

به نام خدا



برنامه نویسی چندهسته‌ای

دستور کار آزمایشگاه ۴

هدف از این آزمایش، انجام عملیات Prefix sum بر روی یک آرایه است.

عملیات Prefix sum به صورت زیر تعریف می‌شود:

$$y[i] = \sum_{j=0}^i x[j]$$

به عبارت دیگر، هر درایه در آرایه خروجی، جمع همه درایه‌های قبل از خود در آرایه ورودی است.

این الگوریتم معمولاً به دو شیوه inclusive و exclusive پیاده می‌شود که تفاوت این دو را در ادامه می‌توانید ببینید:

Inclusive:

x	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰
y	۱	۳	۶	۱۰	۱۵	۲۱	۲۸	۳۶	۴۵	۵۵

Exclusive:

x	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰
y	۰	۱	۳	۶	۱۰	۱۵	۲۱	۲۸	۳۶	۴۵

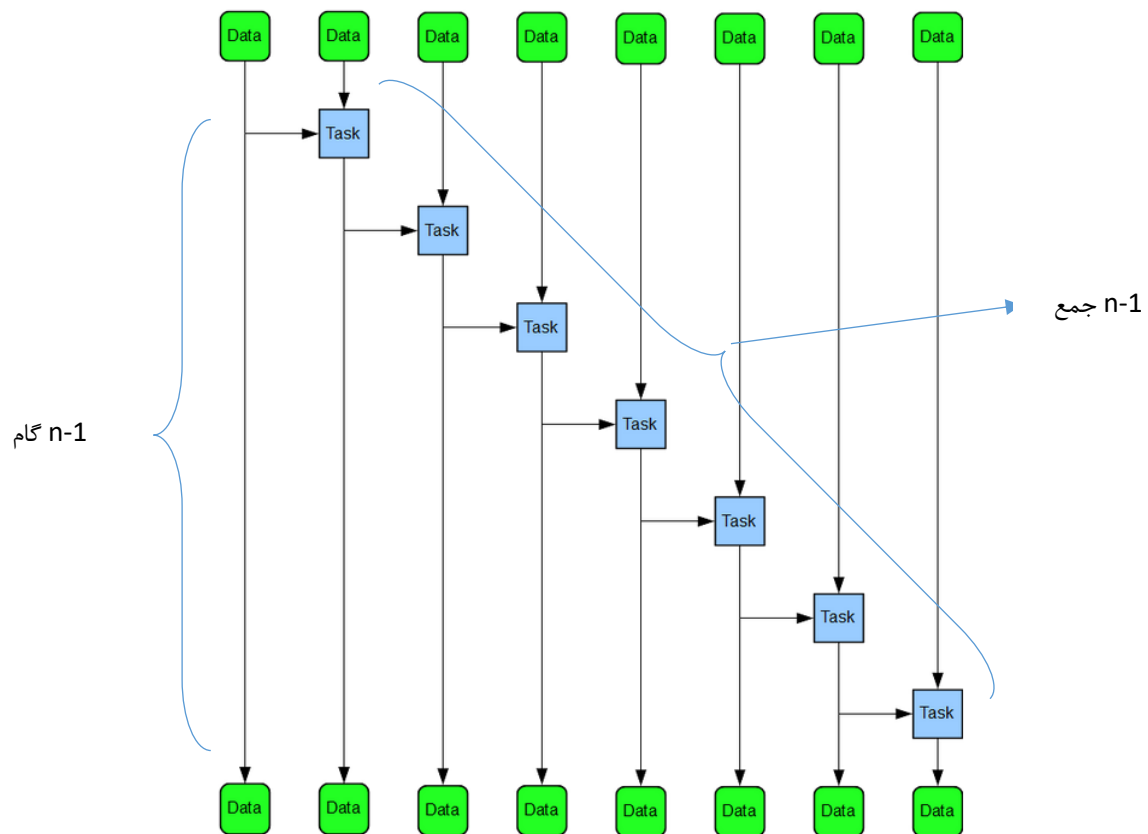
این عملیات کاربرد فراوانی در حوزه‌های مختلف دارد. به عنوان مثال:

اگر آرایه x همه تراکنش‌های مالی یک حساب (میزان کاهش یا افزایش حساب) است، آرایه y مقدار موجودی حساب تا پایان هر تراکنش خواهد بود.

و یا اگر آرایه x مقادیر تابع احتمالی PDF باشد، آرایه y مقادیر تابع احتمالی تجمعی CDF خواهد شد.

کد سریال این الگوریتم با نام `lab_4_serial.c` در اختیار شما قرار دارد.

نحوه انجام این عملیات به صورت شماتیک در شکل زیر مشخص است:



در این آزمایش، سه روش موازی سازی الگوریتم Prefix sum معرفی می شود.

توجه کنید در الگوریتم بالا $n-1$ عمل جمع و نیز $n-1$ گام برای محاسبه خروجی لازم است.

۱- روش اول:

ابتدا آرایه x را بین نخ‌ها به صورت static تقسیم کنید (هر نخ $1/n$ آرایه را پردازش می‌کند). هر نخ عملیات prefix sum را به صورت مستقل بر روی زیرآرایه خود انجام می‌دهد. به عنوان مثال با دو نخ داریم:

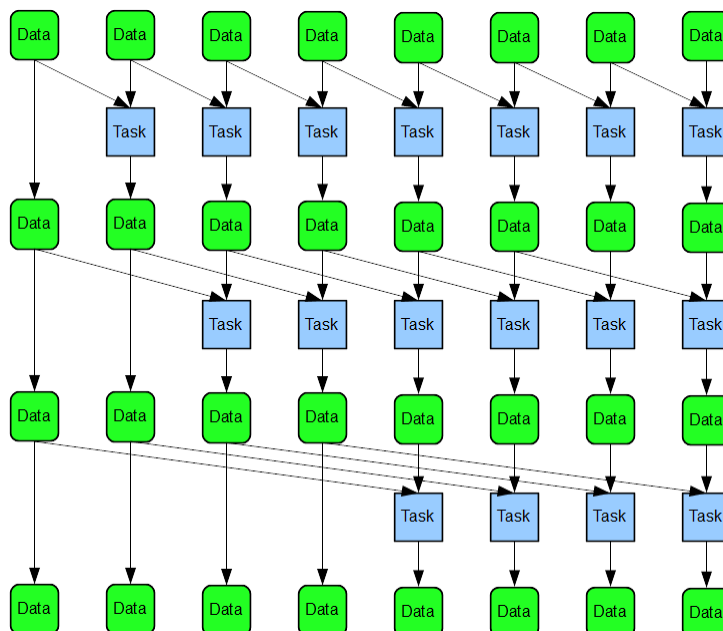
x:	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰
y:	۱	۳	۶	۱۰	۱۵	۶	۱۳	۲۱	۳۰	۴۰

نخ اول
نخ دوم

- پس از انجام کار هر نخ باید مقدار خانه آخر هر زیر آرایه را با تمامی خانه‌های زیرآرایه‌های بعد از آن جمع کنیم. مثلاً مقدار خانه آخر زیرآرایه اول (۱۵) باید با همه خانه‌های زیرآرایه دوم جمع شود. چرا؟
- اگر در مثال بالا تعداد نخ‌ها چهار شود برای به دست آوردن مقدار نهایی یک‌راه حل این است که هر نخ مقدار آخرین خانه محاسبه شده توسط خود را با تمام خانه‌های پس از آن جمع کند. این روش علاوه بر داشتن race، بیش از اندازه مورد نیاز عمل‌های جمع انجام می‌دهد. در این روش به ازای هر خانه از خروجی باید به تعداد بخش‌های قبل از آن روی خانه موردنظر عمل جمع صورت بگیرد. چگونه می‌توان به ازای هر خانه خروجی فقط با انجام یک عمل جمع این به‌روزرسانی را انجام داد و از روی آرایه بالا جواب نهایی را به دست آورد؟
- برای این الگوریتم تعداد جمع‌ها و نیز تعداد گام‌ها را به دست آورید.
- کد موازی این الگوریتم را بنویسید و زمان اجرای آن را به ازای ۱ گیگ اندازه حافظه با حالت سریال مقایسه کنید.

۲- روش دوم:

یک الگوریتم برای محاسبه prefix scan الگوریتمی است به نام Hillis and Steele که در سال ۱۹۸۶ معرفی شده است. شکل زیر الگوی محاسبات آن را نشان می‌دهد:



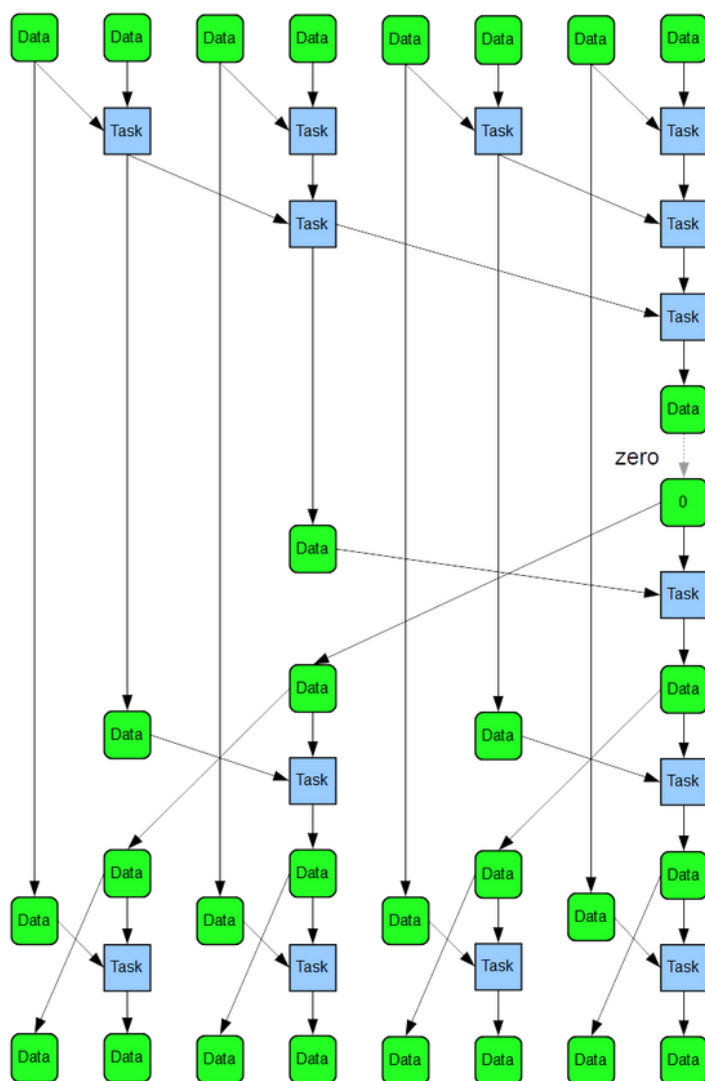
در این شکل، آرایه ورودی دارای ۸ المان است و آرایه خروجی در پایین محاسبه شده است. هر مربع task یک جمع است.

این الگوریتم **time-optimal** است؛ به این معنا که تعداد گام‌ها را کمینه کرده است بنابراین اگر به اندازه داده ورودی مسئله نخ داشته باشیم (تا تمام گام‌ها در یک واحد زمانی انجام شوند) آنگاه زمان اجرای برنامه در حالت موازی کمینه ($\log n$) خواهد شد؛ اما می‌دانیم در **cpu** این تعداد نخ موجود نیست و بنابراین کارهای مربوط به هر گام باید بین نخ‌ها تقسیم‌شده و بنابراین بیشتر از یک واحد زمانی طول خواهد کشید.

- با توجه به توضیحات بالا می‌توانید محاسبه کنید به ازای یک گیگ ورودی چه تعداد نخ لازم داریم تا روی کاغذ بتوان تسریع گرفت؟
- برای این الگوریتم تعداد جمع‌ها و نیز تعداد گام‌ها را به دست آورید.
- کد موازی این الگوریتم را بنویسید و زمان اجرای آن را به ازای ۱ گیگ اندازه حافظه با حالت سریال مقایسه کنید.

این الگوریتم توسط guy blelloch بر اساس balanced-tree در سال ۱۹۹۳ ارائه شد.

توجه کنید که این الگوریتم برخلاف دو روش قبل ماهیت exclusive دارد.



نحوه پیاده‌سازی این الگوریتم را در قطعه کد زیر می‌بینید:

```
void work_efficient_parallel_prefix_sum(int *a, int n) {
    for (int i = 1; i < n; i <= 1)
        for (int j = 0; j < n; j += 2 * i)
            a[2 * i + j - 1] = a[2 * i + j - 1] + a[i + j - 1];

    a[n - 1] = 0;

    for (int i = n / 2; i > 0; i >= 1) {
        for (int j = 0; j < n; j += 2 * i) {
            int temp = a[i + j - 1];
            a[i + j - 1] = a[2 * i + j - 1];
            a[2 * i + j - 1] = temp + a[2 * i + j - 1];
        }
    }
}
```

این الگوریتم **work-optimal** است؛ به این معنا که به دلیل محدودیت ما در داشتن هر تعداد نخ، تعداد عمل‌های جمع برای اجرای این الگوریتم به صورت موازی کمینه شده است و برای رسیدن به این هدف تعداد گام‌ها دو برابر شده است.

- قطعه کد بالا را موازی کرده و زمان اجرای این الگوریتم را با روش‌های قبل مقایسه کنید.
- توضیح دهید این الگوریتم که درواقع بهینه‌شده الگوریتم روش دوم است در چه حالتی می‌تواند نسبت به الگوریتم اول مزیت داشته باشد (راهنمایی: این الگوریتم در GPU بسیار پرکاربرد است).
- این الگوریتم تنها برای توان‌های دو عملکرد صحیحی دارد. چگونه می‌توان برای هر عددی از این الگوریتم استفاده نمود؟
- چگونه می‌توان این الگوریتم را به صورت **inclusive** نیز پیاده‌سازی کرد؟ (نیازی به پیاده‌سازی نیست)