Name: Waleed Asif

Roll no: 21i-0537

Section: H

Assignment: 1

Documentation:

Before implementing the parallel version of the sorting algorithm and mapping threads to specific CPU cores, it's crucial to analyze the system's architecture and come up with an effective plan. Here's a proposed mapping strategy along with justifications:  
Mapping Strategy:  
  
Determine the Number of Available CPU Cores: Use system APIs like sysconf(\_SC\_NPROCESSORS\_ONLN) to determine the number of available CPU cores.  
  
Equal Distribution of Workload: Divide the workload evenly among the available CPU cores. Since we have a linked list to sort, we can divide the list into equal-sized chunks, each of which will be processed by a separate thread.  
  
One-to-One Thread-Core Mapping: Map each thread to a specific CPU core to avoid contention and maximize parallelism. This strategy ensures that each core handles a dedicated portion of the workload without interference from other threads.  
  
Affinity Based on Thread ID: Assign CPU cores based on the thread's ID (or index). For example, if there are 8 CPU cores and 16 threads, each thread can be mapped to one of the 8 cores using the formula core\_id = thread\_id % core\_count. This ensures that threads are evenly distributed across available cores.  
  
Justification:  
  
Maximizing CPU Utilization: By utilizing all available CPU cores, we can maximize parallelism and reduce overall processing time.  
Avoiding Contention: Assigning each thread to a specific core reduces cache contention and avoids unnecessary context switching, leading to better performance.  
Load Balancing: Distributing workload evenly ensures that each core has a similar amount of work, preventing situations where some cores are idle while others are overloaded.  
Simplicity: Mapping threads based on their IDs is a straightforward approach that is easy to implement and understand. It ensures a balanced distribution of workload without the need for complex load-balancing algorithms.  
  
Considerations:  
  
Cache Locality: Assigning threads to specific cores can improve cache locality, as each thread consistently accesses the same cache lines, reducing cache misses.  
Reduced Context Switching: Dedicated cores for each thread minimize context switching overhead, as threads don't need to compete for CPU time on shared cores.  
  
By following this mapping strategy, we can effectively utilize the available hardware resources and achieve efficient parallelization of the sorting algorithm. However, it's essential to benchmark and profile the application to fine-tune the mapping strategy based on the specific characteristics of the system and workload.  
  
  
  
In the serial execution scenario, the sorting algorithm processes the entire dataset sequentially, one element at a time. This means that the algorithm cannot take advantage of parallelism inherent in modern multi-core processors. As a result, the execution time tends to be longer compared to parallel execution, especially for large datasets. In your case, with a serial execution time of around 0.6 seconds, it indicates that the algorithm takes this amount of time to sort the dataset without utilizing multiple CPU cores concurrently.  
  
On the other hand, the parallel execution involves dividing the dataset into smaller chunks and sorting each chunk independently using multiple threads running simultaneously on different CPU cores. This allows the algorithm to exploit the parallel processing power of the hardware, leading to faster execution times. However, achieving linear speedup, where doubling the number of threads halves the execution time, is often not feasible due to overheads associated with thread management, resource contention, and diminishing returns in parallel efficiency. In your scenario, with a parallel execution time ranging from 0.4 seconds with 12 to 16 threads, it demonstrates the benefits of parallelism in reducing execution time compared to the serial approach.  
  
To calculate the speedup, we can use the formula:  
  
Speedup=Tserial / Tparallel

where Tserial/Tserial is the execution time of the serial version and Tparallel/Tparallel is the execution time of the parallel version. For your case, the speedup would be:

Speedup = 0.60/0.4 =1.5

Overall, CPU affinity enhances the parallel version's performance by optimizing cache utilization and reducing context switching overhead. However, it's essential to carefully balance thread-to-core mapping to avoid overloading specific cores and maintain load balance across the system. Additionally, the effectiveness of CPU affinity may vary depending on the specific characteristics of the algorithm, hardware architecture, and workload. Experimentation and profiling are crucial to determine the optimal CPU affinity configuration for a given application