Parallel Computing Challenge 1

24.10.2024

Full Name: Nada Elsayed ID: 10998973

—

1. Experimental Setup

1.1 Implementation Parameters

Tested with diff params until found best suited with local machine

- Minimum parallel size threshold: 1024 elements
- Maximum recursion depth: 3 levels
- Cache line alignment: 64 bytes
- Thread allocation: Dynamic based on array size
    - Full thread utilization for arrays > 4096 elements
    - Limited to 2 threads for smaller arrays
- 3 Options to run: sequentially, parallel, both e.g.:
    - Sequential: `./mergesort-co 2000 s`
    - Parallel: `./mergesort-co 200000 p`
    - Both: `./mergesort-co 1000 both`

2. Performance Measurements

2.1 Methodology

- Test Dataset Sizes: [e.g. 10^3, $2x10^5$, $2x10^7$, $2x10^{10}$ elements]
- Multiple runs per configuration
- Timing measured using `gettimeofday`

2.2 Results

elements	Size (MiB)	Sequential Time (sec)	Parallel Time (sec)	Speedup
10^3	0.003815	0.000154	0.000593	0.259696
$2x10^5$	0.762939	0.088044	0.039071	2.253436
$2x10^7$	76.293945	11.112133	4.822240	2.304351
10^8	381.469727	57.291785	26.635021	2.150995

- **First example shows that sequential gives better results for small data than parallel implementation.**

2.3 Performance Analysis

1. **Scalability**
    - Strong scaling observed for large arrays
    - Overhead dominates for small arrays

3. Design Choices

3.1 Parallelization Strategy

1. **Task Parallelism**
    - OpenMP tasks for recursive divide-and-conquer
    - Dynamic thread adjustment based on problem size
    - Task creation limited to depth < 2 to prevent overhead
2. **Optimization Techniques**

- *Based on machine specs*
    - Early sequential cutoff (MIN_PARALLEL_SIZE = 1024)
    - Limited task creation depth (MAX_DEPTH = 3)
    - Adaptive thread count based on input size

3.2 Key Design Decisions

- **Hybrid Approach**
    - Switch to sequential implementation for small arrays (<1024 elements)
    - Reasoning: Reduce overhead for small subproblems

2. Thread Management

```
if (size < MIN_PARALLEL_SIZE * 4) {
    num_threads = std::min(num_threads, 2);
}
```

- Prevents thread overhead for smaller datasets.
- Ensures efficient resource utilization.

3. Task Creation Control

```
#pragma omp task shared(array, tmp) if(depth < 2)
```

- Limits parallel task creation to upper tree levels.
- Balances parallelism and overhead.