Parallel Programming SS21 Final Project

Project-03: Kd-tree

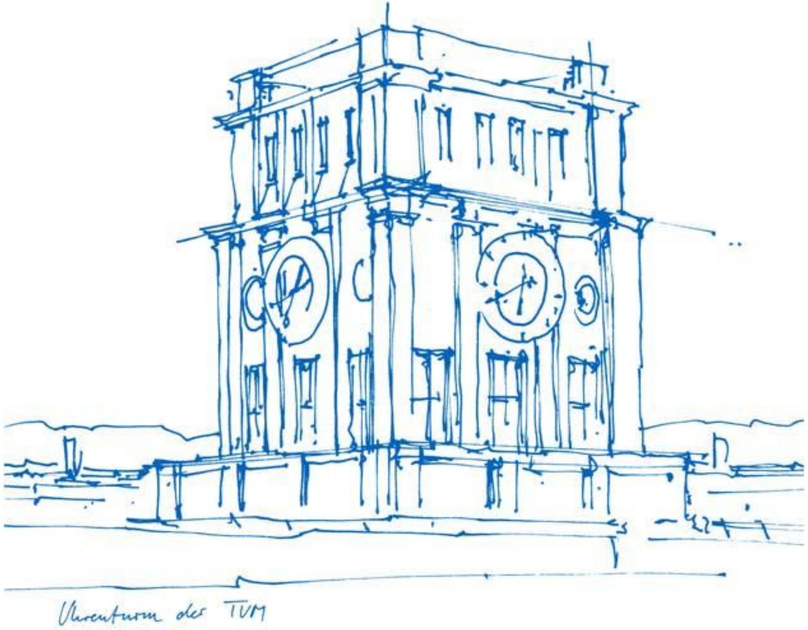
Group: 316

13.07.2021

1- Ertugrul Aypek

2- Amr Amin Amin Elsharkawy

3- Zeynep Erdoğan



Sequential code analysis - Profiling (perf/gprof)

Profiling: "Bottlenecks"

1. distance_squared() : ~53%
2. compare() : ~ 29%

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
Samples: 19K of event 'cycles', Event count (approx.): 15262521850
Overhead  Command     Shared Object        Symbol
  52,56%  sequential  sequential           [.] Point::distance_squared
  28,84%  sequential  sequential           [.] Point::compare
   5,49%  sequential  sequential           [.] std::__introsort_loop<Point**, long, __gnu_
   5,47%  sequential  sequential           [.] Utility::generate_problem
   1,69%  sequential  sequential           [.] build_tree_rec
   1,48%  sequential  sequential           [.] nearest
   0,80%  sequential  libc-2.31.so         [.] cfree@GLIBC_2.2.5
   0,36%  sequential  sequential           [.] std::__insertion_sort<Point**, __gnu_cxx::_
   0,33%  sequential  [kernel.kallsyms]    [k] prepare_exit_to_usermode
```

Call graph:

- Shows that almost all of the time spent in nearest() is actually spent in distance_squared() function
- Similarly for build_tree_rec() and compare()

```
                         Call graph

granularity: each sample hit covers 2 byte(s) for 0.14% of 6.97 seconds

index % time    self  children    called     name
                             9999980             nearest(Node*, Point*, int, Node*, float&) [1]
[1]     50.1    0.09    3.40       0+9999980 nearest(Node*, Point*, int, Node*, float&) [1]
                3.38    0.00 9999960/9999970     Point::distance squared(Point&, Point&) [2]
                0.02    0.00 9999960/9999960     Point::distance squared(Point&) [8]
                             9999980             nearest(Node*, Point*, int, Node*, float&) [1]
```

Sequential code analysis - Profiling (perf)

<u>Profiling: “Cache behavior”</u>

1. L1 cache misses: ~5.3%
2. L3 cache misses: : ~ 44.0%

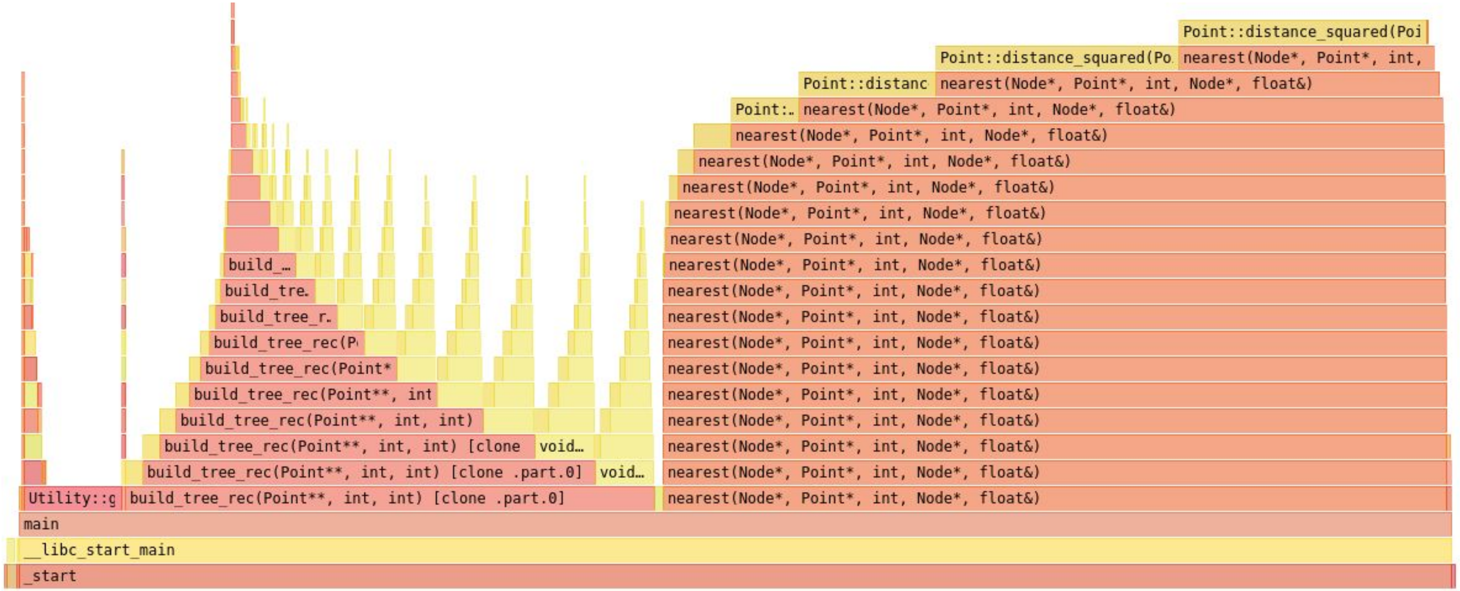
```
Performance counter stats for './sequential 5 128 500000':

          7.550,18 msec task-clock                #    0,870 CPUs utilized
               622      context-switches          #    0,082 K/sec
                 1      cpu-migrations            #    0,000 K/sec
            71.425      page-faults               #    0,009 M/sec
    23.698.599.445      cycles                    #    3,139 GHz                      (50,03%)
    17.526.501.029      instructions              #    0,74  insn per cycle           (62,56%)
     2.526.429.236      branches                  #  334,618 M/sec                    (62,56%)
        42.278.237      branch-misses             #    1,67% of all branches          (62,56%)
     5.472.726.273      L1-dcache-loads           #  724,847 M/sec                    (62,52%)
       290.958.931      L1-dcache-load-misses     #    5,32% of all L1-dcache hits    (62,45%)
       152.731.876      LLC-loads                 #   20,229 M/sec                    (49,92%)
        66.876.846      LLC-load-misses           #   43,79% of all LL-cache hits     (49,96%)

       8,680257944 seconds time elapsed

       7,483315000 seconds user
       0,063891000 seconds sys
```

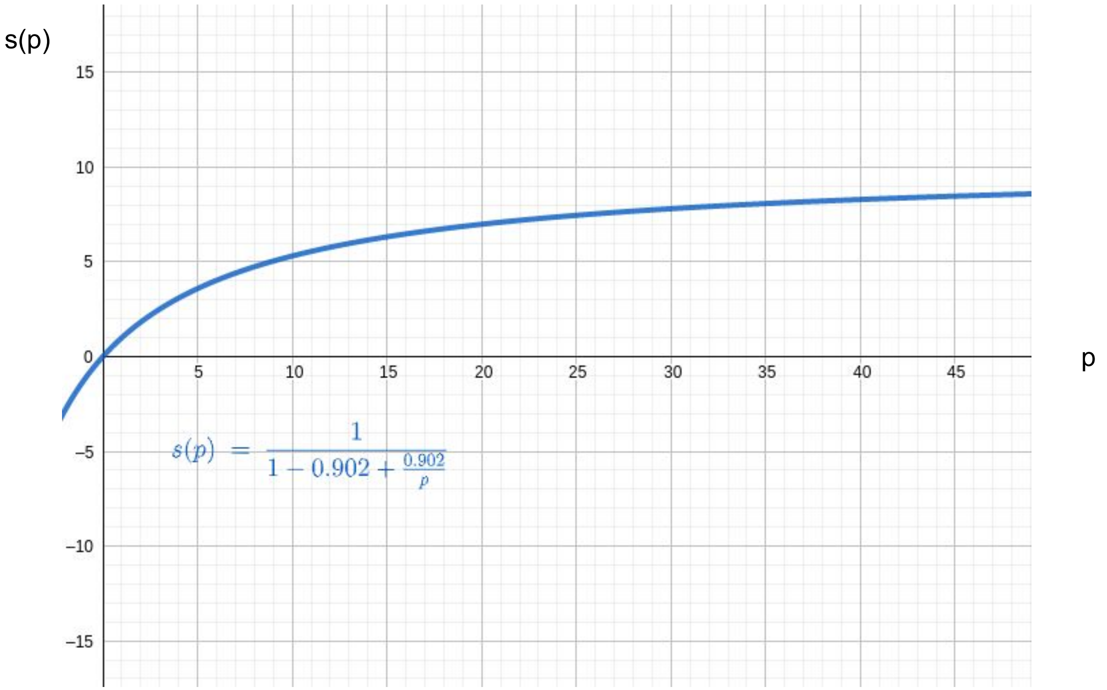
Sequential code analysis - Amdahl's law



Captured with "perf record --call-graph dwarf ./sequential 5 128 500000", visualized using hotspot

nearest fraction : 53,4%
build_tree_rec fraction : 36,8%
Utility::generateProblem fraction : 6,69%

Sequential code analysis - Amdahl's law



parallel fraction : **90,2%** (nearest + build_tree)
maximum speed-up : **10,2041** = 1 / (1 - 0,902 + 0)

OpenMP - Parallelized implementation and approach

1 - Parallelize the queries:

- #pragma omp parallel for Vs. #pragma omp single + tasks -> Performance is almost the same.
- New data structure used to store results of each query.

2 - Parallelize the tree build:

- Sections Vs. Tasks -> Tasks outdid sections (better load balance).

3 - Parallelize the Nearest search:

- Sections Vs. Tasks -> Performance is almost the same.
- In both approaches it was critical to limit the recursive depth at which we either create new task or execute sections in parallel.
- New data structure used to avoid redundant computations.

4 - Any attempt to parallelize the distance function computations results in additional overhead -> Larger runtime (I believe compiler optimization already done enough).

5 - Any attempt to parallelize to points initialization for loop results in additional overhead.

OpenMP - Intermediate Vs. Final Implementation

- In all implementations, either sequential, intermediate results of OMP or final result of OMP, the distance and compare/sort functions are the bottlenecks of the algorithm. The distance and compare functions' computation are simple yet are called many many times.
- So the strategy is to try to utilize the available **12 threads** to reach maximum speedup. We need to find the sweet spot after which thread management overhead comes into play and takes away some of the parallelism performance.
- perf : (**Intermediate result speedup ~ 2.5**)

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

Samples: 18K of event 'cycles', Event count (approx.): 13488660041
Overhead  Command  Shared Object         Symbol
  46,66%  omp      omp                   [.] Point::distance_squared
  25,36%  omp      omp                   [.] std::__introsort_loop<Point**, long, __gn
   6,32%  omp      omp                   [.] build_tree_rec
   6,10%  omp      omp                   [.] Utility::generate_problem
   2,83%  omp      omp                   [.] nearest
```

OpenMP - Intermediate Vs. Final Implementation

Speedup ~ 2.5

build_tree_rec()

```
Node* left_node;
Node* right_node;

#pragma omp parallel sections
{
    // left subtree
    #pragma omp section
    left_node = build_tree_rec(left_points, num_points_left, depth + 1);


    // right subtree
    #pragma omp section
    right_node = build_tree_rec(right_points, num_points_right, depth + 1);

}
```

Load imbalance

#	PID	TID	cycles
	10733	10744	649013465
	10733	10742	803513674
	10733	10733	2841343785
	10733	10743	753392959
	10733	10741	4187066911
	10733	10750	208727
	10733	10749	787048697
	10733	10751	38187
	10733	10745	168879229
	10733	10747	387112845
	10733	10746	760360610

(END)

OpenMP - Intermediate Vs. Final Implementation

Speedup ~ 4.0

```
Node* build_tree(Point** point_list, int num_nodes)
{
    Node* node;
    #pragma omp parallel
    {
        #pragma omp single
        node = build_tree_rec(point_list, num_nodes, 0);
    };
    return node;
}
```

```
//  Let;s build the tree in parallel by creating a task for each branch
Node* left_node;
Node* right_node;
// left subtree : create an omp task
#pragma omp task shared(left_node) if(depth < 12)
    left_node = build_tree_rec(left_points, num_points_left, depth + 1);

// right subtree : create an omp task
#pragma omp task shared(right_node) if(depth < 12)
    right_node = build_tree_rec(right_points, num_points_right, depth + 1);

// wait for the tasks to be complete before returning
#pragma omp taskwait
```

Better load balancing

# PID	TID	cycles
80074	80086	541434555
80074	80081	923136401
80074	80077	571623722
80074	80074	830779230
80074	80080	382192731
80074	80082	5988514
80074	80083	705686988
80074	80076	827197955
80074	80085	508500729
80074	80084	984401112
80074	80078	837088202

(END)

OpenMP - Final Implementation improvements and new speed-up

The rest of code parallelism:

1. Parallelism of queries for loop.
2. Parallelism of Nearest search.

```
// for each query, find nearest neighbor
Node* res[num_queries]; // new data structure to parallelize the queries
Point queries[num_queries]; // new data structure to parallelize the queries

#pragma omp parallel for
    for(int q = 0; q < num_queries; ++q)
    {
        float* x_query = x + (num_points + q) * dim;
        Point query(dim, num_points + q, x_query);
        queries[q] = query;

        res[q] = nearest_neighbor(tree, &query);

    }
```

```
if (depth<1)
{
    // Here let's check both sides anyway in parallel.
    #pragma omp parallel sections shared(The_best)
    {
        #pragma omp section
        {
            nearest(visit_branch, query, depth + 1, The_best);
        }
        #pragma omp section
        {
            nearest(other_branch, query, depth + 1,  The_best);
        }
    }
}
else
{
    // After we exceed a specific depth, let's continue in a sequential way,
    // to avoid the overhead created by the parallelism
    nearest(visit_branch, query, depth + 1, The_best);

    // check the other side if the ball intesects with the hyperplane.
    if (d_axis_squared < The_best->best_global_dist)
    {
        nearest(other_branch, query, depth + 1, The_best);
    }

}
```

Maximum speedup achieved:

- Max. speedup achieved: ~4.0
- Expected speedup by Amdahl's analysis: ~6.0

OpenMP - Speedup Limitation

Speedup limitations:

1. High rate of cache misses (L1/L3).
2. Thread management overhead.
   - Limited amount of work per thread. As we increase parallelism, less and less work done by each thread, till the moment when threading overhead deteriorates the performance.
3. Synchronization overhead.
4. Data locality in NUMA system.

```
Performance counter stats for './omp 5 128 500000':

         6.046,97 msec task-clock           #    2,906 CPUs utilized
              994      context-switches     #    0,164 K/sec
               55      cpu-migrations       #    0,009 K/sec
           71.512      page-faults          #    0,012 M/sec
   16.337.177.143      cycles               #    2,702 GHz                      (49,29%)
    7.310.530.173      instructions         #    0,45  insn per cycle           (61,65%)
      918.825.004      branches             #  151,948 M/sec                    (62,16%)
       46.639.544      branch-misses        #    5,08% of all branches          (62,57%)
    1.542.285.642      L1-dcache-loads      #  255,051 M/sec                    (62,93%)
      345.510.977      L1-dcache-load-misses #  22,40% of all L1-dcache hits    (63,20%)
      171.082.268      LLC-loads            #   28,292 M/sec                    (50,05%)
       71.801.414      LLC-load-misses      #   41,97% of all LL-cache hits     (49,79%)

      2,080869403 seconds time elapsed

      5,945683000 seconds user
      0,103819000 seconds sys
```

```
# Overhead  Command  Shared Object
# ........  .......  ..................
#
    94.37%  omp      omp
     3.32%  omp      [kernel.kallsyms]
     1.88%  omp      libc-2.31.so
     0.33%  omp      libgomp.so.1.0.0
     0.08%  omp      libstdc++.so.6.0.28
     0.02%  omp      ld-2.31.so
     0.00%  perf     [kernel.kallsyms]
     0.00%  omp      [unknown]
```

```
# Overhead  Command     Shared Object
# ........  ..........  ..................
#
    96.48%  sequential  sequential
     2.18%  sequential  [kernel.kallsyms]
     1.24%  sequential  libc-2.31.so
     0.07%  sequential  libstdc++.so.6.0.28
     0.02%  sequential  ld-2.31.so
     0.00%  perf        [kernel.kallsyms]
```

MPI - Parallelized implementation and approach

- Only master and 10 workers (each runs a different query) approach
  - 11 MPI processes in total
  - The other 5 MPI processes exited immediately
  - Whole tree is built by all the 10 workers

```
if (rank == 0) // master
{
    for (int q=0; q<num_queries; q++) {
        float result;
        float *x_query = x + (num_points + q) * dim;
        Point query(dim,  ID: num_points + q, x_query);
        MPI_Recv(&result, 1, MPI_FLOAT, q+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        Utility::print_result_line(query.ID, result);
    }
    std::cout << "DONE" << std::endl;
}
else if(rank >= 1 && rank <= num_queries) // worker
{
    int q = rank - 1;
    float *x_query = x + (num_points + q) * dim;
    Point query(dim,  ID: num_points + q, x_query);
    Node *res = nearest_neighbor(tree, &query);
    float min_distance = query.distance( &: *res->point);
    MPI_Send(&min_distance, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
}
```

MPI - Intermediate Speed-up results, profiling

```
+   88,82%   0,03%  mpi   mpi                  [.] main
+   88,82%   0,00%  mpi   mpi                  [.] _start
+   88,82%   0,00%  mpi   libc-2.31.so         [.] __libc_start_main
+   33,71%   0,00%  mpi   libmpich.so.12.1.8   [.] MPI_Finalize (inlined)
+   28,44%   0,67%  mpi   mpi                  [.] build_tree_rec
+   28,40%   0,00%  mpi   mpi                  [.] build_tree (inlined)
+   28,25%   0,00%  mpi   mpi                  [.] std::sort<Point**, std::_Bind<bool (*(std::_Pl
+   28,25%   0,00%  mpi   mpi                  [.] std::__sort<Point**, __gnu_cxx::__ops::_Iter_c
+   25,70%  25,57%  mpi   mpi                  [.] Point::compare
+   20,84%   1,95%  mpi   mpi                  [.] std::__introsort_loop<Point**, long, __gnu_cxx
+   20,46%   0,00%  mpi   mpi                  [.] std::__unguarded_partition_pivot<Point**, __gn
+   20,31%   0,00%  mpi   mpi                  [.] std::__unguarded_partition<Point**, __gnu_cxx:
+   19,40%   0,00%  mpi   mpi                  [.] __gnu_cxx::__ops::_Iter_comp_iter<std::_Bind<b
+   19,40%   0,00%  mpi   mpi                  [.] std::_Bind<bool (*(std::_Placeholder<1>, std::
+   19,40%   0,00%  mpi   mpi                  [.] std::_Bind<bool (*(std::_Placeholder<1>, std::
+   19,40%   0,00%  mpi   mpi                  [.] std::__invoke<bool (*&)(Point*, Point*, int),
+   19,40%   0,00%  mpi   mpi                  [.] std::__invoke_impl<bool, bool (*&)(Point*, Poi
+   13,85%  11,29%  mpi   mpi                  [.] Utility::generate_problem
+    8,55%   0,00%  mpi   mpi                  [.] std::uniform_real_distribution<float>::operato
+    8,55%   0,00%  mpi   mpi                  [.] std::__detail::_Adaptor<std::mersenne_twister_
+    8,55%   0,00%  mpi   mpi                  [.] std::generate_canonical<float, 24ul, std::mers
+    7,53%   0,00%  mpi   mpi                  [.] std::__final_insertion_sort<Point**, __gnu_cxx
+    7,31%   0,00%  mpi   mpi                  [.] std::__unguarded_insertion_sort<Point**, __gnu
+    7,23%   0,00%  mpi   libmpich.so.12.1.8   [.] PMPI_Init
+    7,08%   0,00%  mpi   mpi                  [.] std::__unguarded_linear_insert<Point**, __gnu_
+    6,86%   0,00%  mpi   mpi                  [.] __gnu_cxx::__ops::_Val_comp_iter<std::_Bind<bo
+    6,86%   0,00%  mpi   mpi                  [.] std::_Bind<bool (*(std::_Placeholder<1>, std::
+    6,86%   0,00%  mpi   mpi                  [.] std::_Bind<bool (*(std::_Placeholder<1>, std::
+    6,86%   0,00%  mpi   mpi                  [.] std::__invoke<bool (*&)(Point*, Point*, int),
+    6,86%   0,00%  mpi   mpi                  [.] std::__invoke_impl<bool, bool (*&)(Point*, Poi
+    6,26%   0,04%  mpi   [kernel.kallsyms]    [k] entry_SYSCALL_64_after_hwframe
+    6,18%   1,21%  mpi   [kernel.kallsyms]    [k] do_syscall_64
+    5,02%   0,00%  mpi   libmpich.so.12.1.8   [.] 0x00007f5675c1dc17
+    5,02%   0,00%  mpi   libmpich.so.12.1.8   [.] 0x00007f5675be32dc
+    4,92%   0,00%  mpi   libmpich.so.12.1.8   [.] 0x00007f9f8182ac17
+    4,92%   0,00%  mpi   libmpich.so.12.1.8   [.] 0x00007f9f817f02dc
+    4,84%   0,00%  mpi   libmpich.so.12.1.8   [.] 0x00007f4c35e93c17
+    4,84%   0,00%  mpi   libmpich.so.12.1.8   [.] 0x00007f4c35e592dc
+    4,64%   0,00%  mpi   libmpich.so.12.1.8   [.] 0x00007fc960180c17
+    4,64%   0,00%  mpi   libmpich.so.12.1.8   [.] 0x00007fc9601462dc
+    4,33%   0,00%  mpi   libmpich.so.12.1.8   [.] 0x00007f52ca01ec17
+    4,33%   0,00%  mpi   libmpich.so.12.1.8   [.] 0x00007f52c9fe42dc
+    4,21%   4,21%  mpi   libmpich.so.12.1.8   [.] 0x00000000001b0e88
+    4,18%   0,00%  mpi   mpi                  [.] std::mersenne_twister_engine<unsigned long, 32
+    3,17%   0,10%  mpi   mpi                  [.] nearest
```

Speed-up: ~1,6

Bottlenecks:

1. Using only 10 workers
2. Build tree vs nearest (28,44% vs 3,17%)
3. Generate problem (13,85%)

Further steps:

- Parallelizing query loop is insufficient
- Need to find another method

MPI - Final Implementation improvements and new speed-up

Time to decompose the tree into workers !

Worker routine

```
int numPointsOfThisWorker = end - start;
Point** points = (Point**)calloc(numPointsOfThisWorker, sizeof(Point*));
for (int n = start; n < end; ++n) {...}

// each worker builds a tree of a specific part of points
Node* tree = build_tree(points, numPointsOfThisWorker);

// each worker runs all the queries on their local trees
for (int q = 0; q < num_queries; ++q) {
    float *x_query = x + (num_points + q) * dim;
    Point query(dim, ID: num_points + q, x_query);
    Node *res = nearest_neighbor(tree, &query);
    float min_distance = query.distance( &: *res->point);

    // each worker sends the min_distance calculated on their local trees.
    // the message is tagged with q (current query_number)
    MPI_Send(&min_distance, 1, MPI_FLOAT, 0, q, MPI_COMM_WORLD);
}
```

Master routine

```
// master expects numTotalMessages from workers
int numTotalMessages = num_queries * (num_procs - 1);

for (int q=0; q < numTotalMessages; q++) {
    float currentDistanceResult;
    MPI_Status status;
    MPI_Recv(&currentDistanceResult, 1, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    // worker sends the query number in MPI_TAG
    int senderTag = status.MPI_TAG;

    // update global_min_distance if local_min_distance is less than global
    if (currentDistanceResult < minDistanceArray[senderTag]) {
        minDistanceArray[senderTag] = currentDistanceResult;
    }
}
```

MPI - Final Implementation improvements and new speed-up

```
+   91,61%   0,00%  mpi    mpi                 [.] _start
+   91,61%   0,00%  mpi    libc-2.31.so        [.] __libc_start_main
+   91,60%   0,01%  mpi    mpi                 [.] main
+   35,64%  24,25%  mpi    mpi                 [.] Utility::generate_problem
+   18,39%   0,00%  mpi    libmpich.so.12.1.8  [.] MPI_Finalize (inlined)
+   18,30%   0,00%  mpi    mpi                 [.] std::uniform_real_distribution<float>::operator
+   18,30%   0,00%  mpi    mpi                 [.] std::__detail::_Adaptor<std::mersenne_twister_e
+   18,30%   0,00%  mpi    mpi                 [.] std::generate_canonical<float, 24ul, std::merse
+   14,55%   0,52%  mpi    mpi                 [.] nearest
+   14,47%   0,00%  mpi    mpi                 [.] nearest_neighbor (inlined)
+   14,01%  13,90%  mpi    mpi                 [.] Point::distance_squared
+   13,09%   0,00%  mpi    libmpich.so.12.1.8  [.] PMPI_Init
+   11,58%   0,05%  mpi    [kernel.kallsyms]   [k] asm_exc_page_fault
+   10,43%   0,03%  mpi    [kernel.kallsyms]   [k] exc_page_fault
+    9,46%   0,09%  mpi    [kernel.kallsyms]   [k] do_user_addr_fault
+    9,37%   0,03%  mpi    [kernel.kallsyms]   [k] entry_SYSCALL_64_after_hwframe
+    9,31%   0,72%  mpi    [kernel.kallsyms]   [k] do_syscall_64
+    9,23%   0,10%  mpi    [kernel.kallsyms]   [k] handle_mm_fault
+    9,21%   0,00%  mpi    mpi                 [.] std::mersenne_twister_engine<unsigned long, 32u
+    9,06%   0,22%  mpi    [kernel.kallsyms]   [k] __handle_mm_fault
+    8,57%   0,08%  mpi    [kernel.kallsyms]   [k] do_anonymous_page
+    7,29%   0,09%  mpi    [kernel.kallsyms]   [k] __alloc_pages_nodemask
+    7,25%   0,04%  mpi    [kernel.kallsyms]   [k] alloc_pages_vma
+    6,06%   0,03%  mpi    [kernel.kallsyms]   [k] __alloc_pages_slowpath.constprop.0
+    5,76%   5,76%  mpi    libmpich.so.12.1.8  [.] 0x000000000001c6add
+    5,62%   0,00%  mpi    [kernel.kallsyms]   [k] try_to_free_pages
+    5,59%   0,00%  mpi    [kernel.kallsyms]   [k] do_try_to_free_pages
+    5,58%   0,02%  mpi    [kernel.kallsyms]   [k] shrink_node
+    5,30%   5,30%  mpi    [kernel.kallsyms]   [k] native_queued_spin_lock_slowpath
+    4,95%   0,04%  mpi    [kernel.kallsyms]   [k] _raw_spin_lock_irq
+    4,80%   0,00%  mpi    libmpich.so.12.1.8  [.] PMPI_Recv
+    4,80%   0,00%  mpi    libmpich.so.12.1.8  [.] 0x00007fa30080de72
+    4,54%   0,00%  mpi    mpi                 [.] std::mersenne_twister_engine<unsigned long, 32u
+    4,37%   0,00%  mpi    libc-2.31.so        [.] __GI___libc_read (inlined)
+    4,36%   0,00%  mpi    [kernel.kallsyms]   [k] __x64_sys_read
+    4,36%   0,00%  mpi    [kernel.kallsyms]   [k] ksys_read
+    4,36%   0,00%  mpi    [kernel.kallsyms]   [k] vfs_read
+    4,33%   0,00%  mpi    [kernel.kallsyms]   [k] new_sync_read
+    4,32%   0,00%  mpi    [kernel.kallsyms]   [k] kernfs_fop_read_iter
+    4,29%   0,00%  mpi    [kernel.kallsyms]   [k] sysfs_kf_bin_read
+    4,29%   0,00%  mpi    [kernel.kallsyms]   [k] pci_read_config
+    4,24%   0,00%  mpi    [kernel.kallsyms]   [k] pci_user_read_config_dword
+    4,00%   0,26%  mpi    mpi                 [.] build_tree_rec
+    3,97%   0,00%  mpi    mpi                 [.] build_tree (inlined)
+    3,83%   0,00%  mpi    mpi                 [.] std::sort<Point**, std::_Bind<bool (*(std::_Pla
```

Speed-up: ~6,64

Benefits of approach:

- Build tree overhead is reduced (28,44% -> 4,00%)
- Due to data locality, cache misses reduced by ~8,2 times (2.989.045.314 -> 363.175.270)

Downsides of approach:

- Each worker sends n_queries messages, instead of 1 message.

Comparison to Amdahl's Law:

- s(16) ≈ 6,47

Hybrid - Parallelized implementation and approach

- Different from MPI part, number of MPI processes reduced from 16 to 4. One master, three workers

- Threading came to the action (thread safe)
  - Let each MPI processes have 12 threads
  - (3 workers) x (12 threads/worker) = 36 worker threads in total

- Approach: Keep the MPI part as it is,
  using OpenMP, parallelise build_tree and nearest routines in each MPI process
  - Tree is divided into three equal parts.

Hybrid - Final Performance Results

```
+   51,95%   0,00%  hybrid   libpthread-2.31.so   [.] start_thread
+   51,40%   0,00%  hybrid   libc-2.31.so         [.] __GI___clone (inlined)
+   28,87%   0,64%  hybrid   libgomp.so.1.0.0     [.] GOMP_task
+   26,33%   0,86%  hybrid   hybrid               [.] build_tree_rec
+   25,16%   0,02%  hybrid   hybrid               [.] main
+   25,06%   0,00%  hybrid   libc-2.31.so         [.] __libc_start_main
+   24,99%   0,00%  hybrid   hybrid               [.] _start
+   24,11%   1,41%  hybrid   hybrid               [.] nearest
+   24,08%   0,58%  hybrid   hybrid               [.] nearest
+   21,55%  21,40%  hybrid   hybrid               [.] Point::distance_squared
+   20,82%   0,00%  hybrid   hybrid               [.] std::sort<Point**, std::_Bind<bool (*(std::_Placeho
+   20,82%   0,00%  hybrid   hybrid               [.] std::__sort<Point**, __gnu_cxx::__ops::_Iter_comp_i
+   18,04%  18,00%  hybrid   hybrid               [.] Point::compare
+   18,02%   0,00%  hybrid   hybrid               [.] build_tree_rec
+   16,48%   0,00%  hybrid   [unknown]            [.] 0xffffffffffffffff
+   14,70%   0,03%  hybrid   libgomp.so.1.0.0     [.] GOMP_taskwait
+   14,69%   2,03%  hybrid   hybrid               [.] std::__introsort_loop<Point**, long, __gnu_cxx::__c
```

Speed-up ~6,38

Comparison to Amdahl's Law:

- s(36) ≈ 8,13

```
// left subtree
#pragma omp task shared(left_node) if(depth < 5)
    left_node = build_tree_rec(left_points, num_points_left, depth: depth + 1);

// right subtree
#pragma omp task shared(right_node) if(depth < 5)
    right_node = build_tree_rec(right_points, num_points_right, depth: depth + 1);

#pragma omp taskwait
```

Bottlenecks:

- 14,7% on omp taskwait

Bonus - Parallelized implementation and approach

- Take hybrid approach as a base and let each MPI process have 12 threads.
- Using intrinsics with unaligned load in distance_squared:

```
__m256 va, vb, tmp, tmp2, partial_sum;
partial_sum = _mm256_set1_ps( w: 0);
for(int i = 0; i < ub; i+=8)
{
    va  = _mm256_loadu_ps(&(a.coordinates[i]));
    vb  = _mm256_loadu_ps(&(b.coordinates[i]));
    tmp = _mm256_sub_ps (va,vb);
    tmp2 = _mm256_mul_ps(tmp,tmp);

    partial_sum = _mm256_add_ps(partial_sum, tmp2);
}
```

→ Speed-up increased from 6,38 to 6,59

Bonus - Parallelized implementation and approach

1- Custom generate problem to support alignment:

```
for(int n = 0; n < num_points; ++n){

    // set MPI process' local tree points
    if (n >= start && n < end) {
        float* coords = (float*)aligned_alloc( alignment: 32, coordSize);
        for(int d = 0; d < dim; ++d) {
            *(coords + d) = distribution( &: random);
        }
        points[n - start] = new Point(dim,  ID: n + 1, coords);
    }
```

2- Aligned load:

```
__m256 va, vb, tmp, tmp2, partial_sum;
partial_sum = _mm256_set1_ps( w: 0);
for(int i = 0; i < ub; i+=8)
{
    va  = _mm256_load_ps(&(a.coordinates[i]));
    vb  = _mm256_load_ps(&(b.coordinates[i]));
    tmp = _mm256_sub_ps (va, vb);
    tmp2 = _mm256_mul_ps(tmp, tmp);

    partial_sum = _mm256_add_ps(partial_sum, tmp2);
}
```

Speed-up increased due to 1 and 2
from 6,59 to 7,44

Bonus - Final Performance Results

- Speed-up ~7,44
- Comparison to Amdahl’s Law:
  s(12 threads x 3 worker MPI processes = 36) ≈ 8,13
- Comparison to hybrid part:
  - Cache misses reduced by ~14,8%
    captured via “*perf stat -e cache-misses*”
  - Overhead of distance_squared function is reduced by 4.97%
    captured via “*perf diff hybrid/perf.data bonus/perf.data*”

Conclusion

- ➢ Sequential code analysis is crucial to get a base understanding of the problem.

- ➢ Using Amdahl’s Law points the perfect parallelism speed-up.
  This way, parallel programmers know where to stop.

- ➢ Parallelism in shared memory systems:
  - Limitation: synchronization (omp waits in this case)
  - Limitation: cache/memory misses
  - Limitation: load-balancing (used omp tasks to distribute equal amount of work)
  - Advantage: inter-thread communication (done via memory, faster than I/O)

Conclusion

- Parallelism in distributed memory systems:
  - Limitation: I/O (communication via I/O, slower than memory)
  - Limitation: need a tricky algorithm to divide data
  - Advantage: enablement to work on powerful clusters

- Parallelism in hybrid systems:
  - Leverage across-nodes and in-node parallelism with MPI and OpenMP

- Pushing the limits of hardware:
  - Using the features of underlying hardware leads to better results (used intrinsics)

Thanks For Your Attention

Discussion