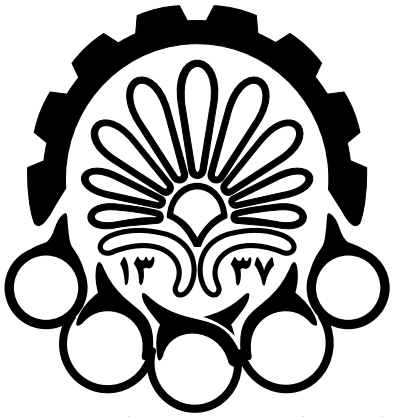



**دانشگاه صنعتی امیرکبیر**
( پلی تکنیک تهران )
**دانشکده مهندسی کامپیوتر**

# برنامه‌نویسی چندهسته‌ای

## تمرین سوم: آشنایی با برنامه‌نویسی OpenMP

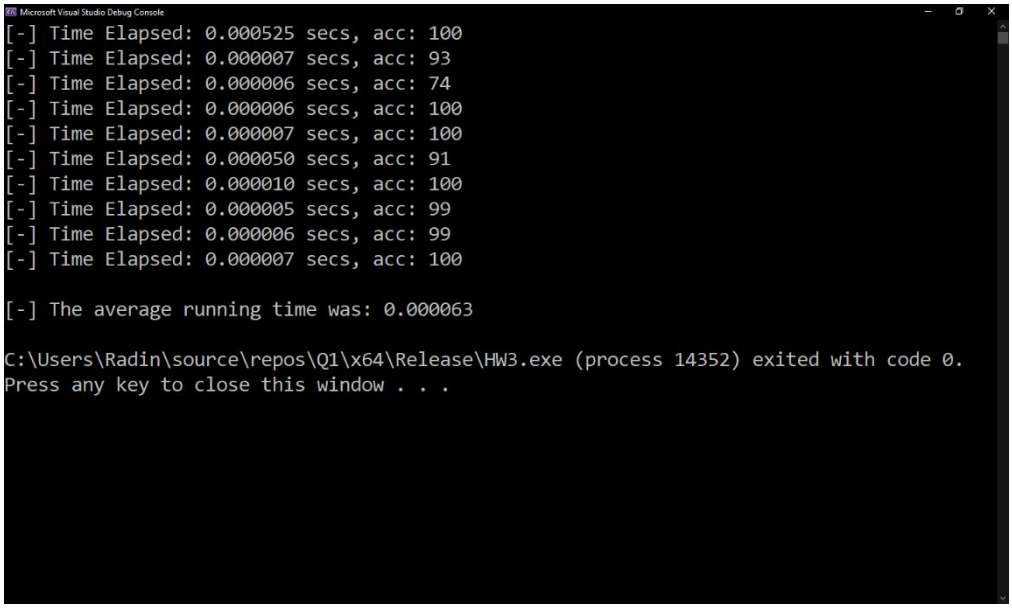
**رادین شایانفر**

**شماره دانشجویی: ۹۷۳۱۰۳۲**

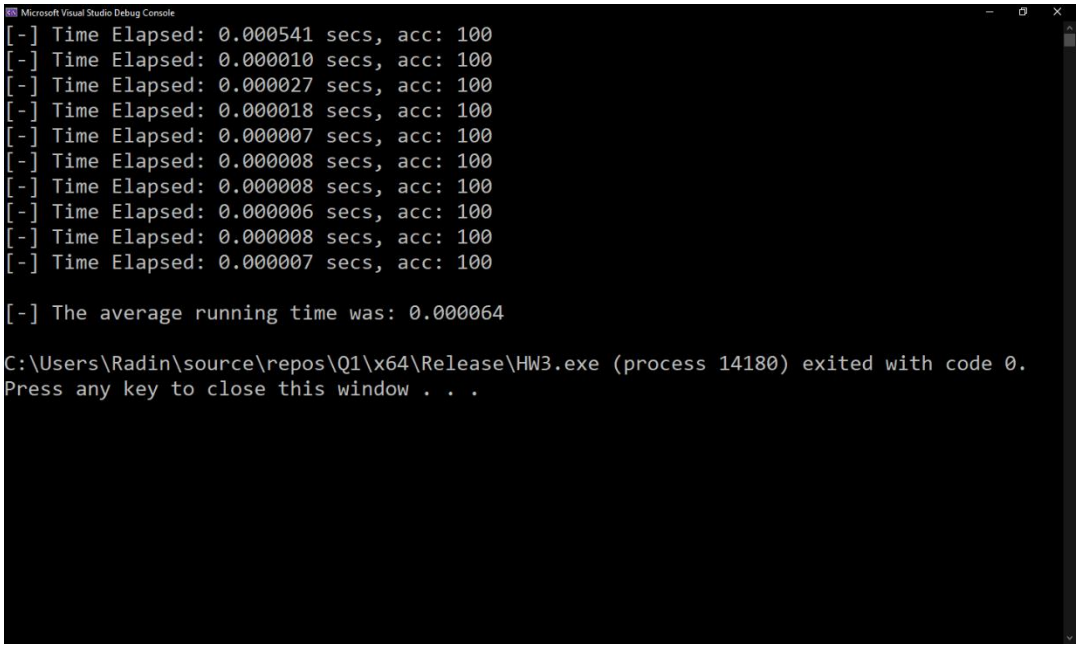
**بهار ۱۴۰۰**

**۱.** کدهای اصلاح شده در فایل‌های `Q1_1.cpp`، `Q1_2.cpp` و `Q1_3.cpp` قرار دارند.

۱) با تغییر کد داده شده، امکان چند بار اجرا و اندازه‌گیری و میانگین گرفتن زمان را به آن اضافه می‌کنیم.




مشاهده می‌کنیم که به علت وجود شرایط مسابقه برای متغیر `acc`، مقدار آن در برخی از اجراها برابر ۱۰۰ نیست. با استفاده از `reduction(+: acc)` این متغیر را در نخ‌ها خصوصی (private) می‌کنیم و در هنگام خروج از ناحیه بحرانی این مقادیر خصوصی را جمع می‌کنیم. در شکل زیر می‌بینیم که این مشکل حل شده و مقدار `acc` در اجراهای مختلف برابر ۱۰۰ است.



از آنجا که تعداد اجراهای حلقه کوچک است، در این ابعاد موازی‌سازی آن به صرفه نیست و سربار موازی‌سازی آن باعث کندتر شدن برنامه نسبت به حالت سریال می‌شود. اما مطمئنا با افزایش اندازه مسئله، موازی‌سازی باعث تسریع این مسئله خواهد شد.

۲) در این مسئله، دسترسی‌های به حافظه به خانه‌های مجاور هم نیست. در نتیجه حافظه کش نمی‌تواند به خوبی زمان دسترسی به حافظه را کاهش دهد. در حالی که با اندکی تغییر در کد برنامه می‌توان به همان نتایج، اما با سرعت بالاتر دست یافت. زمان اجرا در حالت ابتدایی به طور میانگین ۰.۰۰۰۱۴۵ ثانیه است. با جابجایی نام `i` و `j` در داخل حلقه‌ها، زمان اجرا به ۰.۰۰۰۰۷۰ می‌رسد که بیش از ۲ برابر سریع‌تر است.

۳) در اینجا با چاپ کردن بازه‌های `start` و `end` برای هر نخ، می‌بینیم که اگر اندازه آرایه به تعداد نخ‌ها بخش‌پذیر نباشد، چند خانه آخر آرایه (مثلا در اینجا خانه ۹۲۸) پردازش نمی‌شود. باید توجه داشت که هر نخ بازه [start, end) خود را پردازش می‌کند.

```
thread: 0, start: 0, end: 232
thread: 2, start: 464, end: 696
thread: 1, start: 232, end: 464
thread: 3, start: 696, end: 928
```

با محاسبه `workload_size` به شکل

```
int workload_size = ceil((double)arr_size / omp_get_num_threads());
```

و افزودن شرط `i < arr_size` در حلقه، همه خانه‌های آرایه مطابق شکل زیر پردازش می‌شوند.

```
thread: 0, start: 0, end: 233
thread: 2, start: 466, end: 699
thread: 1, start: 233, end: 466
thread: 3, start: 699, end: 932
```

**۲.** کد برنامه در فایل `3DMatMul.cpp` آمده است. به توضیح برنامه می‌پردازیم.

ابتدا کد سریال ضرب ماتریس را می‌نویسیم. در این کد اندازه یک ضلع از ماتریس‌ها توسط آرگومان خط فرمان داده می‌شود. سپس تابع `fillDataset` به ماتریس‌ها حافظه تخصیص داده و مقدار

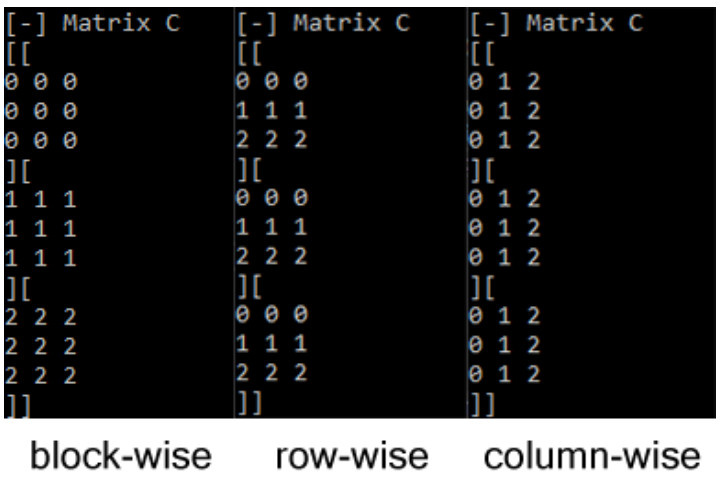
ماتریس‌های A و B را با اعداد تصادفی بین ۰ تا ۹۹ پر می‌کند. همچنین برای صرفه‌جویی در حافظه خانه‌های ماتریس به جای int، به شکل unsigned short تعریف شده‌اند.

پس از کد سریال، سه نسخه از عمل ضرب را در توابع `mulBlock`، `mulRow` و `mulCol` می‌نویسیم. یک نمونه از اجرای موازی کد را در شکل زیر می‌بینیم که نشان‌دهنده درستی محاسبات و تجزیه است.


همچنین برای بهتر دیده شدن تقسیم وظایف، به جای نوشتن حاصل جمع در خانه‌های ماتریس خروجی، توسط خط زیر شماره نخی که آن خانه را پردازش می‌کند را می‌نویسیم.

```
dataset.C[i][j][k] = omp_get_thread_num();
```

تقسیم وظایف را برای ۳ حالت تجزیه، با ۳ نخ و ماتریس‌های ۳×۳×۳ در شکل زیر می‌بینیم:



**توجه شود که پس از دیدن نحوه تخصیص وظایف، مجدد کد به حالت ضرب برگردانده شده است تا زمان اجرا به درستی و برای برنامه خواسته شده اندازه‌گیری شود.**

حال به کمک توابع نوشته شده، زمان‌های اجرای **عمل ضرب** را در جدول‌های ۱، ۲ و ۳ می‌نویسم. پردازنده استفاده شده Intel Core i7 7700 با ۴ هسته و ۸ نخ سخت‌افزاری است. به همین دلیل استفاده از ۱۶ نخ در نتایج زیر مزیتی نسبت به حالت ۸ نخی ندارد.

جدول ۱: نتایج تجزیه block-wise

| تعداد نخ‌ها | زمان اجرا (ثانیه) | | | | تسریع |
|---|---|---|---|---|---|
| | 128×128×128 | 256×256×256 | 512×512×512 | 1024×1024×1024 | |
| ۱ | 0.564591 | 9.314080 | 164.962359 | 2896.394506 | - |
| ۴ | 0.223714 | 3.536650 | 54.576253 | 981.965256 | 2.78 |
| ۸ | 0.145528 | 2.752472 | 44.518635 | 1061.647716 | 3.42 |
| ۱۶ | 0.156492 | 2.560828 | 45.197500 | - | 3.63 |

جدول ۲: نتایج تجزیه row-wise

| تعداد نخ‌ها | زمان اجرا (ثانیه) | | | | تسریع |
|---|---|---|---|---|---|
| | 128×128×128 | 256×256×256 | 512×512×512 | 1024×1024×1024 | |
| ۱ | 0.591247 | 9.893365 | 181.658930 | - | - |
| ۴ | 0.253416 | 4.748615 | 62.964235 | - | 2.43 |
| ۸ | 0.220390 | 4.890985 | 53.499103 | - | 2.70 |
| ۱۶ | 0.208197 | 3.939697 | 58.782256 | - | 2.81 |

جدول ۳: نتایج تجزیه column-wise

| تعداد نخ‌ها | زمان اجرا (ثانیه) | | | | تسریع |
|---|---|---|---|---|---|
| | 128×128×128 | 256×256×256 | 512×512×512 | 1024×1024×1024 | |
| ۱ | 0.626544 | 10.415803 | 184.722504 | - | - |
| ۴ | 0.352073 | 5.403500 | 72.588766 | - | 2.08 |
| ۸ | 0.448551 | 5.858374 | 64.042713 | - | 2.01 |
| ۱۶ | 0.484100 | 5.981062 | 76.701640 | - | 1.81 |

ستون آخر در برخی از جدول‌ها به دلیل زمان اجرای طولانی اندازه‌گیری نشده و تسریع تنها با استفاده از نتایج ۳ ستون اول محاسبه شده است.

همانطور که می‌بینیم تقریبا تجزیه block-wise بهترین عملکرد و تجزیه column-wise بدترین عملکرد را دارد.