Algorithm Engineering

Tasks, Merge Sort, Multithreaded Merging

Mark Blacher

Friedrich Schiller University Jena

Winter Semester of 2022/23

Major OpenMP Constructs We've Covered So Far

- To **create** a **team of threads**

  `#pragma omp parallel`

- To **share work between threads**

  `#pragma omp for`

  `schedule(dynamic), collapse(3), ordered, reduction(+:sum)`

  `#pragma omp single`

- To **prevent conflicts** (prevent races)

  `#pragma omp critical`

  `#pragma omp atomic`

  `#pragma omp barrier`

- **Storage Attributes**

  `private(a)`

  `firstprivate(b)`

  `shared(c)`

OpenMP **Tasks**

- Tasks are **independent units of work**
- Tasks have **potential to parallelize recursive function calls**
- A **queuing system dynamically handles** the **assignment of threads** to the **tasks** () $\Rightarrow$ threads pick up a task from the queue until the queue is empty

Task Example

task_example.cpp

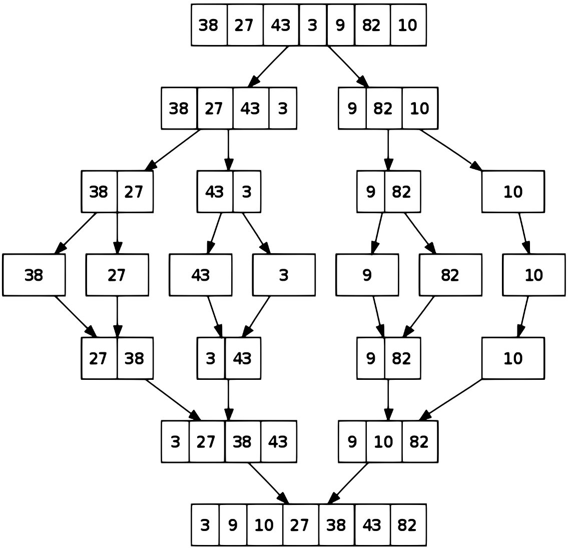
```
// There are two tasks. One prints the word "race ".
// The other one prints "car ". The order in which tasks are
// executed is runtime dependent and may vary across multiple runs.
int main() {
#pragma omp parallel
  { // we need the single construct to guarantee that
    // each task is generated only once
#pragma omp single // one thread generates the tasks
    {
      // Each #pragma omp task directive defines a task
#pragma omp task // define task and put it in the task queue
      { cout << "race "; } // task 1
#pragma omp task // define task and put it in the task queue
      { cout << "car "; } // task 2
    } // end of single region (implicit barrier)
  } // end of parallel region
}
```

Taskwait Construct

taskwait.cpp

```cpp
int main() {
#pragma omp parallel
  { // we need the single construct to guarantee that
    // each task is generated only once
#pragma omp single // one thread generates the tasks
    {
      cout << "A "; // printed before tasks are generated
      // Each #pragma omp task directive defines a task
#pragma omp task // define task and put it in the task queue
      { cout << "race "; } // task 1
#pragma omp task // define task and put it in the task queue
      { cout << "car "; } // task 2
#pragma omp taskwait // wait here until the tasks are completed
      cout << "is fun to watch.\n";
    } // end of single region
  } // end of parallel region
}
```

Merge Sort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Divide and Conquer Example (Merge Sort)

Try to **reduce** the **runtime** of this naive merge sort implementation.

slow_merge_sort.cpp

```cpp
void merge_sort_naive(int *arr, int n) { // slow merge sort
  if (n > 1) { // TODO: use insertion sort for small n
    const int size_a = n / 2;
    const int size_b = n - size_a;
    // TODO: make next recursive call a task
    merge_sort_naive(arr, size_a); // recursive call
    merge_sort_naive(arr + size_a, size_b); // recursive call
    // TODO: here should be a taskwait
    int *c = new int[n]; // TODO: avoid using heap for small n
    merge(arr, arr + size_a, c, size_a, size_b, n);
    memcpy(arr, c, sizeof(int) * n);
    delete[](c);
  }
}
```

Slightly Improved Merge Sort

merge_sort.cpp

```cpp
void merge_sort(int *arr, int n) {
  if (n > 1) {
    if (n < 32) { // insertion sort for n smaller than 32
      insertion_sort(arr, n);
      return;
    }
    const int size_a = n / 2;
    const int size_b = n - size_a;
#pragma omp task if (size_a > 10000) // make task if size_a > 10000, one task is enough
    merge_sort(arr, size_a); // parallel recursive call
    merge_sort(arr + size_a, size_b); // recursive call
#pragma omp taskwait // wait until both subarrays are sorted
    if (n < 8192) { // allocate array on the stack for small n
      // sometimes called _alloca() or _malloca(), or "int c[n]"
      int* c = (int*) alloca(n * sizeof(int));
      merge(arr, arr + size_a, c, size_a, size_b, n);
      memcpy(arr, c, sizeof(int) * n); // copying can be tuned away...
      return;
    }
    int *c = new int[n];
    merge(arr, arr + size_a, c, size_a, size_b, n);
    memcpy(arr, c, sizeof(int) * n); // copying can be tuned away...
    delete[](c);
  }
}

void merge_sort_run(int *arr, int n) {
#pragma omp parallel
#pragma omp single nowait
  merge_sort(arr, n);
}
```

Avoid Copying Data in Merge Sort

merge_sort_no_copying.cpp

```
void merge_sort_helper(int *arr, int *buffer, int n) { // helps to implement no copying strategy
  const int size_a = n / 2;
  const int size_b = n - size_a;
#pragma omp task final(size_a < 10000) // stop creating tasks if size_a < 10000
  merge_sort(arr, buffer, size_a); // parallel recursive call to merge sort
  merge_sort(arr + size_a, buffer + size_a, size_b); // recursive call to merge sort
#pragma omp taskwait // wait until both subarrays are sorted
  merge(arr, arr + size_a, buffer, size_a, size_b, n); // merge from arr into buffer
}

void merge_sort(int *arr, int *buffer, int n) {
  if (n > 1) { // only this function can end a recursion
    if (n < 64) { // insertion sort for n smaller than 64
      insertion_sort(arr, n);
      return;
    }
    const int size_a = n / 2;
    const int size_b = n - size_a;
#pragma omp task final(size_a < 10000) // stop creating tasks if size_a < 10000
    merge_sort_helper(arr, buffer, size_a); // parallel recursive call to helper
    merge_sort_helper(arr + size_a, buffer + size_a, size_b); // recursive call to helper
#pragma omp taskwait // wait until both subarrays are sorted
    merge(buffer, buffer + size_a, arr, size_a, size_b, n); // merge from buffer into arr
  }
}

void merge_sort_run(int *arr, int *buffer, int n) { // buffer is O(n) extra memory
#pragma omp parallel                                // user provides buffer when calling merge_sort_run
#pragma omp single nowait
  merge_sort(arr, buffer, n);
}
```

`if` Clause and `final` Clause

`if(expr)`

If the expression *`expr`* evaluates to true, the construct is executed. Can be applied to `#omp parallel` or `#omp task` to trigger parallel execution. *// simplified explanation*

if_clause.cpp

```
int main() {
  volatile int problem_size = 100;
  // if the problem size is small, don't execute in parallel
#pragma omp parallel if (problem_size > 1000)
  cout << "Hello from thread " << omp_get_thread_num() << endl;
}
```

`final(expr)`

The final clause takes an expression *`expr`*. If it evaluates to true, no more tasks are generated and the code is executed immediately. This is propagated to all the child tasks (once a task is final, all child tasks are final too). `final` can only be applied to `#omp task`.

Storage Attributes Can Also Be Used for **Tasks**

Storage attributes in tasks determine how **variables defined outside the** `#omp task` region **are handled** ("stored") **inside the** `#omp task` region.

storage_attributes_task.cpp

```cpp
int main() {
  int x = 1; // x is shared in a task per default -> outside parallel region
#pragma omp parallel num_threads(4) // begin parallel region
  {
    int y = 7; // y is firstprivate in a task per default -> inside parallel region
#pragma omp single // the code in a single is executed only once
    {
      cout << "x: ";
      for (int i = 0; i < 8; ++i) { // create 8 tasks
#pragma omp task firstprivate(x) shared(y) // overwrite storage attributes
        {
          cout << ++x; // prints always 2 because x is firstprivate in each task
          ++y; // this y comes from the thread that creates the single
          // but y is updated from multiple threads -> possible race condition
        }
      }
    } // implicit barrier of single
#pragma omp critical
    cout << "\ny: " << y; // prints 7 7 7 and (very likely) one 15
  } // end parallel region
}
```

Nested Parallelism

nested_parallelism.cpp

```
int main() {
  omp_set_nested(1); // enable nested parallelism
  int counter = 0; // counter for total threads executed
#pragma omp parallel num_threads(4) // start outer parallel region
  {
    // each thread creates two new threads
#pragma omp parallel num_threads(2) // start inner parallel region
    {
#pragma omp critical
      { // thread id in inner parallel region has nothing
        // to do with the thread id of the outer parallel region
        cout << omp_get_thread_num() << endl; // prints 0 or 1
        ++counter;
      }
    } // end inner parallel region
  } // end outer parallel region
  cout << "counter: " << counter << endl; // prints 8 since 4 * 2 = 8
}
```

libstdc++ Parallel Mode

Converts automatically the **standard** (sequential) `std::...` **algorithms** to the appropriate **parallel equivalents**.

Click here for more information

Compiler flags:

```
-fopenmp -march=native -D_GLIBCXX_PARALLEL
```

Parallel Prefix Sum Computation

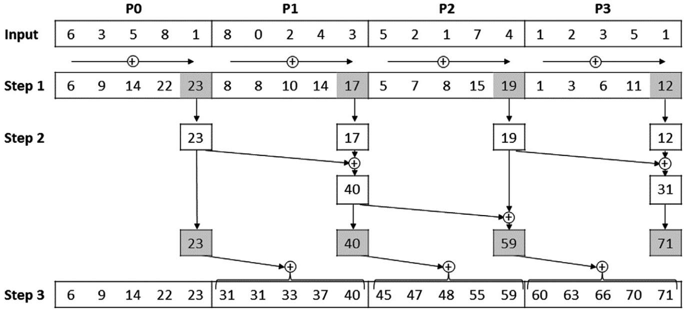


FIGURE 1.9

Parallel prefix computation using four processors where each processor is assigned five elements of the input array. **Step 1:** Local summation within each processor; **Step 2:** Prefix sum computation using only the rightmost value of each local array; **Step 3:** Addition of the value computed in Step 2 from the left neighbor to each local array element.

Parallel Programming Concepts and Practice, p. 14

Idea Behind **Multithreaded Merging**

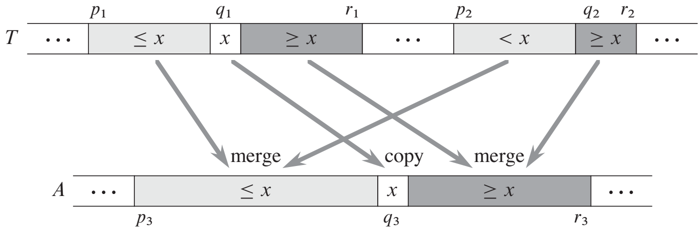


Figure 27.6 The idea behind the multithreaded merging of two sorted subarrays $T[p_1 \mathinner{..} r_1]$ and $T[p_2 \mathinner{..} r_2]$ into the subarray $A[p_3 \mathinner{..} r_3]$. Letting $x = T[q_1]$ be the median of $T[p_1 \mathinner{..} r_1]$ and q_2 be the place in $T[p_2 \mathinner{..} r_2]$ such that x would fall between $T[q_2 - 1]$ and $T[q_2]$, every element in subarrays $T[p_1 \mathinner{..} q_1 - 1]$ and $T[p_2 \mathinner{..} q_2 - 1]$ (lightly shaded) is less than or equal to x, and every element in the subarrays $T[q_1 + 1 \mathinner{..} r_1]$ and $T[q_2 + 1 \mathinner{..} r_2]$ (heavily shaded) is at least x. To merge, we compute the index q_3 where x belongs in $A[p_3 \mathinner{..} r_3]$, copy x into $A[q_3]$, and then recursively merge $T[p_1 \mathinner{..} q_1 - 1]$ with $T[p_2 \mathinner{..} q_2 - 1]$ into $A[p_3 \mathinner{..} q_3 - 1]$ and $T[q_1 + 1 \mathinner{..} r_1]$ with $T[q_2 \mathinner{..} r_2]$ into $A[q_3 + 1 \mathinner{..} r_3]$.

Introduction to Algorithms, 3rd edition, p. 798

Your Paper

- 3 – 4 pages in ACM sigconf format
  https://www.acm.org/publications/proceedings-template
- Example Paper
  http://ceur-ws.org/Vol-2726/paper2.pdf
- Basic LaTeX template
  template_paper.tar.gz
  (use BibTeX **not** Biber to compile bibliography)

Coding Warmup

1. Try to **speed up** the **naive merge sort** implementation on page 7.
2. On page 15 you have become acquainted with **multithreaded merging**. On page 18 is an actual **singlethreaded implementation** of the algorithm. **Parallelize** the singlethreaded implementation with tasks.

Parallelise the Merge Function

merge_single_threaded.cpp

```
    // TODO: parallelize the first recursive call with a task
    merge_dac(t1, t2, p1, q1 - 1, p2, q2 - 1, a, p3, __comp);
    merge_dac(t1, t2, q1 + 1, r1, q2, r2, a, q3 + 1, __comp);
  }
}

template <class _Type, class _Compare>
void parallel_merge(_Type *arr1, _Type *arr2, _Type *out,
        int64_t size1, int64_t size2, int n_threads, _Compare __comp) {
  // TODO: parallelize merge_dac, use n_threads as the amount of threads
  // for the parallel region
  merge_dac(arr1, arr2, 0, size1 - 1, 0, size2 - 1, out, 0, __comp);
}
```

Possible Solution for Parallelising the Merge

merge_multi_threaded.cpp

```
#pragma omp task // make the first recursive call a task
    merge_dac(t1, t2, p1, q1 - 1, p2, q2 - 1, a, p3, __comp);
    merge_dac(t1, t2, q1 + 1, r1, q2, r2, a, q3 + 1, __comp);
  }
}

template <class _Type, class _Compare>
void parallel_merge(_Type *arr1, _Type *arr2, _Type *out,
        int64_t size1, int64_t size2, int n_threads, _Compare __comp) {
#pragma omp parallel num_threads(n_threads)
#pragma omp single nowait
  merge_dac(arr1, arr2, 0, size1 - 1, 0, size2 - 1, out, 0, __comp);
}
```

Exam Assignments

- Explain how **divide and conquer** algorithms can be parallelized with **tasks** in OpenMP.
- Describe some **ways** to **speed up merge sort**.
- What is the idea behind **multithreaded merging**?
- Do the **coding warmup** on **slide 17**.
- Read *What every systems programmer should know about concurrency*.
  https://assets.bitbashing.io/papers/concurrency-primer.pdf
  Discuss **two things** you find particularly interesting.