Parallelism
1202

Lab 3: Divide and conquer parallelism with OpenMP - Sorting

Héctor Ramón Jiménez   Alvaro Espuña Buxó

November 23, 2014
Facultat d'Informàtica de Barcelona

1 Analysis with Tareador

1. **Include the source of the sequential multisort.c code modified with the calls to the Tareador API and the task graph generated.**

   As we can see in `multisort-tareador.c`, in the `main` function we only instrumented the call to `do_sort` with randomly generated data because it's the only one we care. To increase the granularity, and to understand better what's going on with the recursive calls we also instrumented `multisort` with appropriate names (`multisort-i`, `merge-n-i`, `basicsort`). We did a similar thing for merge.

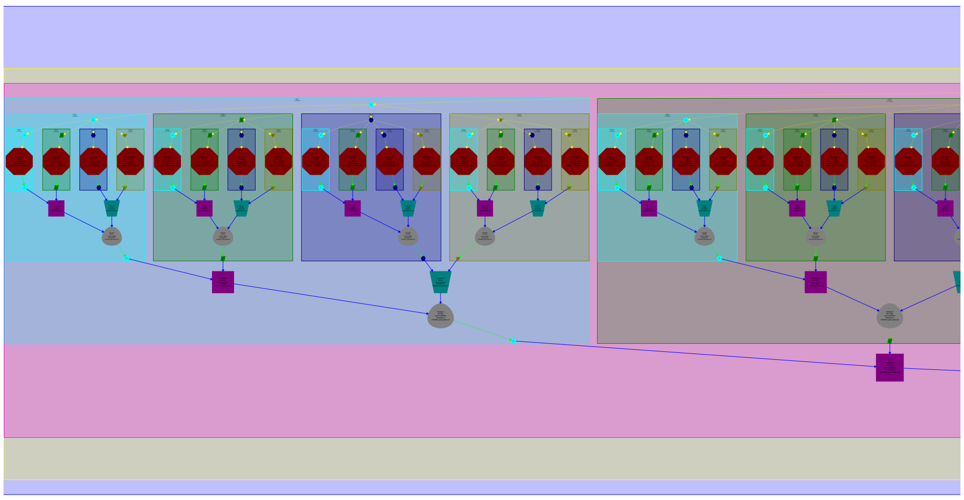

Figure 1: Detail of the output of tareador for multisort-tareador. The full graphic is in `multisort_tareador.pdf`.

   From the figure 1 we can see how the `basicsorts` are fully parallelizable. The two merges of size n/4 for every call to `multisort` are also mutually parallelizable. They only depend on the sorting of the relevant subarray. The call for size n/2 has to wait for them though.

2. **Write a table with the execution time and speed-up predicted by Dimemas (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.**

   In figure 2 we can see that the scalability degrades past 8 threads. This result is expected since due to data dependencies, the program is not fully parallelizable (being the merging the bottleneck). The critical

N	Time (s)	Speedup
1	53.843	-
2	30.379	1.772
4	18.592	2.895
8	12.701	4.239
16	10.091	5.335
32	8.895	6.053
64	8.326	6.467

Figure 2: Table for times and speedup as a function of the number of threads (N)

path for mergesort is `basicsort` → `merge n/4` → `merge n/2`. We can see in the following graphics how the merging is being the bottleneck as we increase the number of threads (clearly noticeable with 32 threads) while the `basicsort`ing (in pink) is clearly fully parallelizable.

Probably it is more scalable than reported by Dimemas, because Tareador is taking into account the initialization of the array, which is sequential.

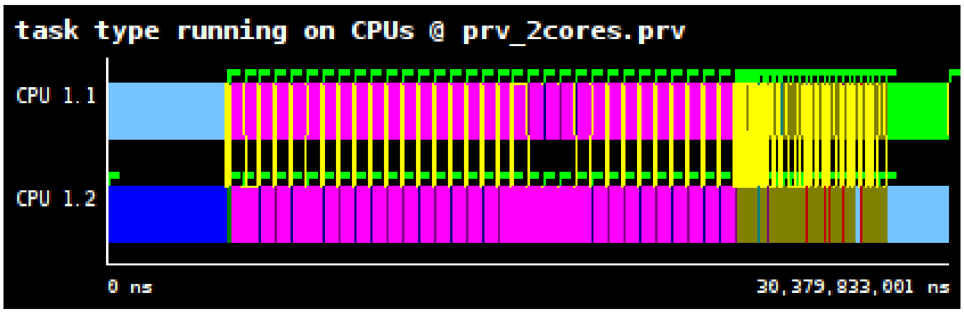

Figure 3: Dimemas simulation with 2 cores

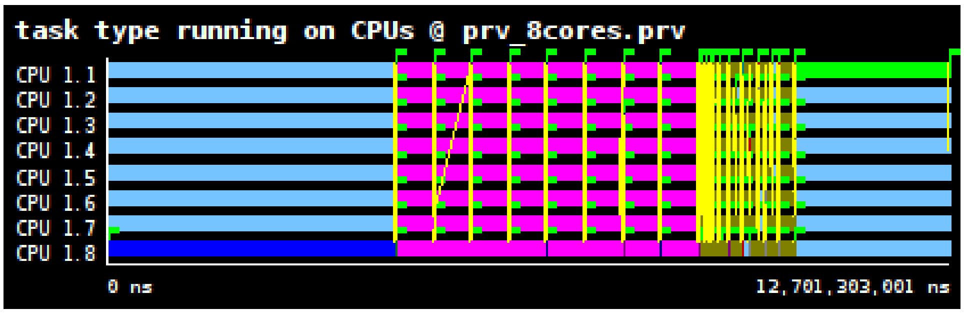

Figure 4: Dimemas simulation with 8 cores

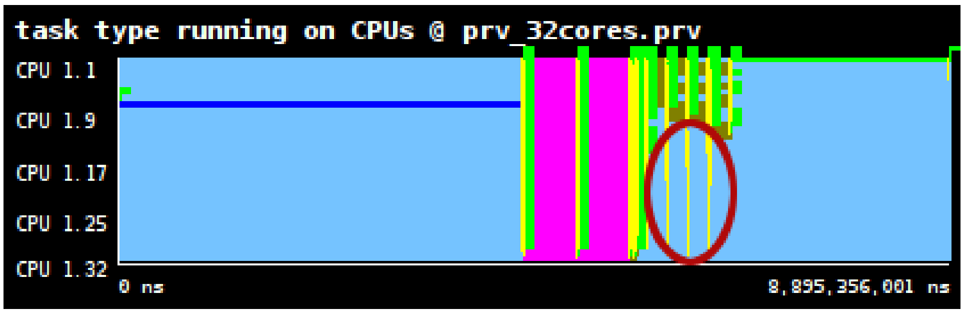

Figure 5: Dimemas simulation with 32 cores. The red *circle* shows how the merging doesn't scale well (although basicsorting does)

2 Parallelization with OpenMP

3. **Briefly describe the two versions (Leaf and Tree) implemented, making references to the source code included in the compressed tar file.**

   The idea in both versions is to create tasks and add them to a pool, so the threads execute them when they can.

2.1 Leaf

The leaf version creates a task for every base case. We need to take care of the syncronization.

2.1.1 merge

We create a task for every `basicmerge`. There's no need of synchronization at this level, both merges are totally parallelizable.

```
void merge(long n, T left[n], T right[n], T result[n*2],
           long start, long length) {
```

```
    if (length < MIN_MERGE_SIZE*2L) {
       // Base case
       #pragma omp task
       basicmerge(n, left, right, result, start, length);
    } else {
       // Recursive decomposition
       merge(n, left, right, result, start, length/2);
       merge(n, left, right, result, start + length/2, length/2);
    }
}
```

2.1.2 multisort

We create a task for every `basicsort`. Here some synchronization is needed. First of all, we cannot start merging until we have the four subparts sorted. But every multisort is completely independent from another. Also, as we saw on figure 1 the merge of size n/2 depends of the previous merges, thus we need to wait after them.

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        multisort(n/4L, &data[0], &tmp[0]);
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        #pragma omp taskwait

        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        #pragma omp taskwait

        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
        #pragma omp task
        basicsort(n, data);
    }
}
```

2.2 Tree

In the tree version, we create a task for every inner node. In this version two more `taskwaits` are needed.

2.2.1 merge

We create a task for every `merge` call. Notice how the `taskwait` here is needed, otherwise, two new `merges` on the same subpart of the vector could be started.

```
void merge(long n, T left[n], T right[n], T result[n*2],
           long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        #pragma omp task
        merge(n, left, right, result, start, length/2);
        #pragma omp task
        merge(n, left, right, result, start + length/2, length/2);
        #pragma omp taskwait
    }
}
```

2.2.2 multisort

We create a task for every `multisort` call. When we are finished sorting, (hence the `taskwait` we can start the merging. Now, we create a task for every call to `merge` with size n/4. As in the leaf case, we need to wait both are merged to start the merging with size n/2. Notice how the taskwait is needed after the last merge. To understand this, assume 4 divisions (1 multisort call). If 1 is done and 0 is still going, a merge of 2 could be started (on the *upper* level), but it could produce an incorrect result.

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp task
        multisort(n/4L, &data[0], &tmp[0]);
        #pragma omp task
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        #pragma omp task
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        #pragma omp task
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        #pragma omp taskwait
```

```
            #pragma omp task
            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
            #pragma omp task
            merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
            #pragma omp taskwait

            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
            #pragma omp taskwait
        } else {
            // Base case
            basicsort(n, data);
        }
    }
```

It's important to note that both versions are probably not technically optimal. When we are done with the two first multisorts, we could start the first merge of size n/4. We avoided this, to benefit from the simplicity of this code.

3 Performance analysis

Write a text, inspired in the one provided below, summarizing the performance results for the two versions (Leaf and Tree) of multisort.

The performance of the two parallelization strategies for multisort has been analyzed on a multiprocessor architecture with 12 Intel cores, distributed in 2 sockets (6 cores each). Using hyperthreading, it has 2 threads per core. It has 3 cache levels: 32 KB L1, 256 KB L2 (both private per core) and a 12 MB L3 shared between cores of the same socket. It has a main memory totalling 23.49 GB.

For all the performance results we have used an input vector of size 8192 Kelements randomly generated by the program itself. The program performs several sort steps in order to verify the influence of the randomness of the data in the input vector.

The recursive depth of the multisort algorithm is controlled with the third and fourht parameters passed to the executable (`argv[2]` and `argv[3]`), that modifies the variables `MIN_SORT_SIZE` and `MIN_MERGE_SIZE`. Using diferent limit sizes in the tree version, we obtain diferent times as shown by figure 6.

The first conclusion of the analysis is that the tree version has better

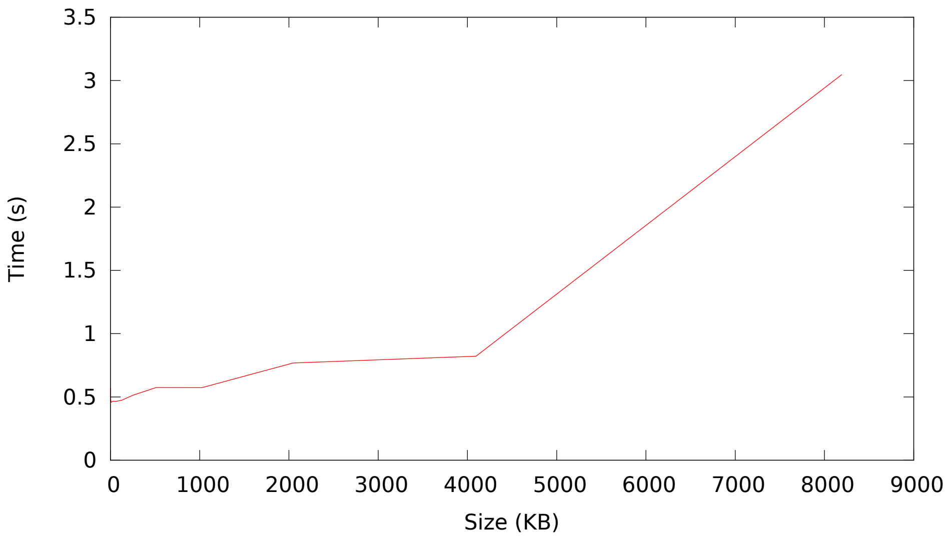

Figure 6: As the size limit grows it takes more time (due to the overhead of the task creation).

scalability than the leaf version. This is due to the fact that the parallelism exploited in the leaf version is limited by the waiting of the merges. The basicsorting tasks will not be really created until the current subtree has been sorted. This critical path is reached much earlier than in the tree version, were there are always tasks to be done.

Figure 7 shows the execution timelines visualized with Paraver, supporting the previous argument. Figure 8 shows the speedup, with respect to the sequential, for different vector sizes (8, 16 and 32 Megaelements), for the two OpenMP versions and for the different invocations of multisort in the main program.

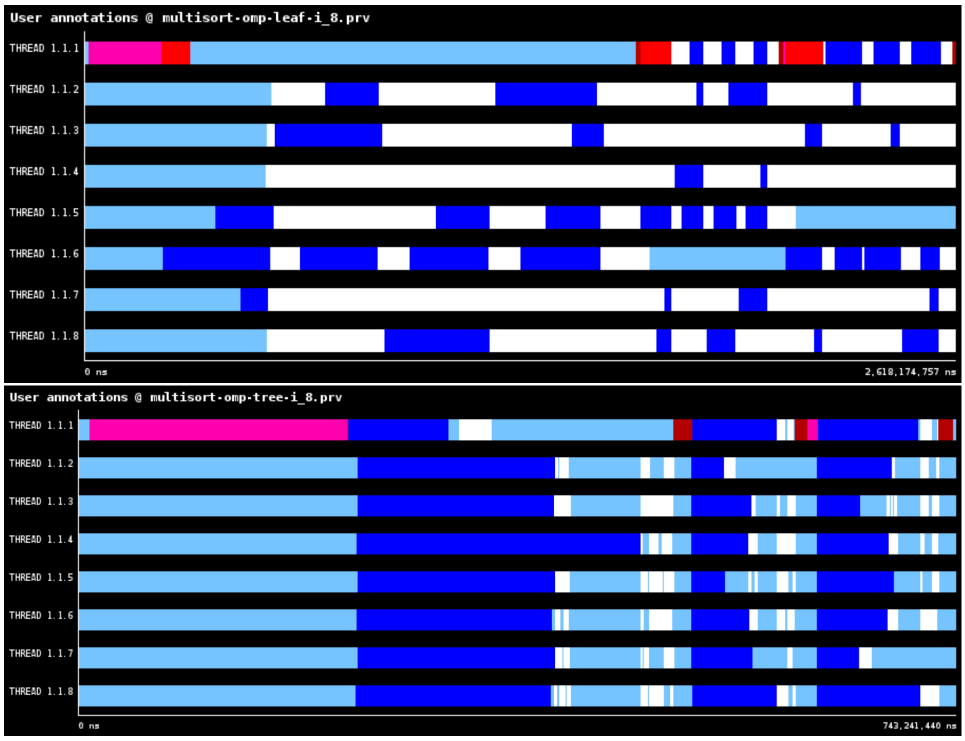

Figure 7: Tree version is faster than leaf version. In white, the time spent on merging.

Leaf			
N	**8192 Ke**	**16384 Ke**	**32768 Ke**
1	1	1	1
2	1.874	1.897	1.826
4	3.741	3.786	3.763
6	3.830	3.798	3.910
8	3.879	3.888	3.990
10	3.771	3.971	4.025
12	3.688	3.937	3.918
Tree			
N	**8192 Ke**	**16384 Ke**	**32768 Ke**
1	1	1	1
2	1.935	1.938	1.925
4	3.768	3.757	3.800
6	3.517	4.061	4.414
8	5.081	5.174	6.031
10	6.150	5.224	7.145
12	6.577	7.088	8.052

Figure 8: We can see that tree scales better than leaf. Even better with bigger sizes.