به نام خدا

تهیه کننده:ابراهیم صدیقی شماره دانشجویی: ۹۹۳۱۰۹۸

سوال ۱)

Explicit Multithreading:

Explicit Multithreading شامل ایجاد و مدیریت دستی رشته ها توسط برنامه نویس است. در این رویکرد، توسعه‌دهنده به صراحت تعریف می‌کند که در کجا و چگونه رشته‌ها در کد ایجاد، همگام‌سازی و خاتمه می‌شوند.

مضایا:
- کنترل ریز دانه: توسعه دهندگان کنترل دقیقی بر ایجاد نخ، هماهنگ سازی و خاتمه دارند.
- عملکرد بهینه: امکان بهینه سازی متناسب بر اساس نیازهای برنامه خاص را فراهم می کند.
- اشکال زدایی: ردیابی و اشکال زدایی مسائل مربوط به رشته گذاری آسان تر است.

معایب:
- پیچیدگی: به دانش عمیق مفاهیم چند رشته ای و مکانیسم های همگام سازی نیاز دارد.
- مستعد خطا: احتمال باگ های مربوط به مدیریت رشته را افزایش می دهد.
- تعمیر و نگهداری: ممکن است به دلیل منطق پیچیده threading، نگهداری و گسترش کد دشوارتر شود.

Implicit Multithreading:

Implicit Multithreading شامل استفاده از کتابخانه ها یا چارچوب هایی است که به طور خودکار رشته ها را بدون دخالت صریح برنامه نویس مدیریت می کنند. سیستم یا کتابخانه ایجاد، زمان‌بندی و همگام‌سازی موضوعات را انجام می‌دهد.

مضایا:
- Simplicity: پیچیدگی مدیریت موضوعات را به صورت دستی کاهش می دهد.
- بهره‌وری: توسعه‌دهندگان می‌توانند بدون نگرانی در مورد جزئیات رشته‌های سطح پایین، روی منطق اصلی برنامه تمرکز کنند.
- مقیاس‌پذیری: فرآیند مقیاس‌پذیری برنامه‌ها را برای استفاده موثر از چندین هسته آسان می‌کند.

معایب:
- کنترل محدود: کاهش کنترل بر روی ایجاد و همگام سازی نخ.
- سربار عملکرد: لایه انتزاعی ممکن است سربار را معرفی کند که بر عملکرد تأثیر می گذارد.
- وابستگی: تکیه بر کتابخانه ها یا چارچوب های خاص برای عملکرد چند رشته ای.

Automatic Multi-threading:

Automatic Multi-threading به مکانیزم های کامپایل یا زمان اجرا اشاره دارد که به طور خودکار کد را موازی می کند تا از چندین هسته بدون دخالت صریح برنامه نویس استفاده کند. این سیستم وظایف قابل موازی سازی را شناسایی کرده و آنها را در هسته های موجود توزیع می کند.

مضایا:

- بهره وری: به طور خودکار کد را برای اجرای موازی بهینه می کند و استفاده از هسته را به حداکثر می رساند.
- سادگی: توسعه دهندگان برای فعال کردن موازی سازی نیازی به تغییر کد موجود ندارند.
- عملکرد: می تواند عملکرد را با استفاده از معماری های چند هسته ای به طور قابل توجهی بهبود بخشد.

معایب:

- پیچیدگی: الگوریتم‌های موازی‌سازی خودکار می‌توانند پیچیده باشند و ممکن است همیشه نتایج بهینه ایجاد نکنند.
- چالش های رفع اشکال: شناسایی و حل مسائل مربوط به موازی سازی خودکار می تواند چالش برانگیز باشد.
- سازگاری: همه کدها ممکن است برای موازی سازی خودکار مناسب نباشند، که منجر به محدودیت هایی در کاربرد آن می شود.

سوال ۲)

مشکل اصلی کد مربوط به مدیریت نادرست متغیر num_threads در ناحیه موازی است که می‌تواند منجر به شرایط مسابقه شود. این به این دلیل است که چندین رشته سعی می کنند به طور همزمان در num_threads بدون همگام سازی بنویسند. علاوه بر این، برای بهبود عملکرد و اطمینان از صحت، بهتر است مجموع کل در ناحیه موازی محاسبه شود تا سربار به حداقل برسد و از اشتراک گذاری نادرست احتمالی جلوگیری شود.

در اینجا یک نسخه بهبود یافته از کد در توضیحات آمده است:

```cpp
#include <iostream>
#include <cmath>
#include <omp.h>
#define NUM_THREADS 8
#define NUM_STEPS 100000

int main()
{
    static double step_len = 10.0 / (double)NUM_STEPS;
    double result = 0.0;

    // Use a single variable for the sum and make it private to each thread to avoid false sharing
    // Reduction clause is used to safely accumulate the results from each thread
#pragma omp parallel num_threads(NUM_THREADS) reduction(+:result)
    {
        double x;
        double sum = 0.0; // Local variable for each thread to accumulate its partial sum
        int thread_id = omp_get_thread_num();
        int num_threads = omp_get_num_threads(); // Correctly moved inside the parallel region

        for (int i = thread_id; i < NUM_STEPS; i += num_threads)
        {
            x = (i + 0.5) * step_len - 5.0;
            sum += exp(-(x * x));
        }
        // Accumulate the partial sum into the result variable using the reduction clause
        result += sum * step_len;
    }

    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

- اصلاح شرایط مسابقه: متغیر num_threads اکنون در داخل منطقه موازی اعلام می‌شود و اطمینان حاصل می‌کند که هر رشته مقدار صحیح را بدون ایجاد شرایط مسابقه دریافت می‌کند.
- Reduction Clause for Summation: از بند کاهش (reduction(+:result)) برای جمع آوری ایمن مبالغ جزئی از هر رشته در متغیر نتیجه استفاده می شود. این رویکرد هم از نظر رزرو ایمن و هم کارآمد است، زیرا سربار مرتبط با همگام سازی نخ را به حداقل می رساند.
- اجتناب از اشتراک گذاری غلط: با استفاده از یک متغیر مجموع محلی در هر رشته و سپس ترکیب آنها با استفاده از عبارت کاهش، کد از اشتراک گذاری نادرست جلوگیری می کند. اشتراک‌گذاری کاذب می‌تواند زمانی رخ دهد که رشته‌هایی روی هسته‌های مختلف تلاش می‌کنند در مکان‌های مجاور در یک خط کش یکسان بنویسند، که منجر به انتقال داده‌های غیرضروری بین هسته‌ها و ترافیک انسجام حافظه پنهان می‌شود.
- عملکرد: این کد برای اجرای موازی بهینه شده است، اطمینان حاصل می کند که کار به طور مساوی بین رشته های موجود توزیع می شود و هزینه های سربار مربوط به همگام سازی رشته ها و اشتراک گذاری داده ها را به حداقل می رساند.

سوال ۴)

a)موازی سازی با OpenMP

برای موازی کردن تابع conv_pool با استفاده از OpenMP، باید دستورالعمل‌های OpenMP را به حلقه‌هایی اضافه کنیم که می‌توانند به صورت موازی اجرا شوند. با توجه به محدودیت‌ها (اندازه ماتریس مضربی زوج از ریشه دوم تعداد نخ است)، می‌توانیم هر دو بخش کانولوشن و ادغام را موازی کنیم. با این حال، باید مراقب بود تا از شرایط مسابقه اجتناب شود، به خصوص هنگام نوشتن روی آرایه mat.
در اینجا نحوه تغییر عملکرد برای استفاده از OpenMP آورده شده است:

```
#include <omp.h> // Include OpenMP header

void conv_pool(int** mat, int** kernel, int N){
 int sum;
 // Parallelize the outer loop with OpenMP
 #pragma omp parallel for private(sum) collapse(2)
 for (int i = 1; i < N-1; i++){
 for (int j = 1; j < N-1; j++) {
 sum = 0;
 for (int k = -1; k < 2; k++)
 for (int l = -1; l < 2; l++)
 sum += mat[i+k][j+l]; // Missing semicolon in original code
 mat[i-1][j-1] = sum;
 }
 }

 // Parallelize the pooling part
 #pragma omp parallel for collapse(2)
 for (int i = 0; i < N-2; i+=2){
 for (int j = 0; j < N-2; j+=2) {
 // Assuming 'a' is meant to be 'mat' as 'a' is not defined
 mat[i/2][j/2] = (mat[i][j] + mat[i+1][j] + mat[i][j+1] + mat[i+1][j+1]) / 4;
 }
 }

 // The resize operation is not parallelized as it is a sequential step
 // equivalent operation: mat = mat[0:(N-2)/2][0:(N-2)/2]
 resize_mat(mat, 0, (N-2)/2, 0, (N-2)/2);

 return;
}
```

مقایسه مصرف حافظه:

نسخه سریال: کل مصرف حافظه در درجه اول ماتریس ورودی و هر حافظه پشته اضافی مورد استفاده برای شاخص های حلقه و متغیرهای موقت است. هیچ تخصیص حافظه پویا اضافی انجام نمی شود.

نسخه موازی: علاوه بر حافظه ای که نسخه سریال استفاده می کند، OpenMP ممکن است از حافظه اضافی برای مدیریت رشته ها (به عنوان مثال پشته های رشته) استفاده کند. با این حال، از آنجایی که ما از حافظه پویا اضافی برای خود محاسبات استفاده نمی کنیم، افزایش استفاده از حافظه ناشی از سربار اجرای موازی به جای مدیریت داده های الگوریتم است.

b) موازی سازی بدون تخصیص حافظه دینامیک اضافی
بله، موازی کردن این تابع بدون استفاده از تخصیص حافظه پویا اضافی امکان پذیر است. نکته کلیدی این است که اطمینان حاصل شود که هر رشته در قسمت جداگانه ای از ماتریس کار می کند که با دیگران همپوشانی ندارد تا از شرایط مسابقه جلوگیری شود. این در نسخه موازی بالا نشان داده شده است. با ساختاربندی دقیق حلقه‌ها و استفاده از متغیرهای خصوصی در صورت لزوم (به عنوان مثال، جمع در بخش کانولوشن)، می‌توانیم از نیاز به حافظه پویا اضافی اجتناب کنیم. استفاده از
#pragma omp parallel for with collapse(2)
تضمین می کند که کار بین رشته ها بدون نیاز به حافظه اضافی برای ذخیره نتایج میانی تقسیم می شود.

نکته پیاده‌سازی: پیاده‌سازی فوق فرض می‌کند که تغییرات ماتریس در طول مراحل کانولوشن و ادغام با یکدیگر تداخل ندارند. این یک جنبه حیاتی برای اطمینان از صحت در نسخه موازی است. تقسیم کار باید به وابستگی های بین بخش های مختلف الگوریتم احترام بگذارد.