روش اول:

- مقدار خانه آخر زیرآرایه اول باید با همه خانه‌های زیر آرایه دوم جمع شود چون که مقدار هر خانه مجموع اندیس‌ها تا آن‌جاست و در این صورت اندیس قسمت قبلی هنوز استفاده نشده و باید جمع شود تا عدد درستی حاصل شود.

```
21      prefix_sum(a, n);
22      double elapsedTime = omp_get_wtime() - startTime;
23      printf("Elapsed Time: %f \n", elapsedTime);
24      printf("Fini: %d", a[n - 1]);
25      free(a);
26      return EXIT_SUCCESS;
27  }
28
29  void prefix_sum(int* a, size_t n) {
30      int i;
31      int parallelCounter = 4;
32      int portion = n / parallelCounter;
33  #pragma omp parallel for num_threads(parallelCounter) schedule(static, 1)
34      for (int j = 0; j < n; j += portion)
35      {
36          // Optimal
37          /*if (j != 0) {
38              a[j + 1] += (j * (j + 1)) / 2;
39          }*/
40          for (i = 1; i < portion; ++i) {
41              a[i+j] +=  a[i + j - 1];
42          }
43          // Worst case
44          for (int k = j + portion; k < n; k += portion)
45          {
46              a[k+1] += a[i + j - 1];
47          }
48      }
49  // Sub-Optimal
50  //  for (int j = portion; j < n; j += portion)
51  //  {
52  //#pragma omp parallel for
53  //      for (i = 1; i < portion; ++i) {
54  //          a[i + j] += a[j - 1];
55  //      }
56  //  }
57  }
58
```

- دلیل اشتباهات احتمالی یکی این است که احتمال دارد نخی که برای جمع زدن قسمت i استفاده می‌شود هم‌زمان با نخ جمع‌زدن قسمت i-1 اجرا شود و به خاطر رخداد race condition اعداد نادرستی بگیریم. دلیل دیگر این است که عددهای این سری از یک جا به بعد از محدوده‌ی اعداد int در زبان C++ بیش‌تر می‌شوند و overflow رخ می‌دهد.

Elapsed Time (Serial): 1.966149
Elapsed Time (worst case): 7.070053
Elapsed Time (Sub-Optimal case): 1.354880
Elapsed Time (Optimal case): 0.785500

روش دوم:

```
    //size_t n = 10;
    size_t n = 1000 * 1000 * 10;
    int* a = new int[n];
    fill_array(a, n);
    double startTime = omp_get_wtime();
    prefix_sum(a, n);
    double elapsedTime = omp_get_wtime() - startTime;
    printf("E
    //printf(
    free(a);
    return EXIT_SUCCESS;
}

void prefix_sum(int* a, size_t n) {
    int lg = log2(n) + 1;
    int** y = new int* [lg];
    for (int i = 0; i < lg; i++) {
        y[i] = new int[n];
#pragma omp parallel for
        for (int j = 0; j < n; j++) {
            y[i][j] = a[j];
        }
    }
    for (int i = 0; i < lg-1; i++)
    {
#pragma omp parallel for
        for (int j = 0; j < n - 1; j++)
        {
            int p = pow(2, i);
            if (j < p)
                y[i + 1][j] = y[i][j];
            else
                y[i + 1][j] = y[i][j] + y[i][j - p];
        }
    }
}
```

(local variable) double elapsedTime

Search Online

به علت این که حافظه‌ی موردنیاز این الگوریتم با نقطه شروع ۱ گیگ از فضای رم دستگاه من خارج می‌شد، اجبارا با فضای ۱۰ مگابایتی انجام شد.

Elapsed Time (Hillis-Steele, 10MB): 0.796279
Elapsed Time (Serial, 10MB): 0.018149

کندتر است چون تعداد جمع‌هایی که نیاز دارد نسبت به الگوریتم‌های دیگر بسیار زیاد است در حالی که الگوریتم‌های دیگر با تعداد جمع کم‌تری همین محاسبات را انجام می‌دهند.

چون تعداد جمع‌های مستقل / کل جمع‌های این الگوریتم نسبت به الگوریتم‌های دیگر خیلی بیش‌تر است، می‌توان با اجرای این الگوریتم بر روی یک gpu موازی‌سازی بسیار بالایی را شاهد بود اما در حالت عادی و بر روی CPU خود تعداد جمع‌ها کم‌تر است پس روش‌های دیگر بهتر هستند. پس وقتی تعداد هسته‌های بسیار زیادی داشته باشیم می‌تواند از الگوریتم‌های دیگر مفیدتر باشد.

از کاربردهای prefix sum می‌توان به counting sort، list ranking (تبدیل لینک‌لیست به آرایه)، binary adders (در معماری کامپیوتر) و... اشاره کرد.