Computational Physics (PHYS6350)

Lecture 24: Introduction to parallel computing

April 25, 2023

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Parallel computing

So far, we've dealt with *sequential* computational workflow

- Operations proceed one after another
- Limited by the processing speed of the logical processing unit
  - For instance, CPU clock speeds have not moved much past 3GHz in the past decade

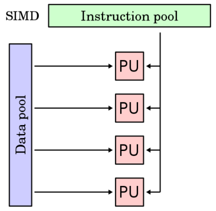
In **parallel computing** multiple processing units perform tasks simultaneously to solve complex problems more efficiently, thereby dividing the workload among multiple processors

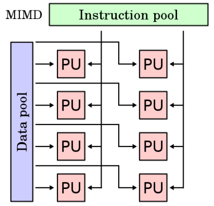
Motivation

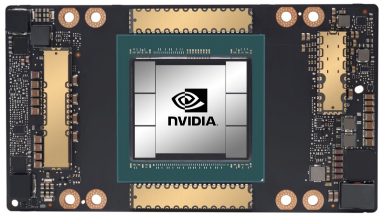
- Better performance (relieves the clock speed bottleneck)
- Resource utilization (use multi-core processors and/or GPUs that are otherwise idle)
- Scalability (adaptation to increasing size and complexity of problems)
- Time-to-solution (get valuable insight from a complex physical simulation in reasonable time)
- Cost-effectiveness

Brief history of parallel computing

- **1940-1960:** Early concepts (von Neumann, IBM,...)
- **1960-1980:** Vector processors, SIMD (single instruction multiple data)
- **1980-2000:** Multi-core architectures, MIMD (multiple instruction multiple data)
- **1990-...:** Clusters and distributed computing
  - Computer networks
  - Many individual computers combine to achieve high performance parallel computing
- **2000-...:** GPUs
  - Originally designed to graphics but have parallel computing potential
  - Suitable for scientific computing, deep learning, data analysis
- **2010-...:** Parallel computing frameworks and cloud computing
  - MPI, OpenMP, OpenCL, CUDA...







Types of parallelism

- **Instruction-Level Parallelism (ILP)**
  - Concurrent execution of multiple instructions within a single processor or core
  - Many compilers optimize code to take advantage of ILP

- **Data-Level Parallelism (DLP)**
  - Performing the same operation on multiple data elements simultaneously
  - Suitable for applications with regular data access patterns
  - Architectures: Single Instruction Multiple Data (SIMD), Vector Processors, Graphics Processing Units (GPUs)
  - Examples: Numerical integration, molecular dynamics step, matrix operations

- **Task-Level Parallelism (TLP)**
  - Concurrent execution of multiple, independent tasks or threads within a program
  - Suitable for applications with irregular data access patterns or complex control flow
  - Programming models: OpenMP (shared-memory), MPI (distributed-memory), CUDA/OpenCL (GPU computing)
  - Examples: Markov Chain Monte Carlo

In practice different types of parallelism are combined together

Parallel Computing Architectures

- **Shared Memory Systems**
  - Multiple processors or cores share a common memory space
  - Communication through memory, no need for explicit message passing
  - Programming models: OpenMP, threads

- **Distributed Memory Systems**
  - Multiple processors or cores, each with its own private memory
  - Communication through message passing between processors
  - Examples: Clusters, Supercomputers, High-Performance Computing (HPC) systems
  - Programming models: Message Passing Interface (MPI)

- **Hybrid Systems**
  - Combination of shared memory and distributed memory architectures
  - E.g. CPU-GPU systems
  - Programming models: OpenMP (shared-memory), MPI (distributed-memory), CUDA/OpenCL (GPU computing)
  - Programming models: OpenMP + MPI, OpenCL/CUDA

Programming Models for Parallel Computing

- **Shared Memory Models**
  - OpenMP (Open Multi-Processing)
    - Compiler directives and runtime library for shared-memory parallelism (C/C++/Fortran)
  - Threads
    - Executing parallel code in threads with control over synchronization (C/C++)
- **Distributed Memory Models**
  - MPI (Message Passing Interface) library
    - Communication and synchronization between processes in distributed-memory systems
- **GPU Computing**
  - NVIDIA CUDA (Compute Unified Device Architecture)
    - Data-parallel programming with GPU-specific language extensions and libraries (C/C++)
  - OpenCL (Open Computing Language)
    - Open standard for parallel programming of heterogeneous systems (CPUs, GPUs, FPGAs), (C/C++)

Performance metrics

- **Speed-up and efficiency**
  - Ratio of the execution time of a serial program to the execution time of its parallel counterpart
  - Ideally: computational time reduced linearly with number of processing elemens (e.g. CPUs)
  - Efficiency is the ratio of speedup to the number of processing elements

- **Scalability**
  - Ability to maintain performance with the increase of
    - number of processing elements
    - problem size

- **Load balancing**
  - Even distribution of work across processing elements and minimization of idle time

- **Overhead**
  - Limitation of performance fue to data exchange and coordination between processing elements

Amdahl's Law

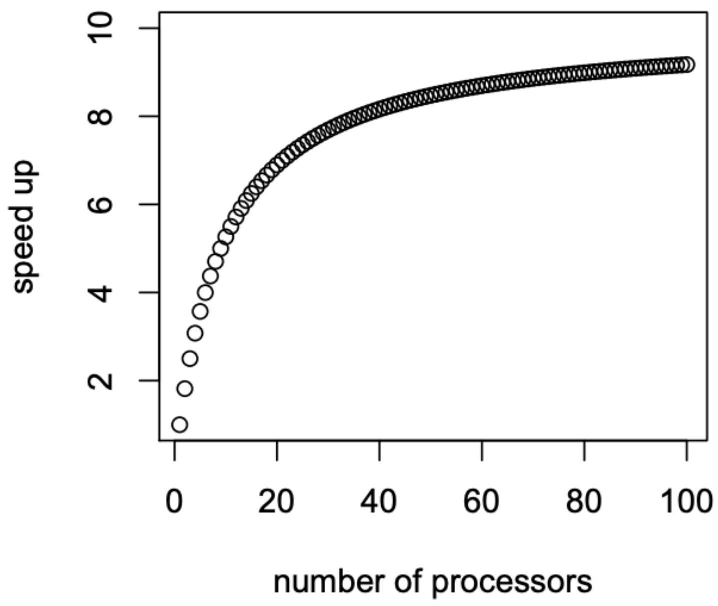
In practice only fraction p of the problem can be parallelized

Amdahl's Law: theoretical speed-up as a function of processing elements n and p

$$S = \frac{1}{(1-p) + p/n}$$

no infinite speed-up possible

Example: p = 90%



Gustafson's Law: Linear speed-up with n if problem size also increases

If the problem is small, parallel computing does not give much advantage

Embarrassingly Parallel Computations

Task can be divided into parts that can be executed separatedly

Examples:

- Numerical integration
  - Split the integration region into many parts, computed them concurrently, and combine the results
- Monte Carlo simulations (generating events in parallel)
  - Generate random samples (events) in parallel using different random seed, then combine the statistics
- Matrix operations
  - E.g. matrix multiplication where each element can be computed concurrently

Sample workflow:

```bash
#!/bin/bash
# Run 8 jobs in parallel
NJOBS=8
for JOB_NUMBER in {1..$NJOBS}; do
    # Run the subtask in the background and move on to the next one
    ./subtask $JOB_NUMBER &
done

# Wait for all the subtasks to finish
wait

# Combine the results
./combine_results
```

Example: Matrix Multiplication

In matrix multiplication, **C = A*B**, each element of $C_{i,j}$ can be computed independently of all other elements

matrix_mult_openmp.cpp:

```
// Set the number of threads to use for the parallel region
omp_set_num_threads(num_threads);
```

...

```
// Perform matrix multiplication using OpenMP
#pragma omp parallel for collapse(2)
for (int i = 0; i < matrix_size; i++) {
    for (int j = 0; j < matrix_size; j++) {
        for (int k = 0; k < matrix_size; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
Usage: ./matrix_mult_openmp <matrix_size> <num_threads>
```

Example: Matrix Multiplication

In matrix multiplication, **C = A*B**, each element of $C_{i,j}$ can be computed independently of all other elements

matrix_mult_openmp.cpp:

```
// Set the number of threads to use for the parallel region
omp_set_num_threads(num_threads);
```

...

```
// Perform matrix multiplication using OpenMP
#pragma omp parallel for collapse(2)
for (int i = 0; i < matrix_size; i++) {
    for (int j = 0; j < matrix_size; j++) {
        for (int k = 0; k < matrix_size; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
Usage: ./matrix_mult_openmp <matrix_size> <num_threads>
```

Example: Multiplying 1000x1000 matrices

1 thread:

```
(base) vlvovch@MacBook-Pro 15_ParallelComputing % ./matrix_mult_openmp 1000 1
Matrix multiplication took 5237 milliseconds.
```

8 threads:

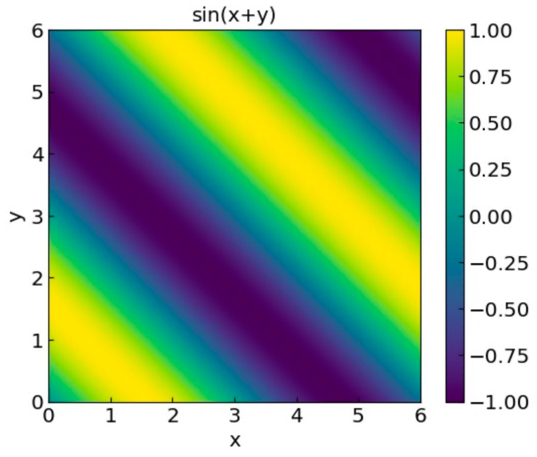
```
(base) vlvovch@MacBook-Pro 15_ParallelComputing % ./matrix_mult_openmp 1000 8
Matrix multiplication took 739 milliseconds.
```

Study the performance (here 500x500):

Threads	Avg Wall Time (ms)	Standard Error (ms)	Speedup Factor	Speedup SE	Efficiency	Efficiency SE
1	729.80	0.92	1.000	0.000	1.000	0.000
2	375.90	1.00	1.941	0.006	0.971	0.003
3	255.80	1.19	2.853	0.014	0.951	0.005
4	190.60	1.51	3.829	0.031	0.957	0.008
5	155.10	1.22	4.705	0.038	0.941	0.008
6	128.00	0.65	5.702	0.030	0.950	0.005
7	111.40	0.60	6.551	0.036	0.936	0.005
8	100.70	1.33	7.247	0.096	0.906	0.012

Example: Multi-dimensional numerical integration

Recall the integral: $I = \int_0^{\pi/2} dx_1 \ldots \int_0^{\pi/2} dx_D \, \sin(x_1 + x_2 + \ldots + x_D).$



Use rectangle rule and OpenMP to compute in 3D

rectanglerule_multi_openmp.cpp

```
double rectangle_rule_multi(double a, double b, int N) {
    double h = (b - a) / N;
    double sum = 0.;

    #pragma omp parallel for reduction(+:sum) collapse(3)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                double x1 = a + i * h + h / 2.;
                double x2 = a + j * h + h / 2.;
                double x3 = a + k * h + h / 2.;
                sum += f(x1, x2, x3);
            }
        }
    }

    return pow(h, Ndim) * sum;
}
```

Example: Use 1000 slices in each dimension

1 thread:

```
(base) vlvovch@MacBook-Pro 15_ParallelComputing % ./rectanglerule_multi_openmp 1000 1
Integral: 2.000000617869063
Numerical integration took  19152 milliseconds.
```

8 threads:

```
(base) vlvovch@MacBook-Pro 15_ParallelComputing % ./rectanglerule_multi_openmp 1000 8
Integral: 2.000000616883211
Numerical integration took  2572 milliseconds.
```

Example: Lennard-Jones molecular dynamics on NVIDIA CUDA

Classical Molecular Dynamics and N-body problem

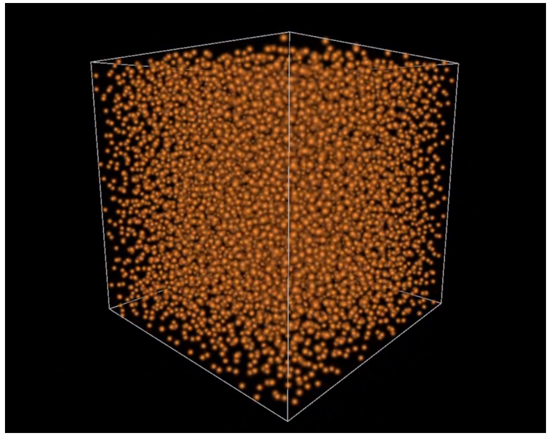
Each step of ODE integration involves computing all the forces

- $O(N^2)$ complexity
- Makes simulation of large systems expensive

```
vector<double> forces(N);
for(int i = 0; i < N; ++i) {
    forces[i] = 0;
    for (int j = 0; j < N; ++j) {
        if (i != j) {
            forces[i] += f(i, j);
        }
    }
}
```

Parallel algorithm:

- Each force can be computed independently of other forces
- Ideally suited for GPUs (large number of parallel threads)
- Things to keep in mind:
  - Need to move data between global and local GPU memory
  - Threads need to be synchronized each integration step



Implementation on CUDA for Lennard-Jones fluid:

open source: https://github.com/vlvovch/lennard-jones-cuda

see also: https://developer.nvidia.com/cuda-code-samples