Parallel Merge Sort

A parallel version of the famous merge sort algorithm using `pthread` library

Haris Sohail (21i-0531)

11/02/2024

Parallel and Distributed Computing

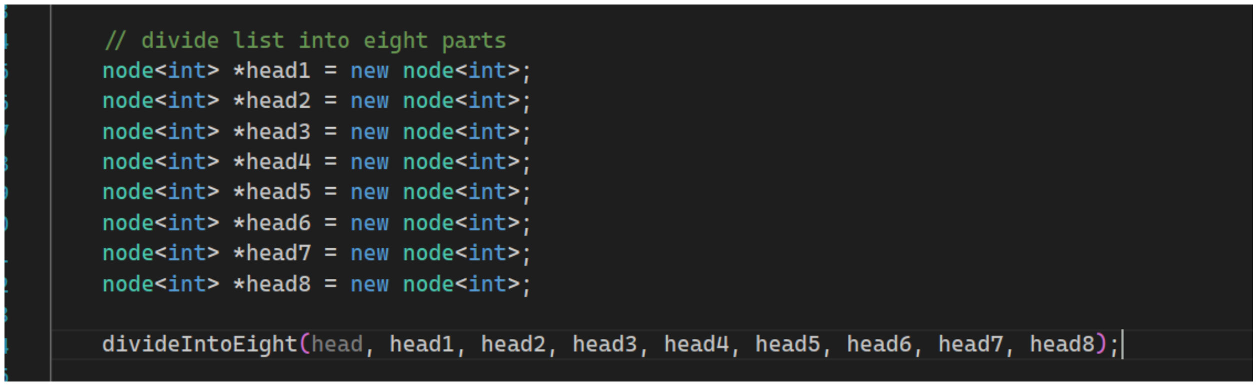
System Architecture

Number of Cores: 4

Mapping Strategy

Mapping Strategy: **Block Domain Decomposition**

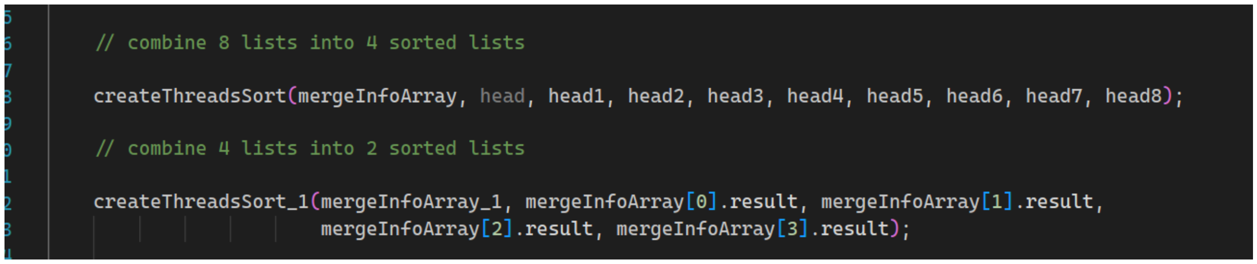
- The linked list is divided into 8 parts



```
// divide list into eight parts
node<int> *head1 = new node<int>;
node<int> *head2 = new node<int>;
node<int> *head3 = new node<int>;
node<int> *head4 = new node<int>;
node<int> *head5 = new node<int>;
node<int> *head6 = new node<int>;
node<int> *head7 = new node<int>;
node<int> *head8 = new node<int>;

divideIntoEight(head, head1, head2, head3, head4, head5, head6, head7, head8);
```

- Combine the 8 lists into 2 sorted lists



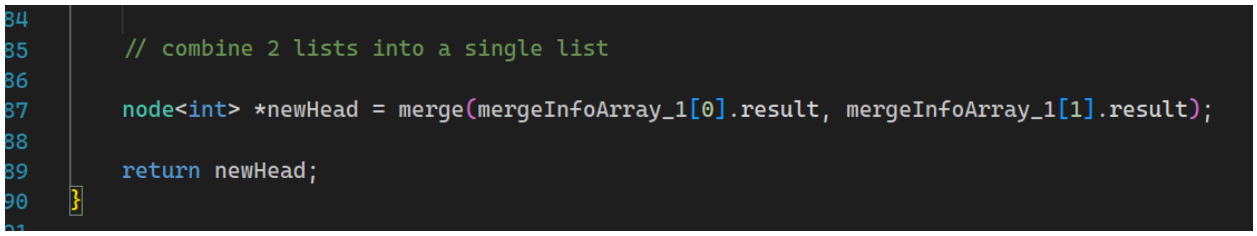
```
// combine 8 lists into 4 sorted lists

createThreadsSort(mergeInfoArray, head, head1, head2, head3, head4, head5, head6, head7, head8);

// combine 4 lists into 2 sorted lists

createThreadsSort_1(mergeInfoArray_1, mergeInfoArray[0].result, mergeInfoArray[1].result,
                    mergeInfoArray[2].result, mergeInfoArray[3].result);
```

- Merge the 2 sorted lists and return the sorted linked list



```
    // combine 2 lists into a single list

    node<int> *newHead = merge(mergeInfoArray_1[0].result, mergeInfoArray_1[1].result);

    return newHead;
}
```

Justification

- **Parallelism:** By dividing the linked list into multiple parts, you create independent subproblems that can be solved concurrently. Each sublist can be sorted independently, allowing for potential parallelization across multiple processing units or threads. This can lead to significant speedup, especially for large lists and systems with multiple processing cores.

- **Load Balancing:** Dividing the linked list into a suitable number of parts (in this case, eight) helps to balance the computational load across processors or threads. If the list is divided unevenly, some processors might finish their work much earlier than others, leading to idle time. By evenly distributing the workload, you can maximize resource utilization and minimize overall execution time.

- **Efficiency:** Merge sort is an efficient sorting algorithm with a time complexity of O(n log n). By parallelizing the merge step, you exploit the inherent efficiency of merge sort while also leveraging parallel processing capabilities to potentially further reduce execution time. This is particularly advantageous for large datasets where traditional sequential sorting algorithms might become impractical.

- **Scalability:** The chosen decomposition strategy allows for scalability as the size of the input data increases. Whether the linked list contains thousands, millions, or even billions of elements, the parallel merge sort can efficiently divide and conquer the problem, adapting to the available resources and maintaining performance.

- **Simplicity:** While more complex parallel sorting algorithms exist, such as parallel quicksort or parallel radix sort, the parallel merge sort with block domain decomposition offers a good balance of simplicity and effectiveness. It is easier to implement and reason about compared to some other parallel sorting algorithms, making it a practical choice in many scenarios.

Execution Time

For varying input sizes this function is used:

```
writeNumbersToFile("./assets/numbers.txt", 100000);
```

Execution Time of serial version:

Input size 1000: 3 milliseconds
Input size 10000: 214 milliseconds
Input size 100000: 20157 milliseconds

Execution Time of parallel version:

Input size 1000: 5 milliseconds
Input size 10000: 36 milliseconds
Input size 100000: 1741 milliseconds

CPU Affinity

Execution Time of parallel version - with affinity:

Input size 100000: 1741 milliseconds

Execution Time of parallel version - without affinity:

Input size 100000: 1956 milliseconds

Execution Time of serial version - without affinity:

Input size 100000: 20157 milliseconds

Actual Speedup

For input size 100000

T_1 = 20157 milliseconds

T4 = 1741 milliseconds

Actual SpeedUp = T1 / T4 = **11.5778**

Speedup (Amdhal's Law)

To calculate ***fp***

Time spent on sequential parts: Reading numbers from the file and writing the result to a file.

Time spent on parallel parts: Creating threads and performing merge sort in parallel.
We'll calculate the proportion of the program that can be parallelized based on these estimates.

Given that we're already measuring the execution time (duration), we can calculate the proportion of parallelization by comparing the time spent on parallel parts with the total execution time.

Let's denote:
Ts as the time spent on sequential parts.
Tp as the time spent on sequential parts.

Then. the proportion of the program that can be parallelized (***fp***) can be estimated as:

Tp / (Tp + Ts)

According to this ***fp*** = 0.4

Therefore, using Amdhal's law, speedup is:

1.428

Implementation of CPU Affinity

CPU Affinity is set using the following function:

```
void setAffinity(pthread_t thread, int coreId)
{
   cpu_set_t cpuset;
   CPU_ZERO(cpuset);
   CPU_SET(coreId, cpuset);

   if (pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset) != 0)
   {
       perror("pthread_setaffinity_np");
   }
}
```

This simple function uses `pthread_setaffinity_np()` call to set the affinity on a per thread basis.

Impact of CPU Affinity

- **Cache Locality:**
  With CPU affinity, threads are bound to specific CPU cores. This can lead to better cache locality because threads tend to reuse data that is already in the cache of the assigned core. As a result, there is less cache thrashing and fewer cache misses, which can significantly improve memory access times.
  Improved cache locality reduces the time spent waiting for data to be fetched from main memory, leading to faster execution times. This is especially beneficial for sorting algorithms like merge sort, which involve frequent data access and manipulation.

- **Reduced Context Switching:**
  CPU affinity reduces the need for frequent context switching between CPU cores. When threads are not bound to specific cores, the operating system may schedule them on different cores over time, resulting in context switches.
  Context switches incur overhead due to saving and restoring thread state, which can impact performance, especially in highly parallel applications. By limiting context switching through CPU affinity, the parallel version can execute more efficiently by maintaining thread state within the same core, avoiding unnecessary overhead.

- **Overall Performance Improvement:**

  The combination of improved cache locality and reduced context switching typically results in better overall performance for the

parallel version with CPU affinity compared to the serial version without affinity.
The observed execution time improvement from 20157 milliseconds (serial without affinity) to 1741 milliseconds (parallel with CPU affinity) for an input size of 100000 indicates a significant performance gain.
This improvement is attributed to the optimized resource utilization and reduced overhead achieved by binding threads to specific CPU cores, thereby maximizing parallel execution efficiency.