"بسمه تعالی"

گزارش آزمایش اول برنامه نویسی چندهسته‌ای – زهرا لطیفی – ۹۹۲۳۰۶۹

مرحله اول – پاسخ به سوالات

۱- بنظر می‌رسد بخشی از برنامه که بیشترین زمان را می‌گیرد، حلقه for داخلی باشد. یعنی این بخش از کد:

```
for (j = 0; j<VERYBIG; j++)
	{
		// increment check sum
		sum += 1;
		// Calculate first arithmetic series
		sumx = 0.0;
		for (k = 0; k<j; k++)
			sumx = sumx + (double)k;
		// Calculate second arithmetic series
		sumy = 0.0;
		for (k = j; k>0; k--)
			sumy = sumy + (double)k;
		if (sumx > 0.0)total = total + 1.0 / sqrt(sumx);
		if (sumy > 0.0)total = total + 1.0 / sqrt(sumy);
	}
```

این بخش از کد به دلیل حلقه‌های تو در تو پیچیدگی زمانی از اردر O(n^2) دارد که n برابر 5000 است. حلقه‌های درونی در حال محاسبه دو سری حسابی هستند که سریالند زیرا واضحا محاسبه هر مرحله به مرحله قبلی بستگی دارد.

اما حلقه بیرونی وابستگی بین iterationها ندارد، بنابراین می‌توان آن را موازی کرد. هر نخ می‌تواند به طور مستقل sumx، sumy و total را برای زیرمجموعه‌ای از تکرارهای VERYBIG محاسبه کند. این مسئله اجازه می‌دهد تا کار بین چندین هسته توزیع شود و زمان اجرا کاهش پیدا کند.

۲- احتمالا فلسفه ۱۰ بار تکرار محاسبات و چاپ نتایج در هر مرحله مربوط به ملاحظات benchmarking است. با اجرای چندین بار محاسبات، برنامه می‌تواند اندازه‌گیری دقیق‌تری از زمان لازم برای انجام عملیات بدست آورد. که می‌تواند به حذف نتایج پرت که ممکن است در یک اجرا به دلیل عواملی مانند زمان بندی CPU، سایر فرآیندهای در حال اجرا در سیستم و غیره رخ دهد، کمک کند.

در مورد بخشی از نتایج که بین تکرارها متفاوت است، این متغیر elapsedtime است. این متغیر مدت زمان تکمیل "Work Loop" را برای هر تکرار حلقه بیرونی اندازه گیری می‌کند. از آنجایی که زمان اجرا به دلیل عوامل بیان شده (زمان‌بندی CPU، سایر فرآیندهای درحال اجرا و غیره) می‌تواند کمی متفاوت باشد، احتمالاً زمان سپری شده برای هر

تکرار کمی متفاوت خواهد بود. متغیرهای sum و total نیز برای هر تکرار متفاوت خواهند بود، اما مقادیر آنها در شروع هر تکرار حلقه بیرونی ۰ می‌شود. بنابراین، با فرض قطعی بودن محاسبات، مقادیر نهایی آنها که در پایان هر تکرار چاپ می‌شوند باید یکسان باشند.

۳- Debug: این حالت برای Debug کردن برنامه استفاده می‌شود. کامپایلر بهینه‌سازی‌های زیادی را اعمال نمی‌کند. این امر ردیابی اشکالات را آسان تر می‌کند. در حالت Debug، کد با اطلاعات کامل و بدون بهینه سازی کامپایل می‌شود. این کار ارتباط کد باینری را با کد اصلی برای دیباگرها ساده‌تر می‌کند و اگر خرابی رخ دهد، دیباگر می تواند دقیقاً خط کدی که باعث آن شده است را نشان دهد.

Release: این حالت برای اجرای نهایی برنامه استفاده می‌شود. کامپایلر بهینه‌سازی‌هایی را برای اجرای سریع‌تر کد اعمال می‌کند. با این حال، این بهینه سازی‌ها می تواند debug کردن را دشوارتر کند. در حالت Release، کد برای سرعت یا اندازه بهینه شده است. این باعث می‌شود برنامه کوچکتر، سریعتر و کارآمدتر شود، اما اگر خرابی رخ دهد، دیباگر نمی تواند اطلاعات زیادی را ارائه دهد.

پس کد کامپایل شده در حالت Release به دلیل بهینه سازی‌های اعمال شده توسط کامپایلر سریعتر است. این بهینه‌سازی‌ها می‌تواند شامل مواردی مانند حذف کدهای غیرضروری، جایگزینی عملیات آهسته با عملیات سریع‌تر، مرتب کردن مجدد دستورالعمل‌ها برای استفاده بهتر از pipeline و ... باشد.

```
C:\Windows\system32\cmd.exe
Serial Timings for 50000 iterations

Time Elapsed: 2.983408 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 2.980769 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 2.964813 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 3.038734 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 2.949153 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 2.920866 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 2.921133 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 2.947474 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 2.945794 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 2.941986 Secs, Total = 30.656747, Check Sum = 50000
```

۱- اجرای کد سریال در حالت Release

```
C:\Windows\system32\cmd.exe
Serial Timings for 50000 iterations

Time Elapsed: 6.743785 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 6.736477 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 6.850007 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 6.548020 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 6.649358 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 6.767253 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 13.301253 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 16.506424 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 6.639780 Secs, Total = 30.656747, Check Sum = 50000
Time Elapsed: 6.579015 Secs, Total = 30.656747, Check Sum = 50000
```

۲- اجرای کد سریال در حالت Debug

۴- برای موازی کردن این کد، می‌توان از مدل موازی‌سازی داده استفاده کرد که شکلی از موازی سازی است که در آن عملیات یکسانی برروی داده‌های مختلف اعمال می‌شود. در این حالت، هر تکرار از حلقه بیرونی مستقل است و می‌تواند به صورت موازی اجرا شود. درواقع محاسبات یکسان در هر تکرار انجام می‌شود، اما بر روی داده‌های متفاوت.

می توان هم از ساختارهای work-sharing برای توزیع تکرارهای حلقه بیرونی در چند نخ استفاده کرد. همانطور که در دستورکار هم آمده، دستورالعمل «#pragma omp parallel for» می‌تواند برای تقسیم خودکار تکرارهای حلقه بین نخ‌های موجود استفاده شود.

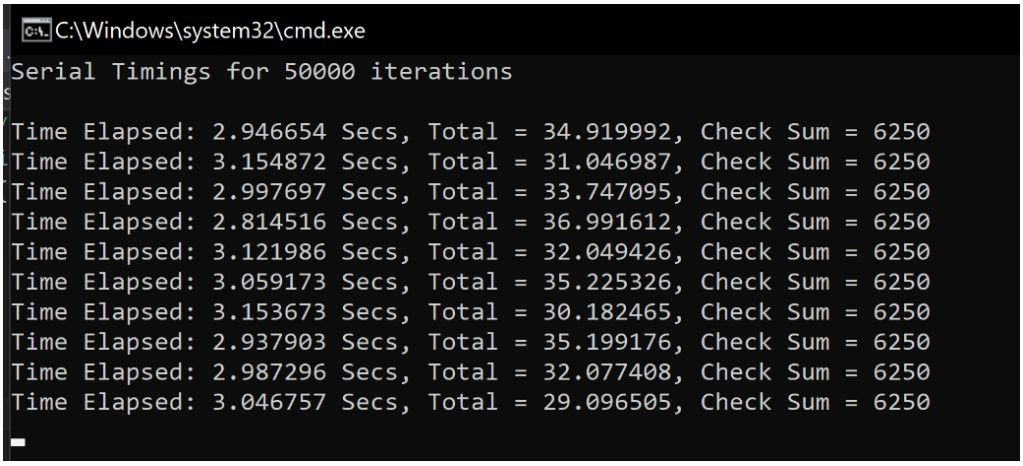
از Reduction هم می‌توان استفاده کرد. متغیرهای "sum" و "total" در هر تکرار حلقه افزایش می‌یابند که می‌تواند منجر به Race شود. می‌توان با استفاده از عبارت Reduction، یک عملیات کاهش (مانند جمع) را روی یک متغیر در یک ساختار موازی انجام داد و نتایج همه نخ‌ها را در پایان ترکیب کرد.

حلقه‌های داخلی که «sumx» و «sumy» را محاسبه می‌کنند نیز می‌توانند با استفاده از Nested Parallelism موازی شوند، اگر مقدار «VERYBIG» به اندازه‌ای بزرگ باشد که این کار مفید باشد زیرا پیچیدگی کد را زیاد می‌کند.

شرح نتایج مرحله دوم:

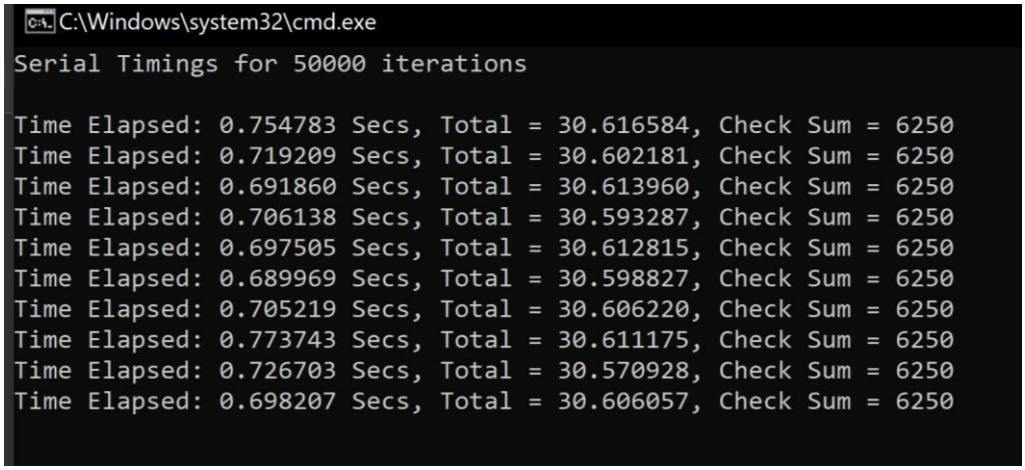
بخش ۱.

بخش ۳.

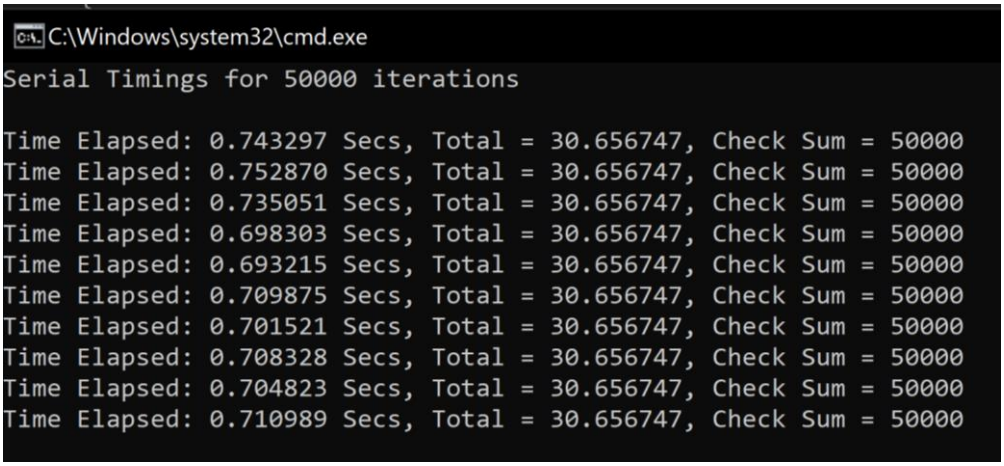


زیرا این برنامه شرایط Race دارد و مشکلات آن باید حل شود. در وضعیت فعلی، متغیرهایsumx ، sumy، totalو همچنین اندیس k به صورت shared است. تنها اندیس j که اندیس حلقه موازی شده است به صورت اتوماتیک اختصاصی (private) می‌شود.

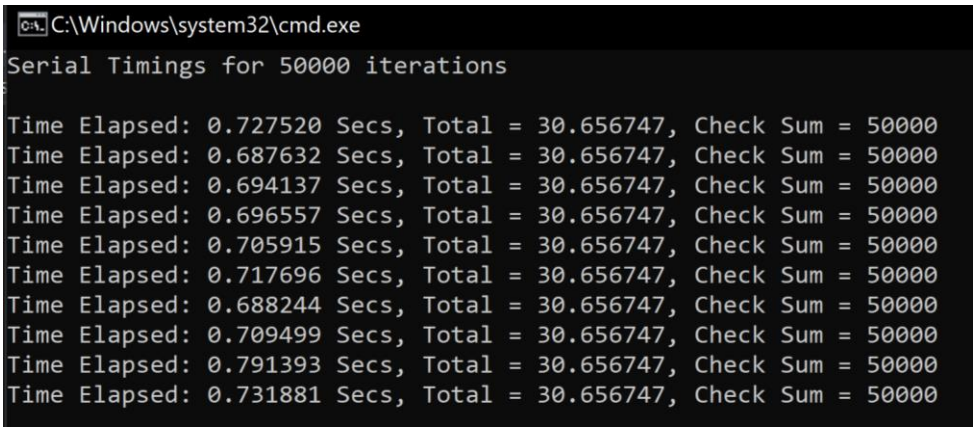
بخش ۵.



بخش ۷.



بخش ۸ و ۹.



بخش ۱۰.

می‌توان با اضافه کردن

```
#pragma omp parallel for private(k)
```

به قبل از حلقه for مربوط به j، این کار را انجام داد به نحوی که هر نخ مقادیر sumx و sumy خود را محاسبه می‌کند که سپس با استفاده از یک عملیات اتمی به متغیر total مشترک اضافه می‌شود. این امر از نیاز به نواحی بحرانی یا reduction جلوگیری می‌کند.

```
#include <stdio.h>
#include <math.h>
#include <omp.h>
const long int VERYBIG = 50000;
int main(void)
{
	#ifndef _OPENMP
		printf("OpenMP is not supported, sorry!\n");
		getchar();
		return 0;
	#endif
	int i;
	long int j, k;
	double starttime, elapsedtime;
	// Output a start message
	printf("Serial Timings for %d iterations\n\n", VERYBIG);
	// repeat experiment several times
	for (i = 0; i<10; i++)
	{
		// get starting time
		starttime = omp_get_wtime();
		// reset running total
		double total = 0.0;
		// Work Loop, do some work by looping VERYBIG times
		#pragma omp parallel for private(k)
		for (j = 0; j<VERYBIG; j++)
		{
			// Calculate first arithmetic series
			double sumx = 0.0;
			for (k = 0; k<j; k++)
				sumx = sumx + (double)k;
			// Calculate second arithmetic series
			double sumy = 0.0;
```

```
                  for (k = j; k>0; k--)
                        sumy = sumy + (double)k;
                  // Update total
                  if (sumx > 0.0) {
                        #pragma omp atomic
                        total += 1.0 / sqrt(sumx);
                  }
                  if (sumy > 0.0) {
                        #pragma omp atomic
                        total += 1.0 / sqrt(sumy);
                  }
            }
            // get ending time and use it to determine elapsed time
            elapsedtime = omp_get_wtime() - starttime;
            // report elapsed time
            printf("Time Elapsed: %f Secs, Total = %lf\n",
                  elapsedtime, total);
      }
      // return integer as required by function header
      getchar();
      return 0;
}
```

مرحله دوم – پاسخ به سوالات

۱- اگر به صراحت تعداد نخ‌ها را در یک ناحیه موازی با عبارت 'num_threads' مشخص نکنیم، تعداد نخ‌هایی که ایجاد می‌شوند توسط متغیر 'OMP_NUM_THREADS' تعیین می‌شود. اگر OMP_NUM_THREADS تنظیم نشده باشد، OpenMP به طور پیش‌فرض به تعداد هسته‌های موجود در سیستم، نخ ایجاد می‌کند. این به این دلیل است که به طور کلی، داشتن Thread های بیشتر از هسته‌ها باعث بهبود عملکرد نمی‌شود و در واقع می‌تواند عملکرد را به دلیل ایجاد overhead و مدیریت نخ‌های بیشتر کاهش دهد.

۲- بله، امکان پذیر است. اما تنها برای عملیات خاصی. دستورالعمل «atomic» در OpenMP یا به طور کامل انجام می‌شود یا اصلاً انجام نمی‌شود و از Race جلوگیری می‌کند اما عملیات «atomic» محدود به عملیات خواندن، تغییر و نوشتن ساده بر روی متغیرهای اسکالر، مانند افزایش (`++`) یا کاهش (`--`) یک متغیر، یا تخصیص ساده است. مثلا تعییر زیر ممکن است:

```
#pragma omp atomic
sum += 1;
```

اما دستورالعمل «critical» می‌تواند برای محافظت از هر بلوکی از کد استفاده شود، اما از نظر کارایی ممکن است پیچیده‌تر باشد زیرا تنها یک نخ می‌تواند بخش critical را در یک زمان اجرا کند. در کد ما به‌روزرسانی‌های «total» شامل یک عبارت پیچیده‌تر است و نمی‌توان آن را اتمی کرد. آنها باید توسط یک ناحیه «critical» یا با استفاده از بند «reduction» موازی شوند.

۳- بله متفاوت است. عبارت «reduction» عموماً مفیدتر از ناحیه «critical» است. این به این دلیل است که "reduction" به هر نخ اجازه می‌دهد تا محاسبات خود را به طور مستقل با استفاده از یک کپی از متغیر انجام دهد و سپس نتایج را در پایان ترکیب کند. این امر از نیاز به همگام سازی در طول اجرای حلقه جلوگیری می‌کند. از سوی دیگر، یک ناحیه «critical» نخ‌ها را مجبور می‌کند تا منتظر نوبت خود بمانند، که می‌تواند سرعت برنامه را کاهش دهد، به خصوص اگر بخش بحرانی بخش مهمی از کد باشد.

پس استفاده از reduction به جای ناحیه بحرانی منجر به زمان اجرای سریع‌تر می‌شود، به‌ویژه برای مقادیر بسیار بزرگ‌تر VERYBIG و زمانی که تعداد نخ‌ها به تعداد هسته‌های سیستم نزدیک باشد.

کد نهایی تا قبل از مرحله ۱۰

```
#include <stdio.h>
#include <math.h>
#include <omp.h>
const long int VERYBIG = 50000;
// ************************************************************************
int main(void)
{
	#ifndef _OPENMP
		printf("OpenMP is not supported, sorry!\n");
		getchar();
		return 0;
	#endif
	int i;
	long int j, k, sum;
	double sumx, sumy, total;
	double starttime, elapsedtime;
	// ------------------------------------------------------------------------
	// Output a start message
	printf("Serial Timings for %d iterations\n\n", VERYBIG);
	// repeat experiment several times
	for (i = 0; i<10; i++)
	{
		// get starting time56 x CHAPTER 3 PARALLEL STUDIO XE FOR THE IMPATIENT
		starttime = omp_get_wtime();
		// reset check sum & running total
		sum = 0;
		total = 0.0;
		// Work Loop, do some work by looping VERYBIG times
		//#pragma omp parallel for
		//#pragma omp parallel for private(k, sumx, sumy)
		#pragma omp parallel for private(k, sumx, sumy) reduction(+:sum, total)
		for (j = 0; j<VERYBIG; j++)
		{
			// increment check sum
			#pragma omp critical
			//#pragma omp atomic
			sum += 1;
			// Calculate first arithmetic series
			sumx = 0.0;
			for (k = 0; k<j; k++)
				sumx = sumx + (double)k;
			// Calculate second arithmetic series
			sumy = 0.0;
			for (k = j; k>0; k--)
				sumy = sumy + (double)k;
			#pragma omp critical
			if (sumx > 0.0)total = total + 1.0 / sqrt(sumx);
			#pragma omp critical
			if (sumy > 0.0)total = total + 1.0 / sqrt(sumy);
		}
		// get ending time and use it to determine elapsed time
		elapsedtime = omp_get_wtime() - starttime;
		// report elapsed time
		printf("Time Elapsed: %f Secs, Total = %lf, Check Sum = %ld\n",
			elapsedtime, total, sum);
	}
	// return integer as required by function header
	getchar();
	return 0;
}
```