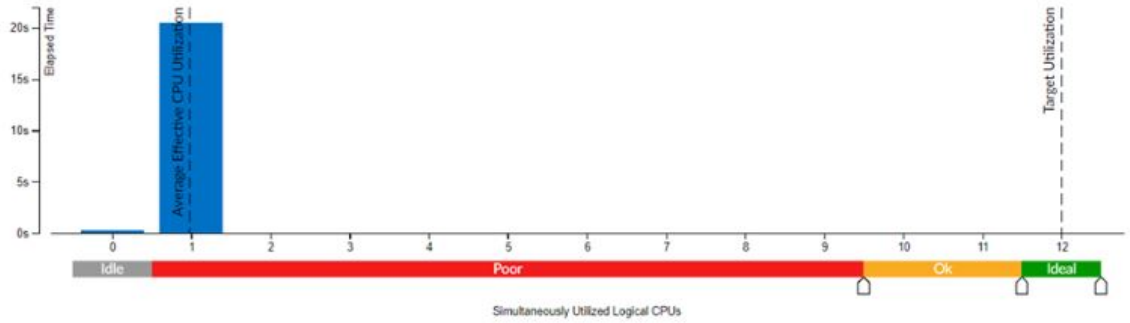
- Wariant sekwencyjny.

```
22  std::set<int> first_sequential(int min, int max) {
23      std::set<int> result;
24
25      if (min == 2)
26          result.insert(2);
27
28      for (int i = min % 2 ? min : min + 1; i <= max; i += 2) {
29          for (int j = 2; j <= ceil(sqrt(i)); j++) {
30              if (i % j == 0) {
31                  break;
32              } else if (j == ceil(sqrt(i))) {
33                  result.insert(i);
34              }
35          }
36      }
37
38      return result;
39  }
```

Effective CPU Utilization ⓘ: 8.2% (0.982 out of 12 logical CPUs)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



Total Thread Count: 1

Thread Oversubscription ⓘ: 0s (0.0% of CPU Time)

Wait Time with poor CPU Utilization ⓘ: 0.004s (100.0% of Wait Time)

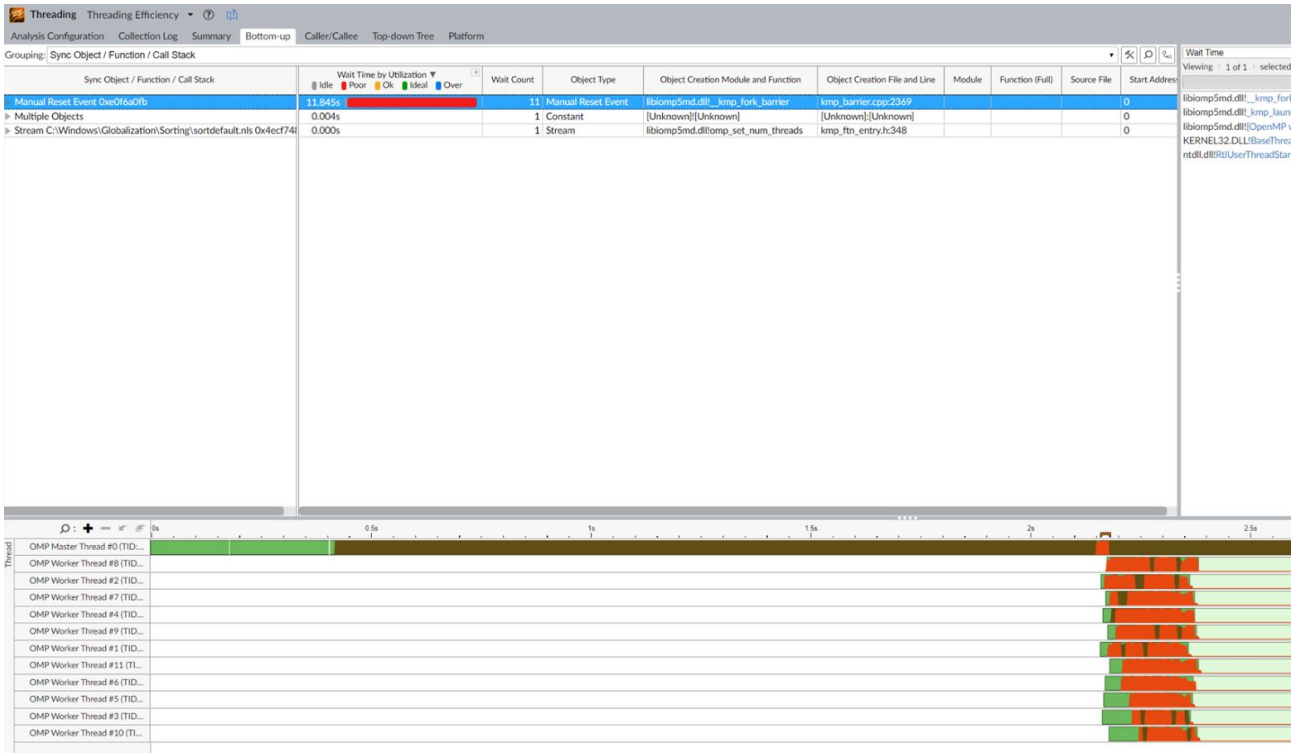
Top Waiting Objects

This section lists the objects that spent the most time waiting in your application. Objects can wait on specific calls, such as sleep() or I/O, or on contended synchronizations.

Sync Object	Wait Time with poor CPU Utilization ⓘ	(% from Object Wait Time) ⓘ	Wait Count ⓘ
Multiple Objects	0.004s	100.0%	2
Stream C:\Windows\Globalization\Sorting\sortdefault.nls 0x4ecf7487	0.000s	100.0%	1

*N/A is applied to non-summable metrics.

Spin and Overhead Time ⓘ: 0s (0.0% of CPU Time)



Jest to implementacja opisanej przez Prowadzącego pierwszej wersji podejścia koncepcyjnego – każda liczba z podanego na wejściu przedziału jest testowana pod względem podzielności przez liczby mniejsze. Zgodnie z sugestią ograniczamy od góry liczbę sprawdzanych dzielników przez pierwiastek kwadratowy z aktualnie badanej liczby.

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony pierwszy.

```
74  std::set<int> parallel_1(int min, int max) {
75      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
76      std::set<int> initial;
77
78      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
79      if (min == 2) initial.insert(initial.end(), 2);
80
81      std::set<int> to_erase[THREADS_NUM];
82
83  #pragma omp parallel for default(none) private(primary_numbers, max) shared(to_erase)
84      for (int x: primary_numbers) {
85          int tn = omp_get_thread_num();
86          for (int y = x; y * x < max; y += 2)
87              to_erase[tn].insert(x * y);
88  //#pragma omp critical
89      }
90
91      for (const auto &te: to_erase) for (int e: te) initial.erase(e);
92      return initial;
93  }
```

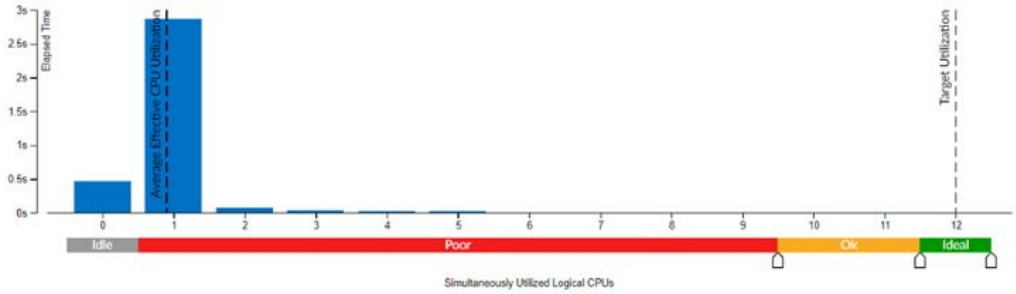
Elapsed Time: 3.450s

Paused Time: 0s

Effective CPU Utilization: 7.5% (0.906 out of 12 logical CPUs)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



OpenMP Analysis. Collection Time: 3.450

Serial Time (outside parallel regions): 3.426s (99.3%)

Top Serial Hotspots (outside parallel regions)

This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most CPU time. Improve overall application performance by

Function	Module	Serial CPU Time
[ucrtbase.dll]	ucrtbase.dll	1.164s
func@0x180011390	ucrtbase.dll	0.884s
_Erase_tr	PUT-PR2.exe	0.329s
_Insert_no	PUT-PR2.exe	0.318s
parallel_1	PUT-PR2.exe	0.139s
[Others]	N/A*	0.129s

*N/A is applied to non-summable metrics.

Parallel Region Time: 0.024s (0.7%)

Total Thread Count: 12

Thread Oversubscription: 0s (0.0% of CPU Time)

Wait Time with poor CPU Utilization: 11.849s (100.0% of Wait Time)

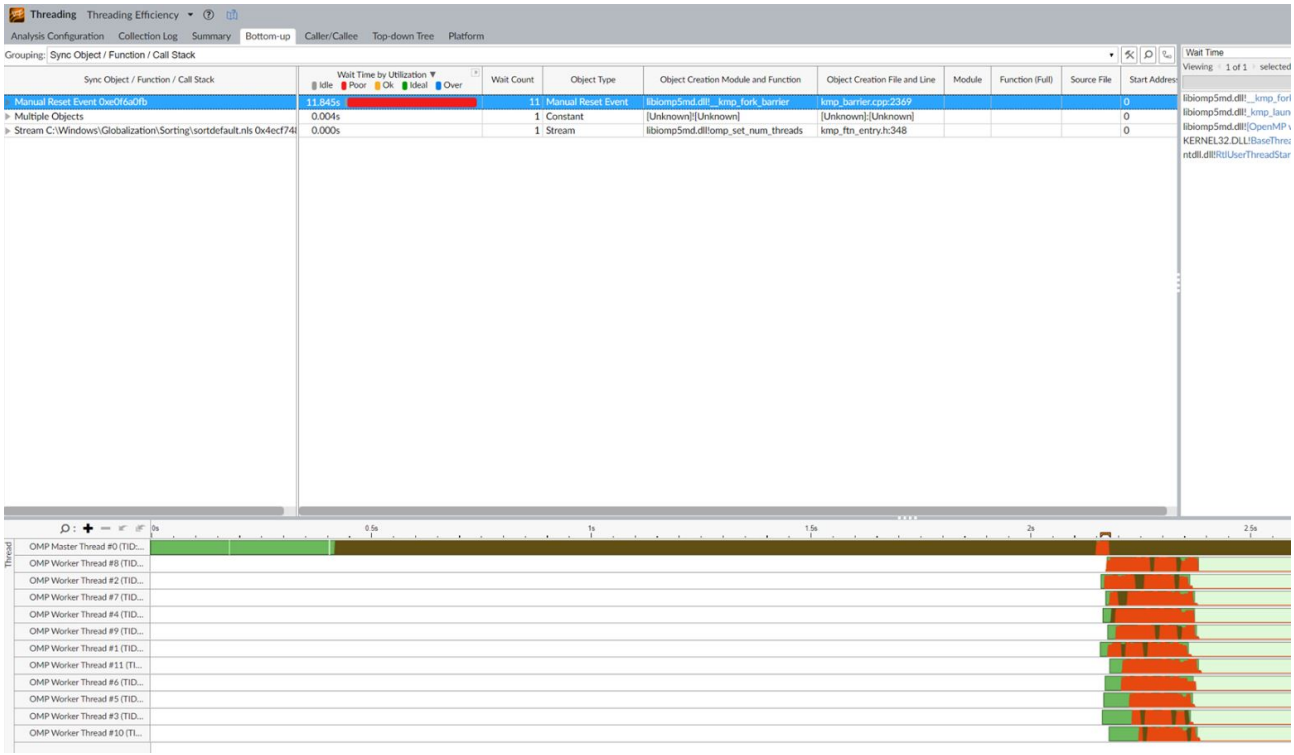
Top Waiting Objects

This section lists the objects that spent the most time waiting in your application. Objects can wait on specific calls, such as sleep() or I/O, or on contended synchronizations. A significant amount

Sync Object	Wait Time with poor CPU Utilization	(% from Object Wait Time)	Wait Count
Manual Reset Event 0xe0f6a0fb	11.845s	100.0%	11
Multiple Objects	0.004s	100.0%	1
Stream C:\Windows\Globalization\Sorting\sortdefault.nls 0x4ecf7487	0.000s	100.0%	1

*N/A is applied to non-summable metrics.

Spin and Overhead Time: 1.655s (34.6% of CPU Time)



W tej modyfikacji wszystkie wątki współdzielą zbiór wykreśleń, natomiast każdy wątek obrabia efektywnie część zbioru liczb pierwszych. Do wyznaczenia zbioru liczb pierwszych w tej wersji kodu (podobnie jak we wszystkich poniższych) wykorzystujemy funkcję ***first_sequential_v***, która zwraca wektor (**std::vector<int>**) liczb pierwszych znajdujących się w podanym jako parametr przedziale.

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony drugi.

```
95  std::set<int> parallel_2(int min, int max) {
96      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
97      std::set<int> initial;
98
99      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
100     if (min == 2) initial.insert(initial.end(), 2);
101
102 #pragma omp parallel for default(none) private(primary_numbers, max) shared(initial)
103     for (int x: primary_numbers) {
104         std::set<int> to_erase;
105         for (int y = x; y * x < max; y += 2)
106             to_erase.insert(x * y);
107 #pragma omp critical
108         for (int e: to_erase) initial.erase(e);
109     }
110
111     return initial;
112 }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony trzeci.

```
114  std::set<int> parallel_3(int min, int max) {
115      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
116      std::set<int> initial;
117
118      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
119      if (min == 2) initial.insert(initial.end(), 2);
120
121      std::set<int> to_erase;
122
123  #pragma omp parallel for default(none) private(primary_numbers, max, to_erase) shared(initial)
124      for (int x: primary_numbers) {
125          for (int y = x; y * x < max; y += 2)
126              to_erase.insert(x * y);
127  #pragma omp critical
128          for (int e: to_erase) initial.erase(e);
129      }
130
131      return initial;
132  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony czwarty.

```
134  std::set<int> parallel_4(int min, int max) {
135      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
136      std::set<int> initial;
137
138      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
139      if (min == 2) initial.insert(initial.end(), 2);
140
141
142  #pragma omp parallel for default(none) private(primary_numbers, max) shared(initial)
143      for (int x: primary_numbers) {
144          for (int y = x; y * x < max; y += 2)
145              initial.erase(x * y);
146      }
147
148      return initial;
149  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony piąty.

```
151  std::set<int> parallel_5(int min, int max) {
152      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
153      std::set<int> initial;
154
155      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
156      if (min == 2) initial.insert(initial.end(), 2);
157
158
159  #pragma omp parallel for default(none) private(primary_numbers, max) shared(initial) collapse(2)
160      for (int x: primary_numbers) {
161          for (int y = 3; y < max / 3; y += 2)
162              initial.erase(x * y);
163      }
164
165      return initial;
166  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony szósty.

```
174   std::set<int> parallel_6(int min, int max) {
175       std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
176       std::set<int> initial;
177
178       for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
179       if (min == 2) initial.insert(initial.end(), 2);
180
181
182   #pragma omp parallel for default(none) private(primary_numbers, max) shared(initial) schedule(static)
183       for (int x: primary_numbers) {
184           for (int y = x; y * x < max; y += 2)
185               initial.erase(x * y);
186       }
187
188       return initial;
189   }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony siódmy.

```
191  std::set<int> parallel_7(int min, int max) {
192      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
193      std::set<int> initial;
194
195      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
196      if (min == 2) initial.insert(initial.end(), 2);
197
198
199  #pragma omp parallel for default(none) private(primary_numbers, max) shared(initial) schedule(static, 100)
200      for (int x: primary_numbers) {
201          for (int y = x; y * x < max; y += 2)
202              initial.erase(x * y);
203      }
204
205      return initial;
206  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony ósmy.

```
209  std::set<int> parallel_8(int min, int max) {
210      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
211      std::set<int> initial;
212
213      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
214      if (min == 2) initial.insert(initial.end(), 2);
215
216
217  #pragma omp parallel for default(none) private(primary_numbers, max) shared(initial) schedule(dynamic)
218      for (int x: primary_numbers) {
219          for (int y = x; y * x < max; y += 2)
220              initial.erase(x * y);
221      }
222      return initial;
223  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony dziewiąty.

```
225  std::set<int> parallel_9(int min, int max) {
226      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
227      std::set<int> initial;
228
229      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
230      if (min == 2) initial.insert(initial.end(), 2);
231
232
233  #pragma omp parallel for default(none) private(primary_numbers, max) shared(initial) schedule(dynamic, 100)
234      for (int x: primary_numbers) {
235          for (int y = x; y * x < max; y += 2)
236              initial.erase(x * y);
237      }
238
239      return initial;
240  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony dziesiąty.

```
242  std::set<int> parallel_10(int min, int max) {
243      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
244      std::set<int> initial;
245
246      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
247      if (min == 2) initial.insert(initial.end(), 2);
248
249
250  #pragma omp parallel for default(none) private(primary_numbers, max) shared(initial) schedule(guided)
251      for (int x: primary_numbers) {
252          for (int y = x; y * x < max; y += 2)
253              initial.erase(x * y);
254      }
255
256      return initial;
257  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony jedenasty.

```
259  std::set<int> parallel_11(int min, int max) {
260      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
261      std::set<int> initial;
262
263      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
264      if (min == 2) initial.insert(initial.end(), 2);
265
266
267  #pragma omp parallel for default(none) private(primary_numbers, max) shared(initial) schedule(guided, 10)
268      for (int x: primary_numbers) {
269          for (int y = x; y * x < max; y += 2)
270              initial.erase(x * y);
271      }
272
273      return initial;
274  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony dwunasty.

```
276  std::set<int> parallel_12(int min, int max) {
277      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
278      std::set<int> initial;
279
280      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
281      if (min == 2) initial.insert(initial.end(), 2);
282
283      std::set<int> to_erase;
284
285  #pragma omp parallel for default(none) private(primary_numbers, max, to_erase) shared(initial) schedule(static)
286      for (int x: primary_numbers) {
287          for (int y = x; y * x < max; y += 2)
288              to_erase.insert(x * y);
289  #pragma omp critical
290          for (int e: to_erase) initial.erase(e);
291      }
292
293      return initial;
294  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony trzynasty.

```
296  std::set<int> parallel_13(int min, int max) {
297      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
298      std::set<int> initial;
299
300      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
301      if (min == 2) initial.insert(initial.end(), 2);
302
303      std::set<int> to_erase;
304
305  #pragma omp parallel for default(none) private(primary_numbers, max, to_erase) shared(initial) schedule(static, 100)
306      for (int x: primary_numbers) {
307          for (int y = x; y * x < max; y += 2)
308              to_erase.insert(x * y);
309  #pragma omp critical
310          for (int e: to_erase) initial.erase(e);
311      }
312
313      return initial;
314  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony czternasty.

```
316  std::set<int> parallel_14(int min, int max) {
317      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
318      std::set<int> initial;
319
320      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
321      if (min == 2) initial.insert(initial.end(), 2);
322
323      std::set<int> to_erase;
324
325  #pragma omp parallel for default(none) private(primary_numbers, max, to_erase) shared(initial) schedule(dynamic)
326      for (int x: primary_numbers) {
327          for (int y = x; y * x < max; y += 2)
328              to_erase.insert(x * y);
329  #pragma omp critical
330          for (int e: to_erase) initial.erase(e);
331      }
332
333      return initial;
334  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony piętnasty.

```
336  std::set<int> parallel_15(int min, int max) {
337      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
338      std::set<int> initial;
339
340      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
341      if (min == 2) initial.insert(initial.end(), 2);
342
343      std::set<int> to_erase;
344
345  #pragma omp parallel for default(none) private(primary_numbers, max, to_erase) shared(initial) schedule(dynamic, 100)
346      for (int x: primary_numbers) {
347          for (int y = x; y * x < max; y += 2)
348              to_erase.insert(x * y);
349  #pragma omp critical
350          for (int e: to_erase) initial.erase(e);
351      }
352
353      return initial;
354  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony szesnasty.

```
356  std::set<int> parallel_16(int min, int max) {
357      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
358      std::set<int> initial;
359
360      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
361      if (min == 2) initial.insert(initial.end(), 2);
362
363      std::set<int> to_erase;
364
365  #pragma omp parallel for default(none) private(primary_numbers, max, to_erase) shared(initial) schedule(guided)
366      for (int x: primary_numbers) {
367          for (int y = x; y * x < max; y += 2)
368              to_erase.insert(x * y);
369  #pragma omp critical
370          for (int e: to_erase) initial.erase(e);
371      }
372
373      return initial;
374  }
```

- [PODEJŚCIE FUNKCYJNE] Wariant zrównoleglony siedemnasty.

```
376  std::set<int> parallel_17(int min, int max) {
377      std::vector<int> primary_numbers = first_sequential_v(3, ceil(sqrt(max)));
378      std::set<int> initial;
379
380      for (int i = min % 2 ? min : min + 1; i < max; i += 2) initial.insert(initial.end(), i);
381      if (min == 2) initial.insert(initial.end(), 2);
382
383      std::set<int> to_erase;
384
385  #pragma omp parallel for default(none) private(primary_numbers, max, to_erase) shared(initial) schedule(guided, 10)
386      for (int x: primary_numbers) {
387          for (int y = x; y * x < max; y += 2)
388              to_erase.insert(x * y);
389  #pragma omp critical
390          for (int e: to_erase) initial.erase(e);
391      }
392
393      return initial;
394  }
```

Wariant	Czas dla liczby wątków = *1*	Czas dla liczby wątków = *3*	Czas dla liczby wątków = *6*	Czas dla liczby wątków = *12*
sequential	522.890290	---	---	---
parallel_1	45.245497	59.373896	64.112328	59.715072
parallel_2	64.308417	59.520607	64.881943	60.102123
parallel_3	64.400735	60.419365	64.950640	60.309975
parallel_4	64.719053	60.326552	64.666392	60.497431
parallel_5	64.960069	60.449499	64.785784	60.148380
parallel_6	64.116249	60.063787	64.331333	60.212495
parallel_7	64.360763	60.276481	63.696052	59.909399
parallel_8	64.558589	59.738300	64.438075	59.946487
parallel_9	64.123436	59.987361	64.521003	59.747855
parallel_10	63.489519	59.936017	64.423157	59.929436
parallel_11	64.457955	60.404314	64.479184	60.402309
parallel_12	64.487153	59.936047	64.895313	60.126156
parallel_13	64.345228	60.413464	64.542227	60.279202
parallel_14	64.161730	60.077114	64.453470	60.391956
parallel_15	63.936027	60.099122	63.898915	59.917218
parallel_16	64.305679	60.081819	64.915416	60.410302
parallel_17	64.711414	60.053957	65.039139	60.130069