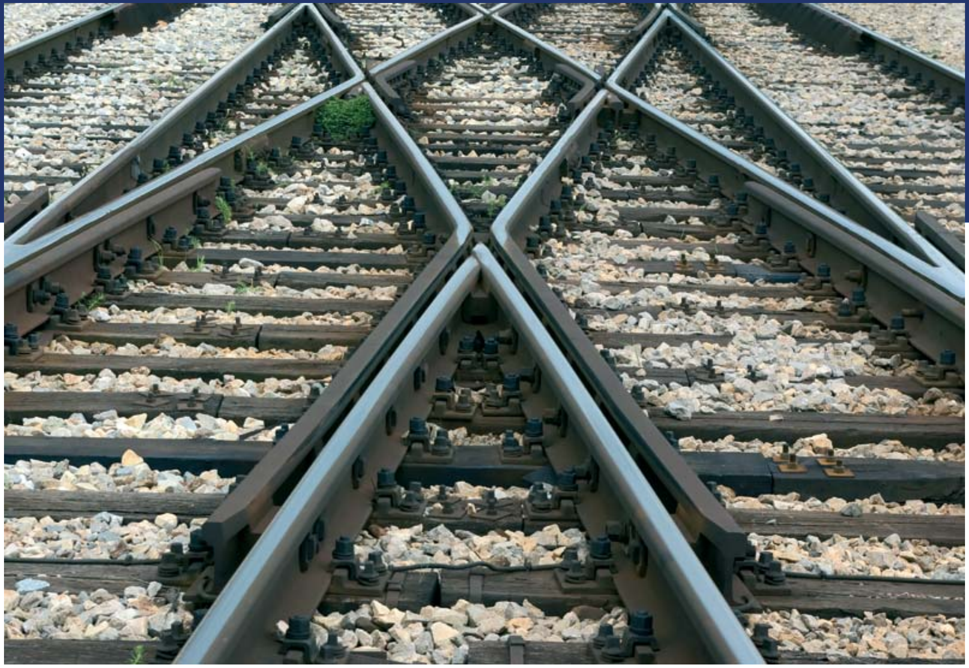
Programming with OpenMP



Libor Bukata a Jan Dvořák

FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

Introduction to OpenMP

- OpenMP (Open Multi-Processing) provides constructs (API) to support parallel programming in C++, C, and Fortran on Linux, MacOS, and Windows.
- A sequential code is transformed to a parallel one by adding pragmas, so if a compiler does not support OpenMP, the pragmas are skipped and the output is a sequential program.
- OpenMP 4.0 added constructs for the vectorization, offloading, and extended tasks.
- OpenMP is used in software like Blender, fftw, OpenBLAS, and eigen to accelerate computations.
- It is relatively easy to use in scientific applications.

First program - Hello world!

```
#include <iostream>
#include <omp.h>

using namespace std;

int main()    {
        int numThreads = omp_get_max_threads();
        cout<<numThreads<<" threads to be spawned..."<<endl;
        #pragma omp parallel
        {
                #pragma omp critical
                cout<<"Hello from thread "<<omp_get_thread_num()<<endl;
        }
        cout<<"Threads finished."<<endl<<endl;

        omp_set_num_threads(12);
        #pragma omp parallel
        {
                #pragma omp critical
                cout<<"Goodbye from thread "<<omp_get_thread_num()<<endl;
        }

        return 0;
}
```

Possible output:

```
4 threads to be spawned...
Hello from thread 3
Hello from thread 2
Hello from thread 0
Hello from thread 1
Threads finished.


Goodbye from thread 1
...
Goodbye from thread 11
```

How to compile and run the program from a command line:

```
g++ -fopenmp -std=c++11 -o your_prog your_prog.cpp
OMP_NUM_THREADS=4 ./your_prog
```

Sequential code – matrix summation

```
#include <iostream>
#include <chrono>
#include <vector>

using namespace std;
using namespace std::chrono;

using MatrixRow = vector<double>;
using MatrixColumn = vector<double>;
using Matrix = vector<vector<double>>;

int main()      {
        // Initialize the matrix by zero values.
        constexpr int M = 8000, N = 8000;
        Matrix m(M, MatrixRow(N, 0.0));

        // Fill the matrix such that total sum is one.
        for (int i = 0; i < M; ++i)     {
                for (int j = 0; j < N; ++j)
                        m[i][j] = 1.0/(M*N);
        }

        ...
```

Sequential code – matrix summation

```
    ...

    high_resolution_clock::time_point start = high_resolution_clock::now();

    // Sum all the entries in the matrix.
    MatrixColumn sumOfRows(M, 0.0);
    for (int i = 0; i < M; ++i)     {
        for (int j = 0; j < N; ++j)
            sumOfRows[i] += m[i][j];
    }

    double totalSum = 0.0;
    for (int i = 0; i < M; ++i)
        totalSum += sumOfRows[i];

    cout<<"Sum of matrix elements: "<<totalSum<<endl;
    double runtime = duration_cast<duration<double>>(
                                            high_resolution_clock::now()-start).count();
    cout<<"Total runtime: "<<runtime<<" s"<<endl;

    return 0;
}
```

OpenMP – the first attempt

```
#include <iostream>
#include <chrono>
#include <vector>
#include <omp.h>

using namespace std;
using namespace std::chrono;

    ...

    // Sum all the entries in the matrix.
    MatrixColumn sumOfRows(M, 0.0);
    for (int i = 0; i < M; ++i)    {
        #pragma omp parallel for
        for (int j = 0; j < N; ++j)
            sumOfRows[i] += m[i][j];
    }

    ...
```

It declares OpenMP functions accessible from API.

Distribution of the loop work between threads.

We slightly accelerated the algorithm but the code is incorrect!
The expected result, i.e. 1.0, cannot be achieved as threads race each other in accessing elements of `sumOfRows` vector.

Correction of the calculation

Version that is using lock API

```
...
omp_lock_t updateLock;
omp_init_lock(&updateLock);

// Sum all the entries in the matrix.
MatrixColumn sumOfRows(M, 0.0);
for (int i = 0; i < M; ++i)     {
        #pragma omp parallel for
        for (int j = 0; j < N; ++j)     {
                omp_set_lock(&updateLock);
                sumOfRows[i] += m[i][j];
                omp_unset_lock(&updateLock);
        }
}

omp_destroy_lock(&updateLock);
...
```

Simple version that is using pragmas

```
...
// Sum all the entries in the matrix.
MatrixColumn sumOfRows(M, 0.0);
for (int i = 0; i < M; ++i)     {
        #pragma omp parallel for
        for (int j = 0; j < N; ++j)     {
                #pragma omp critical
                sumOfRows[i] += m[i][j];
        }
}
...
```

The parallel code is correct, but it is more than two orders of magnitude slower than the sequential one!

Using atomics instead of locks

Version that is using atomics

```
...
// Sum all the entries in the matrix.
MatrixColumn sumOfRows(M, 0.0);
for (int i = 0; i < M; ++i)    {
        #pragma omp parallel for
        for (int j = 0; j < N; ++j)    {
                #pragma omp atomic update
                sumOfRows[i] += m[i][j];
        }
}
...
```

Note that it is possible to omit ‘update’ in the pragma since it is used by default.

Atomics operations in OpenMP

```
#pragma omp atomic read
int current_sum = sum;

#pragma omp atomic write
sum = 0.0;

#pragma omp atomic update
sum += increment;

#pragma omp atomic capture
{
        load_sum = sum;
        sum += increment;
}
```

The parallel code is slightly faster but still much slower than the sequential version.

OpenMP reduction

```
...
// Sum all the entries in the matrix.
MatrixColumn sumOfRows(M, 0.0);
for (int i = 0; i < M; ++i)     {
        double rowSum = 0.0;
        #pragma omp parallel for reduction(+: rowSum)
        for (int j = 0; j < N; ++j)
                rowSum += m[i][j];

        sumOfRows[i] = rowSum;
}
...
```

List of variables:
var_1, var_2, …, var_n

Operators:
+, *, -,
&, |, ^, &&, ||,
max, min

Operator dependent initial value.

Finally, the code is correct and it runs faster than the sequential version (speedup 1.87 on Intel Core i7-3520M).

OpenMP - distribution of work

Use this modified snippet of the code to explore how threads are scheduled:

```
...
vector<int> threadMapping(sumOfRows.size());
double totalSum = 0.0;
#pragma omp parallel for reduction(+: totalSum) schedule(...)
for (int i = 0; i < M; ++i)     {
        totalSum += sumOfRows[i];
        threadMapping[i] = omp_get_thread_num();
}

for (int &threadId : threadMapping)
        cout<<" "<<threadId;
cout<<endl;
...
```

static
static, 8
dynamic
dynamic, 8
guided
guided, 8

Based on your experiments answer the following questions:

- How is the work distributed among the threads for various kind of schedules?
- Which kind of schedule divides the work before loop execution?
- What is the default kind of the schedule?
- Try to devise, based on the expected overhead and behaviour of different kinds of schedules, when it is suitable to use static, dynamic, or guided schedule.

Alternative ways of parallelization

Distributing rows of matrix between threads:

```
...
// Sum all the entries in the matrix.
MatrixColumn sumOfRows(M, 0.0);
#pragma omp parallel for if(M*N > 10e6)
for (int i = 0; i < M; ++i)    {
        for (int j = 0; j < N; ++j)
                sumOfRows[i] += m[i][j];
}
...
```

Threads are only created for large matrices. Small matrices are summed sequentially since it does not pays off to create threads.

The most efficient parallel version.

Distributing elements of matrix between threads:

```
...
// Sum all the entries in the matrix.
MatrixColumn sumOfRows(M, 0.0);
#pragma omp parallel for collapse(2) if(M*N > 10e6)
for (int i = 0; i < M; ++i)    {
        for (int j = 0; j < N; ++j)
                sumOfRows[i] += m[i][j];
}
...
```

Higher granularity of the parallelization, the work is divided based on indices i and j.

Unprotected reduction, impossible to use reduction clause since `sumOfRows` is a vector.

OpenMP - Final parallel version

```
...
double totalSum = 0.0;

#pragma omp parallel num_threads(omp_get_max_threads()) if(M*N > 10e6)
{
	double threadSum = 0.0;

	// Sum all the entries in the matrix.
	#pragma omp for collapse(2) nowait
	for (int i = 0; i < M; ++i)	{
		for (int j = 0; j < N; ++j)
			threadSum += m[i][j];
	}

	#pragma omp atomic update
	totalSum += threadSum;

	#pragma omp barrier
	#pragma omp single
	{
		cout<<"thread "<<omp_get_thread_num()<<" prints the results..."<<endl;
		cout<<"Sum of matrix elements: "<<totalSum<<endl;
		double runtime = duration_cast<duration<double>>(
								high_resolution_clock::now()-start).count();
		cout<<"Total runtime: "<<runtime<<" s"<<endl;
	}
}
...
```

It sets the number of threads.

Do not synchronize the threads after the loop.

Atomically add the thread subtotal to the total sum.

Wait until all threads add their subtotal.

Instead of clause `single` it is possible to use `master`. Try to find out, what is the difference.

OpenMP - assignments

- Calculate π by a parallel Monte Carlo method.
- Parallelize the matrix vector multiplication.
- Transform your implementation of LU decomposition such that OpenMP is used instead of C++11 threads/pthreads.
- You can download a summary card from: http://openmp.org/mp-documents/OpenMP-4.0-C.pdf

That’s all!

Thank you for your attention.