Matrix-Multiply: SIMD and Task Parallelism

Matrix Multiply

This text describes a **divide-and-conquer** algorithm for matrix multiplication. The **divide-and-conquer** design pattern recursively breaks down a large problem into smaller, independent sub-problems. The cost of dividing the problem and combining the partial solutions should be low compared to the cost of solving the sub-problems themselves. Eventually, the sub-problems become small enough that a direct serial computation is more efficient than further subdivision.

The proposed divide-and-conquer algorithm for matrix multiplication decomposes the multiplication of two n×n matrices into eight sub-problems. Each of these involves multiplying matrices of size n/2 × n/2. A diagram on the right illustrates this strategy.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

8 multiplications of $n/2 \times n/2$ matrices.
1 addition of $n \times n$ matrices.

We provide a C++ recursive implementation that assumes the matrix dimension, N, is an exact power of two. Since two-dimensional matrices are represented using one-dimensional arrays, the recursive function must receive not only the size of the current sub-matrix (SZ) but also the dimension of the original matrix (N). This original dimension N is essential for correctly calculating element positions within the linearized array. A threshold value, DQSZ, determines when a sub-problem is sufficiently small to be solved directly, bypassing further recursion. The code uses a C++ template to provide flexibility in choosing the matrix element data type at compile time.

```
template <class type>
  void MM_DQ ( const type *a, const type *b, type *c, int SZ, const int N )
{
  // SZ: dimension of submatrices a, b and c.
  // N:  size of original input matrices (size of a row)

  if (SZ <= DQSZ)
  { // Classical algorithm for base case
    for (int i=0; i<SZ; i++)
      for (int j=0; j<SZ; j++)
        for (int k=0; k<SZ; k++)
          c[i*N+j] += a[i*N+k]*b[k*N+j];
    return;
  }

  // Divide task into 8 subtasks
  SZ = SZ/2;  // assume SZ is a perfect power of 2

  MM_DQ<type> ( a,          b,          c,          SZ, N);
  MM_DQ<type> ( a,          b+SZ,       c+SZ,       SZ, N);
  MM_DQ<type> ( a+SZ*N,     b,          c+SZ*N,     SZ, N);
  MM_DQ<type> ( a+SZ*N,     b+SZ,       c+SZ*(N+1), SZ, N);
  MM_DQ<type> ( a+SZ,       b+SZ*N,     c,          SZ, N);
  MM_DQ<type> ( a+SZ,       b+SZ*(N+1), c+SZ,       SZ, N);
  MM_DQ<type> ( a+SZ*(N+1), b+SZ*N,     c+SZ*N,     SZ, N);
  MM_DQ<type> ( a+SZ*(N+1), b+SZ*(N+1), c+SZ*(N+1), SZ, N);
}
```

The partial results from each sub-matrix multiplication are stored directly in the pre-allocated output matrix, **c**. This approach eliminates the need for dynamic memory allocation for temporary partial results. By doing so, it avoids a final summation step after all partial results are generated, as illustrated in the top-right diagram. Instead, the resulting sub-matrices are accumulated directly into their corresponding section of the output matrix (the addition is performed in-place on matrix **c**). The code provided also includes a classic iterative implementation, enabling a direct comparison of both performance and the relative ease of parallelizing each approach.

The C++ program code is split across two files: MM_main.cpp and MM.h. The two functions discussed previously are located in MM.h. This structure facilitates the creation of new code versions (e.g., MMv1.h, MMv2.h), where only these two functions need to be modified, leaving the common code in MM_main.cpp untouched.

To compile the program, since the code is written in C++, you must use the **g++** command instead of **gcc**. To simplify generating different versions, the -DTYPE=value flag is used to define the matrix data type, and the -include MM.h flag is used to select the file containing the desired version of the two main functions.

```
$ g++ -Ofast -fopenmp -DTYPE=int -include MM.h MM_main.cpp -o MM

$ MM 1024 0      ← first argument defines N, second argument= 0 indicates iterative version

$ MM 1024 32     ← second argument defines DQSZ: threshold to stop recursion
```

Guide to Parallelization

Step 1: Theoretical Analysis (on paper)

Perform this analysis without modifying or compiling any code.

- For the **Iterative** Parallel Version: it must <u>statically</u> divide the work of multiplying two matrices into **k** parallel tasks, where **k** is the number of threads specified for execution. For simplicity, assume **k** = 2. Illustrate which sections of matrices **a**, **b**, and **c** are read and/or written by each of the two threads.
- For the **Divide-and-conquer** Parallel Version: it can generate a total number of tasks, **t**, that is much larger than the number of concurrent threads, **k**. Perform a theoretical analysis to calculate the following for the cases listed below:
  1. The total **number of tasks** generated.
  2. The **maximum number of concurrent tasks** (i.e., the number that could be executing simultaneously with an unlimited number of threads).

  These two values are not identical if synchronization operations force certain tasks to wait for the completion of previous ones.

  Analysis Cases:
  ```
      N = 1024 and DQSZ = 512
      N = 1024 and DQSZ = 128
      N = 1024 and DQSZ = 16
  ```

Methodology:

To perform this analysis, you must first draw a task graph for the simplest case (`N=1024, DQSZ=512`). This graph should depict the tasks generated during execution and their dependencies. Once you understand this first case, you can analyze the other two theoretically. Furthermore, for the **divide-and-conquer** parallel version you graphed, illustrate which sections of matrices **a**, **b**, and **c** are read and/or written by each task. **Important**: Verify with your instructor that the task graph you have drawn is correct before proceeding.

Step 2: Practical Implementation

Parallelize the **iterative** version using loop-based parallelism and the **divide-and-conquer** version using task-based parallelism. **Code Structure**: To do this, create a new header file named MMpar.h where all parallel implementations and modifications will be made. You must not modify the main file, MM_main.cpp. **Thread Management**: If the number of threads (**k**) is not explicitly specified, it will be determined by the Operating System based on the characteristics of the processor. Use the **perf stat** command to verify the number of threads created during the execution of the parallel versions.

Validation: The program includes a built-in functionality to thoroughly verify the correctness of your parallel implementations. You can pass a third parameter to define how many times the new version should be executed and have its results compared against the original, sequential version. Example:

```
$ MM 1024 256 2    ← the value 2 as the 3rd parameter indicates the result is verified twice
```

It is inherently difficult to demonstrate that concurrency errors can occur. The probability of a data race actually altering the final result is often small, and when it does, the change may be subtle and hard to detect. Furthermore, these problematic execution scenarios are notoriously difficult to replicate. To systematically force these errors to appear, we recommend the following strategy: (1) **Use Small Matrices**: Employ 8x8 element matrices. This minimizes the memory space in which threads write, thereby statistically increasing the probability that two threads will attempt to write to the same memory location; (2) **Limit Task Granularity**: Small matrices naturally generate a small number of tasks, making it easier to reason about and observe thread interactions; and (3) **Introduce Execution "Noise"**: Provoke threads to yield execution voluntarily or be descheduled by the OS. This creates non-deterministic "noise" that disrupts the natural flow of execution and significantly increases the likelihood of overlapping, unsafe memory accesses.

```
$ MMpar 8 0 100000  ← iterative version with 8x8 Matrices, check 100K times
$ MMpar 8 4 100000  ← 8x8 Matrices decomposed into 4x4 matrices, check 100K times
$ MMpar 8 2 100000  ← 8x8 Matrices decomposed into 2x2 matrices, check 100K times
```

Demonstrate that your parallel versions are functionally correct. Once you believe they are correct, intentionally introduce modifications (like removing synchronization operations) to create data races, generating executions that are detected as incorrect.

Matrix-Multiply: SIMD and Task Parallelism

Step 3: Performance Analysis

1. Establish a **Baseline**: Begin by measuring the performance of the original single-threaded (serial) version.
2. **Optimization Goal**: Your objective is to find a configuration for the parallel version that delivers fast performance for multiplying two matrices of size N = 2048.
3. **Systematic Tuning**: Perform this tuning intelligently by investigating the effect of the two key parameters you can control: **DQSZ** (base case size for the recursive algorithm) and **k** (number of threads).
4. **Analysis Methodology**: (1) *Isolate Variables*: Conduct simple comparisons where only a single parameter is changed at a time. This allows you to understand the individual impact of each parameter. (2) *Explain the Effects*: Analyze and explain how each parameter (DQSZ and k) influences performance. (3) *Identify Bottlenecks*: For each configuration you test, try to identify the primary performance bottleneck (e.g., thread management overhead, cache inefficiency, load imbalance). A comprehensive analysis is complex, so focus on this controlled, step-by-step approach to draw meaningful conclusions.

Step 4: Additional Optimizations

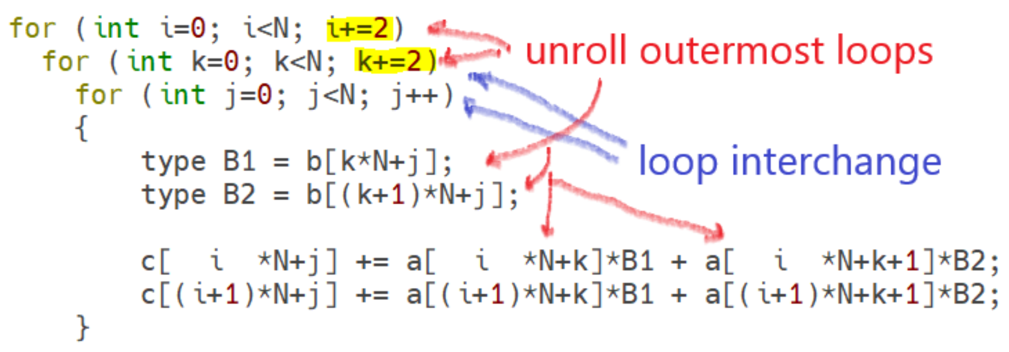
While this step would ideally be performed before parallelization, we will now apply the following two code transformations to optimize both the Classic and the Divide-and-Conquer versions of the program. Evaluate the performance improvement for matrix sizes N = 2048 and N = 4096 and explain the reasons for the performance gain. You may fine-tune a fast parallel version specifically for the N = 2048 case to measure these optimizations.

Optimization 1: Loop Interchange

Interchange the positions of the two innermost loops in the matrix multiplication kernel. The goal is to ensure the memory access pattern is cache-friendly, accessing matrix elements in the order they are stored in memory (row-major order in C/C++) to maximize spatial locality and minimize cache misses.

Optimization 2: Register Blocking / Loop Unrolling

Apply register blocking by unrolling the two outermost loops. This technique encourages the compiler to hold small blocks of the matrices in CPU registers for reuse, significantly reducing the total number of accesses to the main memory (RAM) and alleviating the CPU's memory bandwidth bottleneck.



```
for (int i=0; i<N; i+=2)
  for (int k=0; k<N; k+=2)
    for (int j=0; j<N; j++)
    {
        type B1 = b[k*N+j];
        type B2 = b[(k+1)*N+j];

        c[  i  *N+j] += a[  i  *N+k]*B1 + a[  i  *N+k+1]*B2;
        c[(i+1)*N+j] += a[(i+1)*N+k]*B1 + a[(i+1)*N+k+1]*B2;
    }
```

The previous code optimizations are provided on the file MM2.h. To generate a new version of the code you just have to replace the include file, as shown next.

```
$ g++ -Ofast -fopenmp -DTYPE=int -include MM2.h MM_main.cpp -o MM2
```