OpenMP Performance Considerations

By
Dr. Sandhya Parasnath Dubey and Dr. Manisha
Assistant Professor
Department of Data Science and Computer Applications - MIT,
Manipal Academy of Higher Education, Manipal-576104, Karnataka, India.
Email: sandhya.dubey@manipal.edu; manisha.mit@manipal.edu

OpenMP Performance considerations

- Major performance considerations
- How to measure performance
- How to avoid common performance problems

Introduction

- It may be possible to quickly write a correctly functioning OpenMP program
- But not so easy to create a program that provides the **desired level of performance**
  - It is often because some basic programming rules have not been adhered to.
- Just like in sequential programming, there are some well-established rules of thumb for writing efficient code.
  - Guarantees certain base level performance.
  - This can also be extended to OpenMP programs.
- Best practice: **Write an efficient sequential program. Then introduce OpenMP constructs.**

Performance Considerations for Sequential Programs

Memory Access Patterns and Performance

- **Memory hierarchy in modern computer systems:** Modern processors are fast, but they rely on different levels of memory to keep data accessible as quickly as possible. The memory closest to the processor is called cache.
- **Cache levels:** The cache is typically divided into several levels (L1, L2, L3), with L1 being the smallest and fastest and L3 being larger but slower. The main memory, or RAM, is even larger but significantly slower than the cache.
- When your program accesses data, it first looks in the cache. If the data isn't there, a **cache-miss** occurs, and the system has to fetch the data from the slower main memory. This process is much more expensive in terms of time.
- **Cache-miss** on highest level of memory hierarchy is expensive.
  - **5-10 times** more expensive than fetching the data from the cache
  - **Higher frequency- poor program performance:** If your program frequently misses the cache, its performance will drop significantly.
- In a shared memory systems:
  - The adverse effect is more
  - More number of threads are involved
  - **A cache-miss: results in additional traffic on the system interconnect**
  - No systems in the market has interconnect with sufficient bandwidth

Memory Access Patterns

- **Memory Hierarchy:**
  - **Main Memory (RAM):** The largest and slowest part of memory.
    - Main memory is organized into pages, and only a subset of these pages is available to a given application at any time.
  - **Cache Memory:** Levels closer to the processor are progressively smaller but faster.
    - This includes L1, L2, and L3 caches. These caches hold data that is frequently accessed to speed up processing.
- When the program is compiled, the compiler will arrange its data objects to be stored in the main memory.
- They will be transferred to cache when needed.
- If data is not present in the cache, a cache-miss occurs. The data must be retrieved from the slower main memory, which incurs a performance penalty.

Memory Access Patterns

- If the data requested is not present in cache, its known as Cache-miss.
- It must be **retrieved** from higher levels of the memory hierarchy. This retrieval process is slower than accessing data already present in the cache.
- Data is brought into the cache in chunks called **blocks**.
- If the cache is full and new data needs to be loaded, some existing data may need to be removed or evicted to make room for the new block.
- **Memory hierarchy cannot be programmed by the programmer or the compiler.**
- **We can only control the data fetched into the cache and evicted from the cache.**
- The goal is to reduce cache-misses by organizing data accesses so that data remains in the cache as long as possible and is used efficiently.

Memory Access Patterns

- Example: Let's consider Arrays
  - C typically specify that the elements of arrays be stored contiguously in memory.
  - Thus, if an array element is fetched into cache, "nearby" elements of the array will be in the same cache block and will be fetched as part of the same transaction.
  - If a computation that uses any of these values can be performed while they are still in cache, it will be beneficial for performance.
- Rowwise storage:
  - For example, consider a 2D array stored in row-major order.
  - Accessing elements in a row consecutively makes efficient use of the cache because adjacent elements are loaded into the cache together.
  - If a computation can be performed while the data is still in the cache, it will improve performance.

Memory Access Patterns

- The way you access elements in an array can impact cache performance.

Row-wise Array Access

```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        sum += a[i][j];
```

Figure 5.1: **Example of good memory access** – Array a is accessed along the rows. This approach ensures good performance from the memory system.

- Elements of the array are accessed along the rows.
- When data is fetched into the cache, all elements of the current row are brought in as a block. For example, if you access array[i][j] and then array[i][j+1], these elements are likely in the same cache block, which means accessing them is fast and cache-efficient.
- Since all elements in the current row are used before moving to the next row, it minimizes cache-misses and makes efficient use of the cache.
- This type of access pattern is often referred to as "**unit stride**."

Column-wise Array Access

```
for (int j=0; j<n; j++)
    for (int i=0; i<n; i++)
        sum += a[i][j];
```

Figure 5.2: **Example of bad memory access** – Array a is accessed columnwise. This approach results in poor utilization of the memory system. The larger the array, the worse its performance will be.

- Elements of the array are accessed along the columns.
- In contrast, the loop in Figure 5.2 is not cache friendly. Each reference to a[i][j] may bring a new block of data into the cache.
- The next element in the line it occupies is not referenced until the next iteration of the outer loop is begun, by which time it might have been replaced in cache by other data needed during the execution of the inner loop.

Translation-Lookaside Buffer

- What is meant by virtual memory?
  - Virtual Memory is a memory management technique used by computer systems to create an illusion of a large, continuous block of memory for applications, even if the physical memory (RAM) is smaller or fragmented. It allows the system to run larger applications or multiple programs simultaneously without being limited by the actual amount of physical memory available.
- How are the programs stored in the main memory? What do you mean by pages and frames?
  - Programs are divided into smaller chunks known as **pages**. These pages are stored in the main memory in units called **frames**. Each page of the program is mapped to a specific frame in the main memory.
  - The size of a page and a frame is typically the same, allowing for a straightforward mapping between the two.
  - **Pages**: A page is a fixed-length block of data that represents a portion of a program's virtual memory. Pages are logical units created by the operating system.
  - **Frames**: A frame is a fixed-size block of physical memory in the main memory where a page is stored.

Translation-Lookaside Buffer

- Why we need page table?
  - The Page Table is a data structure used by the operating system to keep track of the mapping between virtual addresses and physical addresses.
  - It helps in translating the virtual address (used by programs) into the physical address (used by the hardware).
  - The page table stores the base address of each page in physical memory and is essential for accessing the correct data.
  - When a program accesses a memory address, the page table is consulted to find out where in physical memory that address is located. Without a page table, the system wouldn't know where to look in physical memory for the data corresponding to a virtual address.
- What is meant by TLB?
  - The Translation-Lookaside Buffer (TLB) is a specialized cache used by the CPU to reduce the time taken to access the page table. It stores a small number of page table entries, which speeds up the translation of virtual addresses to physical addresses.
  - The TLB is critical for performance because it's involved in every memory access. If the information needed to determine the physical location of data is not in the TLB (known as a TLB miss), the processor has to access the page table, which takes more time.
  - TLB Miss: When the required translation is not found in the TLB, the processor waits until the page table is accessed and the translation is completed, leading to a delay in execution.

Loop Optimizations

- **Loop Interchange** is a technique where you swap the order of nested loops in a program. This can help improve the program's performance by making better use of the memory hierarchy, particularly the cache.
- The order in which loops are executed can greatly affect how data is accessed in memory. By changing the loop order, you can make the program access memory in a more efficient way, reducing cache misses and improving overall performance.
- This strategy is called **loop interchange** (or **loop exchange**).

```
for (int j=0; j<n; j++)
    for (int i=0; i<n; i++)
        sum += a[i][j];
```

```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        sum += a[i][j];
```

Loop Optimizations

- How efficiently the program execute a loop can heavily influence overall performance.
- When loops access arrays, the order in which data is accessed is critical. If data is accessed in a way that aligns with how it is stored in memory (such as in row-major order), the program will benefit from better cache utilization.
- A programmer should consider transforming a loop
  - If accesses to arrays in the loop nest do not occur in the order in which they are stored in memory.
  - If a loop has a large body (many operations) and the references to an array element or its neighbors are far apart, this can lead to data being evicted from the cache before it's reused, which is inefficient. (In a large loop where an array element is accessed, and then the loop does many other operations before returning to that element, the data might be evicted from the cache, forcing another slow memory access.)

Loop Optimizations

- They can be applied if the changes to the code do not affect correct execution of the program or output of the program.
- If a memory location is accessed multiple times within a loop (e.g., reading from and writing to the same array element), the order of these accesses is crucial.
- If at least one of these references modifies the value at that memory location, you must ensure that the transformation does not alter the relative order of these operations.
- Changing the order of memory accesses in a loop where values are modified can lead to incorrect results.
- For example, if a value is written to an array and then read again later in the loop, changing the order could mean reading a stale or incorrect value.

Loop Optimizations

- Loop transformations have other purposes:
  - They may help the compiler to better utilize the instruction pipeline or may increase the amount of exploitable parallelism. (CPU can execute instructions more efficiently by keeping the pipeline full, reducing idle times and enhancing overall performance.)
  - They can also be applied to increase the size of parallel regions. (If a loop can be split into independent sections that don't rely on each other's results, those sections can potentially be executed in parallel, speeding up execution.)
- **Loop unrolling**
  - It is a technique where the iterations of a loop are expanded or "unrolled" so that multiple iterations are performed within a single loop cycle.
  - This reduces the overhead associated with loop control (like updating the loop counter and checking the exit condition) and can lead to more efficient execution.
  - Loop unrolling can help to improve cache line utilization by improving data reuse. Since multiple operations are performed on data loaded into the cache, the chances of a cache miss are reduced.
  - It can also help to increase the instruction-level parallelism: more instructions can be executed in parallel, which is especially beneficial on modern CPUs that support multiple execution units.

Loop Optimizations

```
for (int i = 0; i < N; i++) {
    sum += array[i];
}
```

```
for (int i = 0; i < N; i += 4) {
    sum += array[i];
    sum += array[i + 1];
    sum += array[i + 2];
    sum += array[i + 3];
}
```

```
for (int i=1; i<n; i++) {
      a[i] = b[i] + 1;
      c[i] = a[i] + a[i-1] + b[i-1];
}
```

```
for (int i=1; i<n; i+=2) {
      a[i] = b[i] + 1;
      c[i] = a[i] + a[i-1] + b[i-1];
      a[i+1] = b[i+1] + 1;
      c[i+1] = a[i+1] + a[i] + b[i];
}
```

Figure 5.3: **A short loop nest** – Loop overheads are relatively high when each iteration has a small number of operations.

Figure 5.4: **An unrolled loop** – The loop of Figure 5.3 has been unrolled by a factor of 2 to reduce the loop overheads. We assume the number of iterations is divisible by 2.

- In this example, the loop body executes 2 iterations in one pass.
- This number is called the **“unroll factor”.**
- A higher value tends to give better performance but also increases the number of registers needed.
- **If the unroll factor does not divide the iteration count, the remaining iterations must be performed outside this loop nest.**
- **This is implemented through a second loop, the “cleanup” loop.**

Loop Optimizations

- **Unroll and jam** is an extension of loop unrolling that is appropriate for some loop nests with multiple loops.
- It is a loop transformation technique that involves **unrolling** the iterations of an inner loop and then **jamming** or merging the resulting code into the outer loop. This method is particularly useful in **loop nests**—scenarios where one loop is nested inside another.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j] = b[i][j] + c[i][j];
        d[i][j] = e[i][j] + f[i][j];
    }
}
```

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j += 2) {
        a[i][j] = b[i][j] + c[i][j];
        d[i][j] = e[i][j] + f[i][j];

        // Unrolled part for j+1
        if (j + 1 < n) {  // Boundary check to prevent out-of-bounds access
            a[i][j+1] = b[i][j+1] + c[i][j+1];
            d[i][j+1] = e[i][j+1] + f[i][j+1];
        }
    }
}
```

Loop Optimizations

```
for (int j=0; j<n; j++)
   for (int i=0; i<n; i++)
       a[i][j] = b[i][j] + 1;
```

```
for (int j=0; j<n; j+=2){
   for (int i=0; i<n; i++)
       a[i][j] = b[i][j] + 1;
   for (int i=0; i<n; i++)
       a[i][j+1] = b[i][j+1] + 1;
}
```

```
for (int j=0; j<n; j+=2)
   for (int i=0; i<n; i++) {
       a[i][j]   = b[i][j] + 1;
       a[i][j+1] = b[i][j+1] + 1;
}
```

- Each iteration of the unrolled loop must perform the same computation as the original loop.
- Since the original loop only handled one column (j) at a time, unrolling it by 2 means we now handle two columns (j and j+1).
- However, to maintain the row-wise processing for each column, we have to separate the operations into two distinct loops.

Loop Optimizations

- Can this loops be optimized using loop Interchange?

```
for (int j=0; j<n; j++)
 for (int i=0; i<m; i++)
   a[i][j+1] = a[i+1][j] + b;
```

Loop Optimizations

- **Loop fusion** merges two or more loops to create a bigger loop
- This might enable **data in cache to be reused more frequently**
- May **increase the amount of computation per iteration** in order to improve the instruction-level parallelism

```
for (int i=0; i<n; i++)
     a[i] = b[i] * 2;
for (int i=0; i<n; i++)
   {
     x[i] = 2 * x[i];
     c[i] = a[i] + 2;
   }
```

```
for (int i=0; i<n; i++)
 {
     a[i] = b[i] * 2;
     c[i] = a[i] + 2;
     x[i] = 2 * x[i];
 }
```

Loop Optimizations

- **Loop fission** is a transformation that breaks up a loop into several loops
- Sometimes, we may be able to improve use of cache this way
- Isolate a part that inhibits full optimization of the loop
- This technique is likely to be most useful if a loop nest is large and its data does not fit into cache or if we can optimize parts of the loop in different ways

Loop Optimizations

```
for (int i=0; i<n; i++)
 {
  c[i] = exp(i/n) ;
  for (int j=0; j<m; j++)
      a[j][i] = b[j][i] + d[j] * e[i];
 }
```

```
for (int i=0; i<n; i++)
  c[i] = exp(i/n) ;

for (int j=0; j<m; j++)
  for (int i=0; i<n; i++)
      a[j][i] = b[j][i] + d[j] * e[i];
```

Loop Optimizations

- **Loop tiling or blocking**
  - It is a powerful transformation aimed at optimizing the number of memory references inside a loop iteration.
  - This optimization ensures that these references fit into the cache, which can significantly improve the performance of programs, especially when working with large data sets.
  - If data sizes are large and memory access is bad: When data sizes are large, straightforward memory access can be inefficient because not all of the data can fit into the cache. This can lead to frequent cache misses, causing the CPU to wait while data is fetched from the slower main memory.
  - If there is data reuse in the loop: Effective when there is data reuse within the loop. By carefully structuring the loop iterations, the same data can be reused multiple times while it remains in the cache, minimizing memory access time.
- Loop tiling replaces the original loop by a pair of loops.

Loop Optimizations

- **Original Loop:** In a typical loop structure, the entire data set is processed sequentially. For large data, this can lead to inefficient cache usage.
- **Tiled Loop:** Loop tiling breaks the original loop into smaller chunks, or "tiles." These tiles are processed in such a way that they fit into the cache, leading to fewer cache misses and better overall performance.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Matrix Multiplication without Loop Tiling: Every element of matrix A and matrix B is accessed n times, leading to frequent memory access and potential cache misses.

```
int tileSize = 32; // Example tile size
for (int i = 0; i < n; i += tileSize) {
    for (int j = 0; j < n; j += tileSize) {
        for (int k = 0; k < n; k += tileSize) {
            // Process the tile
            for (int ii = i; ii < i + tileSize; ii++) {
                for (int jj = j; jj < j + tileSize; jj++) {
                    for (int kk = k; kk < k + tileSize; kk++) {
                        C[ii][jj] += A[ii][kk] * B[kk][jj];
                    }
                }
            }
        }
    }
}
```

Matrix Multiplication with Loop Tiling: The outer loops (i, j, k) iterate over the matrix in blocks of size tileSize.The inner loops (ii, jj, kk) process each tile.This method ensures that each tile fits into the cache, minimizing cache misses and making memory access more efficient.

```java
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        C[i][j] = 0;
        for (int k = 0; k < 4; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```java
int tileSize = 2;
for (int i = 0; i < 4; i += tileSize) {
    for (int j = 0; j < 4; j += tileSize) {
        for (int k = 0; k < 4; k += tileSize) {
            // Process each tile
            for (int ii = i; ii < i + tileSize; ii++) {
                for (int jj = j; jj < j + tileSize; jj++) {
                    for (int kk = k; kk < k + tileSize; kk++) {
                        C[ii][jj] += A[ii][kk] * B[kk][jj];
                    }
                }
            }
        }
    }
}
```

```
A = | 1  2  3  4 |   B = | 5   6   7   8 |
    | 5  6  7  8 |       | 9  10  11  12 |
    | 9 10 11 12 |       | 13 14  15  16 |
    | 13 14 15 16 |      | 17 18  19  20 |

C = | 0 0 0 0 |
    | 0 0 0 0 |
    | 0 0 0 0 |
    | 0 0 0 0 |
```

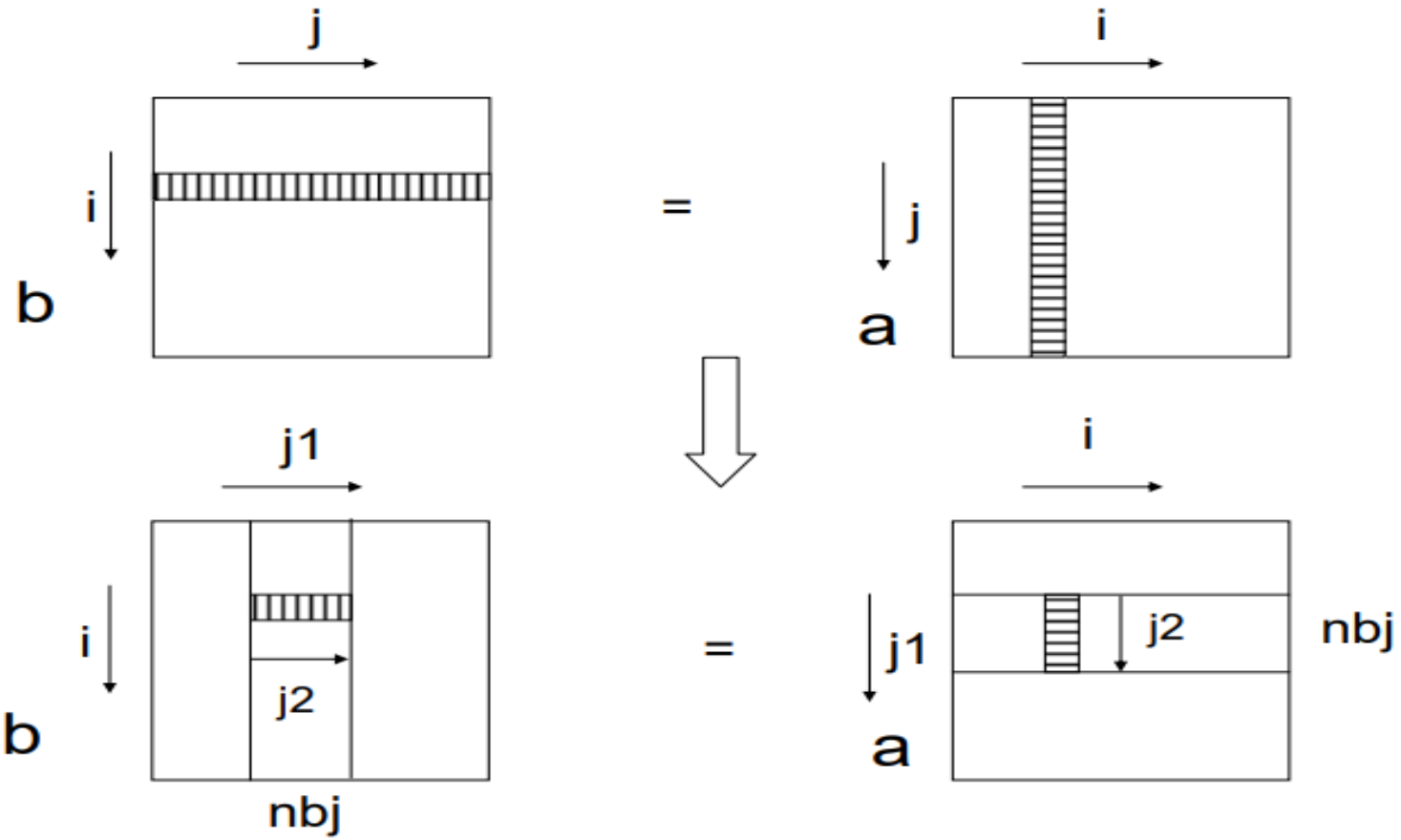
```
C = | 23 26 0 0 |
    | 79 90 0 0 |
    |  0  0 0 0 |
    |  0  0 0 0 |
```

Output after first iteration: Processing the First Tile (Top-left 2x2 submatrix): (i=0, j=0, k=0)

Loop Optimizations

```
for (int i=0; i<n; i++)
  for (int j=0; j<m; j++)
      b[i][j] = a[j][i];
```

```
for (int j1=0; j1<n; j1+=nbj)
  for (int i=0; i<n; i++)
   for (int j2=0; j2 < MIN(n-j1,nbj); j2++)
       b[i][j1+j2] = a[j1+j2][i];
```



Use of Pointers and Contiguous Memory in C

- The memory model in C is such that, without additional information, one must assume that all pointers may reference any memory address
  - This is generally referred to as the **pointer aliasing problem. It occurs when two or more pointers reference the same memory location.**
  - It prevents a compiler from performing many program optimizations because it cannot guarantee that operations on one pointer do not affect operations on another.
- If pointers are guaranteed to point to portions of nonoverlapping memory, optimizations can be applied**.**

```c
void optimized_example(int *a, int *b) {
    for (int i = 0; i < 10; i++) {
        a[i] = a[i] + b[i];  // No aliasing if a and b are non-overlapping
    }
}
```

```c
int arr[10];
optimized_example(arr, arr);  // a and b are the same, hence aliased
```

```c
int arr1[10];
int arr2[10];
optimized_example(arr1, arr2);  // a and b are different, no aliasing
```

- If a and b are guaranteed to point to different (non-overlapping) memory regions, the compiler has more freedom to optimize. For instance, it could:
  - **Unroll the loop** to perform multiple additions in parallel.
  - **Reorder operations** for better performance.
  - **Eliminate redundant loads** because it knows that the data in b[i] won't be changed by operations on a[i].

Using Compilers

- Modern compilers use **data dependence analysis** to determine whether different parts of the code can be executed in parallel or if certain optimizations are safe. This analysis helps in loop optimization and code reordering.

Ex: Compiler may optimize loops to parallelize them if it determines that iterations are independent.

- They perform a variety of analyses to determine whether they may be applied
  - The main one is known as data dependence analysis
- Compilers use various techniques, such as loop unrolling, instruction reordering, and reducing the number of operations, to better exploit the hardware and enhance performance.
- It is worthwhile to experiment with compiler options to achieve the maximum performance for your application.

```
gcc -O0 -o my_program my_program.c   # No optimization
gcc -O2 -o my_program my_program.c   # Moderate optimization
gcc -O3 -o my_program my_program.c   # Aggressive optimization
```

Amdahl’s law

- Amdahl’s Law provides a theoretical framework for understanding the potential speedup of a program when parallelizing it.
- It helps estimate the maximum possible speedup and highlights the impact of the non-parallelizable portion of a program.
- If we denote by T1 the execution time of an application on 1 processor, then in an ideal situation, the execution time on P processors should be T1/P
- If TP denotes the execution time on P processors, then the speedup ratio is

$$\mathbf{S = T1/TP}$$

- This ratio indicates how many times faster the parallel execution is compared to the sequential execution.

Amdahl's law

- **Ideal Speedup:** If a program could be perfectly parallelized, the execution time would be:

$$TP_{ideal} = T1/P$$

- The speedup in an ideal scenario is:

$$S_{ideal} = T1/(T1/P) = P$$

- **Realistic Speedup:** Virtually all programs contain some regions that are suitable for parallelization and other regions that are not.
- By using an increasing number of processors, the time spent in the parallelized parts of the program is reduced, but the sequential section remains the same.
- Eventually the execution time is completely dominated by the time taken to compute the sequential portion, which puts an upper limit on the expected speedup.

$$\mathbf{S = 1/ (fpar/P + (1 - fpar))}$$

- fpar is the parallel fraction of the code and P is the number of processors.

Amdahl's law

- Suppose that 70% of a program execution can be speed up if the program is parallelized and run on 16 processing units instead of one. What is the maximum speedup that can be achieved by the whole program?
- What is the maximum speedup if we increase the number of processing units to 32, then to 64, and then to 128

$$\mathbf{S = 1/ (fpar/P + (1 - fpar))}$$

For 16 processors: S = 1/((0.7/16)+(1-0.7)) ≈ **2.91**

For 32 processors: S ≈ **3.11**

For 64 processors: S ≈ **3.22**

For 128 processors: S ≈ **3.27**

Amdahl’s law

Obstacles to Perfect Linear Speedup

Overheads:

- Forking and Joining Threads: Creating and managing threads introduces overhead that can impact performance.
- Thread Synchronization: Synchronizing threads to avoid race conditions can cause delays.
- Memory Accesses: Managing memory access in parallel environments can lead to additional overhead.

Parallel Scalability:

- Program's ability to maintain or improve its performance as the number of processors increases.
- A measure of a program’s ability to decrease the execution time of the code with an increasing number of processors is referred to as **parallel scalability.**
- A program is said to have good scalability if its speedup increases proportionally with the number of processors.

Measuring OpenMP Performance

- To achieve good OpenMP performance, we need to know how it can be measured and identify what factors determine overall program performance.
- On Unix systems if we use standard operating system command: **/bin/time ./a.out**

```
$ /bin/time ./program.exe

real    5.4
user    3.2
sys     1.0
```

Measuring OpenMP Performance

- **Real:** Program took 5.4 seconds from beginning to end. It includes everything that happened while the program was running, including waiting for other processes, input/output operations, and any delays.
- **User:** The time the program spent executing outside any operating system services. This is the time the CPU spent executing the instruction in your program.
- **Sys:** The time spent on operating system services, on behalf of your program, such as handling input/output operations, memory management, or other system-related tasks.

```
$ /bin/time ./program.exe

real    5.4
user    3.2
sys     1.0
```

- **CPU time:** The sum of **user** and **system** time, which in this case is 3.2 + 1.0 = 4.2 seconds. This is the total time the CPU was actively working on your program, either running your code or the operating system's code on behalf of your code.
- The real time is also referred to as **wall-clock time** or **elapsed time.**

Measuring OpenMP Performance

- There is a difference between real time and CPU time.
  - The application did not get a full processor to itself, because of a high load on the system

```
$ /bin/time ./program.exe

real    5.4
user    3.2
sys     1.0
```

- OpenMP program has additional overheads
  - These overheads are collectively called the ***parallel overhead***
  - It includes the time to
    - Create, start, and stop threads.
    - The extra work needed to figure out what each task is to perform.
    - The time spent waiting in barriers and at critical sections and locks.
    - The time spent computing some operations redundantly.
  - Hence, the total CPU time is likely to exceed the CPU time of the serial version.

Measuring OpenMP Performance

$$T_{CPU}(P) = (1 + O_P \cdot P)T_{Serial}$$

$$T_{Elapsed}(P) = ((\frac{f}{p}) + 1 - f + O_P \cdot P)T_{Serial}$$

- T_{Serial} is the CPU time of the original serial version of the application
- P is the number of processors
- $O_P \cdot P$ is the parallel overhead with O_P assumed to be a constant percentage
- $f \in [0, 1]$ is the fraction of execution time that has been parallelized. A value of zero for f implies that application is serial. A perfectly parallel application corresponds to f = 1.

Measuring OpenMP Performance

- If the original program takes T_{Serial} = 10.20 seconds to run and code corresponding to 95% of the execution time has been parallelized. Assume that each additional processor adds a 2% overhead to the total CPU time. Compute Speedup and Efficiency of the parallel program with 4 processors. Also estimate estimate the T_{CPU} and $T_{Elapsed}$ of the given program.
  - **1 Processor:** There's no parallelism, so the time is simply T_{serial} = 10.20 seconds.
  - **2 Processors:** Here, the parallel part is split between the two processors, reducing its time, but overhead adds up.
    - Parallel time reduces by half (f / P becomes 0.95 / 2 = 0.475).
    - Serial time remains (1 − f = 0.05).
    - Overhead is doubled (Op · P = 0.02 · 2 = 0.04).
    - Total time: (0.475 + 0.05 + 0.04) · 10.20 ≈ 5.80 seconds.
  - **4 Processors:** Parallel time is further reduced
    - Parallel time (f / P = 0.95 / 4 = 0.2375).
    - Serial time remains (1 − f = 0.05).
    - Overhead quadruples (Op · P = 0.02 · 4 = 0.08).
    - Total time: (0.2375 + 0.05 + 0.08) · 10.20 ≈ 3.71 seconds.
- Adding processors helps reduce time but not in a perfectly linear way, because of overhead and the remaining serial portion.

Measuring OpenMP Performance

- If the original program takes T_{Serial} = 10.20 seconds to run and code corresponding to 95% of the execution time has been parallelized. Assume that each additional processor adds a 2% overhead to the total CPU time. Compute Speedup and Efficiency of the parallel program with 4 processors. Also estimate estimate the T_{CPU} and $T_{Elapsed}$ of the given program.
  - **Speedup and Efficiency**

➢ Speedup is how much faster the program runs with P processors compared to just 1 processor.

➢ For 4 processors: Speedup = T_{serial} / $T_{Elapsed(4)}$ ≈ 10.20 / 3.71 ≈ 2.75.

➢ Efficiency is the speedup divided by the number of processors.

➢ For 4 processors: Efficiency ≈ 2.75 / 4 ≈ 0.69 (or 69%).

➢ As you keep adding processors, the benefits diminish because overheads increase, and the remaining serial portion (1 − f) becomes a bigger bottleneck.

Measuring OpenMP Performance

- If the original program takes T_{Serial} = 18.3 seconds to run and code corresponding to 73% of the execution time has been parallelized. Assume that each additional processor adds a 4% overhead to the total CPU time. Compute Speedup and Efficiency of the parallel program with 8 processors. Also estimate estimate the T_{CPU} and $T_{Elapsed}$ of the given program.

Measuring OpenMP Performance

- The observable performance of OpenMP programs is influenced by at least the following factors
  - The manner in which memory is accessed by the individual threads.
  - The fraction of the work that is sequential, or replicated **(Sequential overheads)**
  - The amount of time spent handling OpenMP constructs **(Parallelization overheads)**
    - When a work-sharing directive is implemented, the work to be performed by each thread is usually determined at run time.
  - The load imbalance between synchronization points **(Load imbalance overheads)**
    - Threads perform different amounts of work in a work-shared region.
    - Threads might have to wait for a member of their team to carry out the work of a single construct
  - Other synchronization costs **(Synchronization overheads)**

Measuring OpenMP Performance

- Suppose that 65% of a program execution can be sped up if the program is parallelized and run on 8 processing units instead of one. What is the maximum speedup that can be achieved by the whole program? What is the maximum speedup if we increase the number of processing units to 16.
- If the original program takes T_{Serial} = 9.4 seconds to run and code corresponding to 68% of the execution time has been parallelized. Assume that 6 processors incur a total overhead of 0.24 units to the total CPU time. Compute Speedup and Efficiency of the parallel program with 6 processors. Also estimate the T_{CPU} and $T_{Elapsed}$ of the given program.

Best Practices:

- General recommendations on how to write an efficient OpenMP program
- Optimize Barrier Use
  - Barriers are expensive operations
  - Can use Nowait clause to ignore the barriers

```
#pragma omp parallel
{
        .........
  #pragma omp for
    for (i=0; i<n; i++)
        .........
  #pragma omp for nowait
    for (i=0; i<n; i++)

} /*-- End of parallel region - barrier is implied --*/
```

Figure 5.20: **A construct with one barrier less** – The `nowait` clause is used to omit the implied barrier at the end of the second loop.

Since the second parallel loop is back to back with the end of the parallel region, we can safely omit the implied barrier for the second loop

Best Practices

- Optimize Barrier Use
  - First ensure that the OpenMP program works correctly and then use the `nowait` clause where possible, carefully inserting explicit barriers at specific points in the program as needed.
  - When doing this, one needs to take particular care to identify and order computations that write and read the same portion of memory.

```
#pragma omp parallel default(none) \
        shared(n,a,b,c,d,sum) private(i)
{
   #pragma omp for nowait
   for (i=0; i<n; i++)
      a[i] += b[i];

   #pragma omp for nowait
   for (i=0; i<n; i++)
      c[i] += d[i];

   #pragma omp barrier

   #pragma omp for nowait reduction(+:sum)
   for (i=0; i<n; i++)
      sum += a[i] + c[i];
} /*-- End of parallel region --*/
```

Figure 5.21: **A reduced number of barriers** – Before reading the values of vectors a and b all updates on these vectors have to be completed. The barrier ensures this.

Best Practices

- Avoid the Ordered Construct
  - The ordered construct ensures that the corresponding block of code within a parallel loop is executed in the order of the loop iterations.
  - The runtime system has to keep track which iterations have finished and possibly keep threads in a wait state until their results are needed.
  - It is expensive to implement.
  - This inevitably slows program execution.
- The ordered construct can often (perhaps always) be avoided. It might be better, for example, to wait and perform I/O outside of a parallelized loop.

Best Practices

- Avoid Large Critical Regions
  - The more code contained in the critical region, however, the greater the likelihood that threads have to wait to enter it, and the longer the potential wait times

```
#pragma omp parallel shared(a,b) private(c,d)
{
     ......
#pragma omp critical
    {
      a += 2 * c;
      c = d * d;
    }
} /*-- End of parallel region --*/
```

- Each thread updates the shared variable ***a***.
- However, the second statement assigns a new value to a private variable, and hence there is no chance of a data race.
- This second statement unnecessarily increases the amount of work in the region and potentially extends the amount of time threads must wait for each other. It should thus be removed from the critical region.

Figure 5.22: **A critical region** – Without the critical region, the first statement here leads to a data race. The second statement however involves private data only and unnecessarily increases the time taken to execute this construct. To improve performance it should be removed from the critical region.

Best Practices

- Maximize Parallel Regions
  - Overheads are associated with starting and terminating a parallel region
  - Large parallel regions offer more opportunities for using **data in cache and provide a bigger context for other compiler optimizations**
  - Minimize the number of parallel regions

```
#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop 1 --*/
}

#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop 2 --*/
}
        .........

#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop N --*/
}
```

Figure 5.23: **Multiple combined parallel work-sharing loops** – Each parallelized loop adds to the parallel overhead and has an implied barrier that cannot be omitted.

Best Practices

```
#pragma omp parallel
{
    #pragma omp for  /*-- Work-sharing loop 1 --*/
    { ..... }

    #pragma omp for  /*-- Work-sharing loop 2 --*/
    { ..... }

       .........

    #pragma omp for  /*-- Work-sharing loop N --*/
    { ..... }
}
```

Figure 5.24: **Single parallel region enclosing all work-sharing for loops** — The cost of the parallel region is amortized over the various work-sharing loops.

Best Practices

- Avoid Parallel Regions in Inner Loops
  - We repeatedly experience the overheads of the parallel construct
  - the overheads of the ***#pragma omp parallel for*** construct are incurred **n^2** times
  - #pragma omp parallel for directive is inside the j loop. This means that for every iteration of the i and j loops, a new parallel region is created and destroyed. If n is large, this could happen n^2 times, leading to significant overhead because:

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        #pragma omp parallel for
        for (k=0; k<n; k++)
            { .........}
```

Figure 5.25: Parallel region embedded in a loop nest – The overheads of the parallel region are incurred n^2 times.

```
#pragma omp parallel
   for (i=0; i<n; i++)
       for (j=0; j<n; j++)
           #pragma omp for
           for (k=0; k<n; k++)
               { .........}
```

Figure 5.26: Parallel region moved outside of the loop nest – The parallel construct overheads are minimized.

Best Practices

- Address Poor Load Balance
  - In parallel programming, a common challenge is load balancing: ensuring that all threads or processes have roughly the same amount of work to do.
  - When load balancing is poor, some threads may finish their work earlier and sit idle, waiting for the slower threads to complete, which leads to inefficiency.
  - Solution is to use schedule clause
  - The dynamic and guided workload distribution schedules have higher overheads than does the static scheme

```
for (i=0; i<N; i++) {
   ReadFromFile(i,...);
   for (j=0; j<ProcessingNum; j++ )
       ProcessData(); /* lots of work here */
   WriteResultsToFile(i);
 }
```

Figure 5.27: **Pipelined processing** – This code reads data in chunks, processes each chunk and writes the results to disk before dealing with the next chunk.

Best Practices

- To minimize the overheads of setting up parallel regions, the entire computation is enclosed in a single parallel region.
- The code first reads in the chunk of data needed for the first iteration of the i loop. Since execution cannot proceed until this data is available, the implicit barrier is not removed.
- Next, one of the threads starts to read the next chunk of data.
- Because of the nowait clause, the other threads immediately begin to execute the processing loop.
- Dynamic scheduling is used for processing the data. This allows threads to take on new chunks of work as soon as they finish their current task, rather than waiting for all threads to complete a specific chunk. This flexibility helps ensure better load balancing

```
#pragma omp parallel
{
   /* preload data to be used in first iteration of the i-loop */
   #pragma omp single
      {ReadFromFile(0,...);}

   for (i=0; i<N; i++) {

      /* preload data for next iteration of the i-loop */
      #pragma omp single nowait
         {ReadFromFile(i+1...);}

      #pragma omp for schedule(dynamic)
         for (j=0; j<ProcessingNum; j++)
            ProcessChunkOfData();  /* here is the work */
      /* there is a barrier at the end of this loop */

      #pragma omp single nowait
         {WriteResultsToFile(i);}

   }  /* threads immediately move on to next iteration of i-loop */
} /* one parallel region encloses all the work */
```

Figure 5.28: **Parallelized pipelined processing** – This code uses a dynamic work-sharing schedule to overlap I/O and computation.

Additional Performance Considerations

- The Single Construct Versus the Master Construct
  - A **single region** can be executed by any thread, typically the first to encounter it.
  - A single construct has an implicit barrier.
  - Whereas this is not the case for the master region.
  - A master construct can be more efficient: **Single construct requires more work in the OpenMP library.**
  - The single construct might be more efficient if the master thread is not likely to be the first one to reach it and the threads need to synchronize at the end of the block.

Additional Performance Considerations

- Avoid False Sharing
  - One of the factors limiting scalable performance is false sharing
  - Computers use a system called cache to store frequently used data closer to the processor, so it can be accessed faster.
  - But cache isn't managed at the level of individual data pieces; instead, it works with larger chunks called cache lines.
  - When two threads are working on different parts of data that happen to be in the same cache line, they inadvertently interfere with each other.
  - This interference is known as false sharing

```
#pragma omp parallel for shared(Nthreads,a) schedule(static,1)
    for (int i=0; i<Nthreads; i++)
        a[i] += i;
```

Figure 5.29: **Example of false sharing** – `Nthreads` equals the number of threads executing the `for`-loop. The chunk size of 1 causes each thread to update one element of `a`, resulting in false sharing.

- Each thread updates a different element of array a.
- However, if a[i] and a[i+1] reside on the same cache line, updating a[i] by one thread will invalidate the cache line for other threads that are working on adjacent elements (like a[i+1]).
- This causes significant performance issues, especially if the updates are frequent.

Additional Performance Considerations

- Avoid False Sharing
  - array padding can be used to eliminate the problem
  - changing the indexing from a[i] to a[i][0] eliminates the false sharing
- False sharing is likely to significantly impact performance under the following conditions:
  - Shared data is modified by multiple threads
  - The access pattern is such that multiple threads modify the same cache line(s)
  - These modifications occur in rapid succession: If these modifications happen quickly and repeatedly, the performance hit due to false sharing becomes more pronounced, as the system constantly invalidates and reloads cache lines.

Additional Performance Considerations

- Shared Data refers to variables that are accessible by all threads in a parallel region. This can be beneficial when:
- Data Needs to be Accessible by All Threads: If multiple threads need to read from the same data, shared data is practical.
- Data Is Read-Only: If data is only being read and not modified, sharing it is efficient as it avoids unnecessary duplication.
- False Sharing: This occurs when threads on different processors modify variables that reside on the same cache line. This can lead to performance degradation due to frequent invalidation of cache lines.
- Concurrency Issues: When multiple threads write to the same shared variable, it can lead to race conditions unless properly managed with synchronization mechanisms.
- Private data is data that is unique to each thread. Each thread has its own copy of this data, which can be beneficial when:
- Data Needs to Be Modified Independently: If each thread needs to work with its own copy of data, private data ensures that modifications do not interfere with other threads.
- Avoiding False Sharing: By allocating separate memory for each thread, you avoid the problem of false sharing since each thread's data is stored in different cache lines.
- Use Private Data for Unique Access: When each thread requires a unique read/write access to data (e.g., a per-thread counter or temporary storage), private data is typically more efficient.
- Avoid Shared Data for Frequently Modified Data: When data is frequently modified by multiple threads, using shared data can result in performance bottlenecks due to increased synchronization overhead and false sharing.