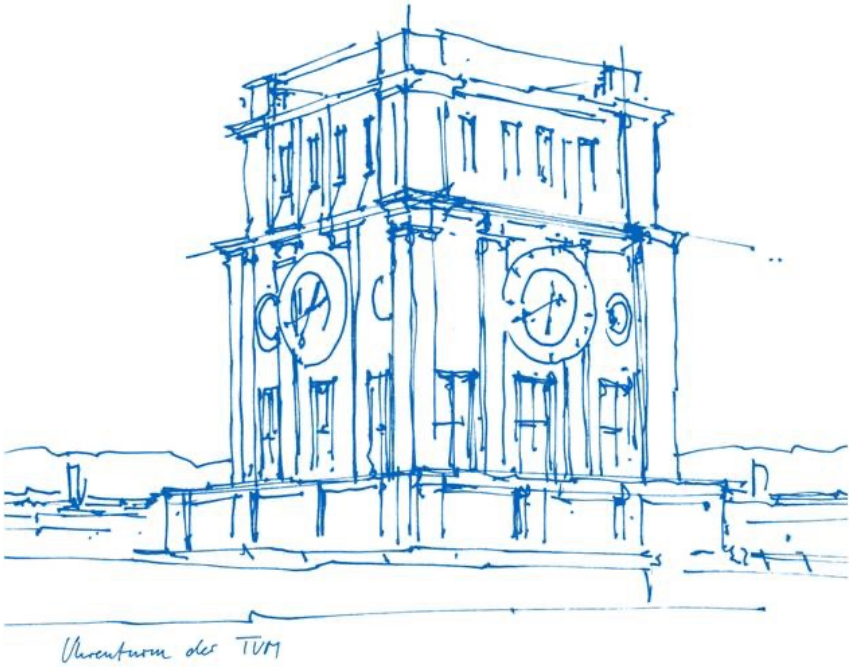
Parallel Programming SS21 Final Project

Project 1: Graph Convolutional Neural Networks

Group 119

14.07.2021

Batuhan Erden



Outline

1. **Sequential code analysis**
2. Improved sequential code analysis
3. OpenMP
4. Message Passing Interface (MPI)
5. Hybrid (OpenMP + MPI)
6. Bonus
7. Conclusion

Sequential code analysis - Profiling

Perf results (79K samples):

```
Samples: 79K of event 'cycles:u', Event count (approx.): 60334208124
Overhead  Command     Shared Object        Symbol
  99.12%  sequential  sequential           [.] first_layer_transform
   0.30%  sequential  [unknown]            [k] 0xffffffff96c010a7
   0.10%  sequential  sequential           [.] second_layer_transform
   0.08%  sequential  sequential           [.] first_layer_aggregate
   0.05%  sequential  sequential           [.] second_layer_aggregate
   0.03%  sequential  libstdc++.so.6.0.29  [.] __dynamic_cast
   0.01%  sequential  libstdc++.so.6.0.29  [.] std::__detail::_List_node_base::_M_hook
   0.01%  sequential  sequential           [.] create_graph
   0.01%  sequential  libc-2.33.so         [.] 0x0000000000088647
   0.01%  sequential  sequential           [.] Model::initialize_weights
   0.01%  sequential  sequential           [.] Node::get_prediction
   0.01%  sequential  libc-2.33.so         [.] 0x00000000000889eb
   0.01%  sequential  libstdc++.so.6.0.29  [.] __cxxabiv1::__si_class_type_info::__do_dyncast
   0.01%  sequential  libc-2.33.so         [.] 0x0000000000163b81
   0.01%  sequential  libstdc++.so.6.0.29  [.] __cxxabiv1::__vmi_class_type_info::__do_dyncast
   0.00%  sequential  libstdc++.so.6.0.29  [.] std::locale::id::_M_id
   0.00%  sequential  libstdc++.so.6.0.29  [.] std::__ostream_insert<char, std::char_traits<char> >
   0.00%  sequential  sequential           [.] main
   0.00%  sequential  sequential           [.] Node::Node
   0.00%  sequential  libc-2.33.so         [.] 0x000000000008855f
   0.00%  sequential  libstdc++.so.6.0.29  [.] std::basic_filebuf<char, std::char_traits<char> >::xsgetn
   0.00%  sequential  sequential           [.] Model::read_binary
```

Sequential code analysis - Amdahl‘s law

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

f = fraction of parallel execution
p = number of parallel tasks/threads/processes

From the output of perf in the previous slide, we can deduce that **99.5%** of the code is parallelizable. As we have **16 cores** on the server, we can apply the formula:

$$\frac{1}{(1-99.5) + \frac{99.5}{16.0}}$$

As can be seen from the calculation above, with no improvements on the sequential code, we should get a maximum speedup of **14.88x** from the parallelization.

However, this is barely above the minimum passing threshold. It was an imperative thing to improve the sequential code before the parallelization to get a good speedup.

Outline

1. Sequential code analysis
2. **Improved sequential code analysis**
3. OpenMP
4. Message Passing Interface (MPI)
5. Hybrid (OpenMP + MPI)
6. Bonus
7. Conclusion

Improved sequential code analysis - Profiling

Perf results (6K samples):

```
Samples: 6K of event 'cycles:u', Event count (approx.): 4524630865
Overhead  Command     Shared Object        Symbol
  89.79%  sequential  sequential           [.] first_layer_transform
   3.76%  sequential  [unknown]            [k] 0xffffffff96c010a7
   1.27%  sequential  sequential           [.] first_layer_aggregate
   0.56%  sequential  sequential           [.] second_layer_aggregate
   0.56%  sequential  sequential           [.] second_layer_transform
   0.34%  sequential  libstdc++.so.6.0.29  [.] __dynamic_cast
   0.19%  sequential  libc-2.33.so         [.] 0x00000000000889eb
   0.11%  sequential  libc-2.33.so         [.] 0x0000000000163b81
   0.09%  sequential  libc-2.33.so         [.] 0x0000000000088647
   0.09%  sequential  sequential           [.] create_graph
   0.08%  sequential  libstdc++.so.6.0.29  [.] __cxxabiv1::__vmi_class_type_info::__do_dyncast
   0.08%  sequential  sequential           [.] Model::initialize_weights
   0.08%  sequential  libstdc++.so.6.0.29  [.] std::basic_filebuf<char, std::char_traits<char> >::xsgetn
   0.07%  sequential  libstdc++.so.6.0.29  [.] __cxxabiv1::__si_class_type_info::__do_dyncast
   0.07%  sequential  libc-2.33.so         [.] 0x000000000008897d
   0.07%  sequential  [unknown]            [k] 0xffffffff96c00158
   0.06%  sequential  libstdc++.so.6.0.29  [.] std::__detail::_List_node_base::_M_hook
   0.06%  sequential  sequential           [.] Node::get_prediction
   0.05%  sequential  sequential           [.] Node::Node
   0.05%  sequential  libstdc++.so.6.0.29  [.] std::locale::id::_M_id
   0.04%  sequential  libstdc++.so.6.0.29  [.] std::ostream::_M_insert<long>
   0.04%  sequential  libstdc++.so.6.0.29  [.] std::basic_streambuf<char, std::char_traits<char> >::xsputn
   0.04%  sequential  sequential           [.] main
   0.04%  sequential  sequential           [.] std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul,
```

Improved sequential code analysis - Amdahl‘s law

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

f = fraction of parallel execution
p = number of parallel tasks/threads/processes

From the output of perf in the previous slide, we can deduce that **92.5%** of the code is parallelizable. As we have **16 cores** on the server, we can apply the formula:

$$\frac{1}{(1-92.5) + \frac{92.5}{16.0}}$$

As can be seen from the calculation above, with no improvements on the sequential code, we should get a maximum speedup of **7.53x** from the parallelization.

Just the improved version of the sequential code gets a speedup of **7x**. So, in the end, according to the Amdahl’s law, we should get a speedup of **7.53x * 7x ≈ 53x**.

Outline

1. Sequential code analysis
2. Improved sequential code analysis
3. **OpenMP**
4. Message Passing Interface (MPI)
5. Hybrid (OpenMP + MPI)
6. Bonus
7. Conclusion

OpenMP - Parallelized implementation and approach

- Parallelization of the outermost loops (starting with the one in **first_layer_transform**) with a simple **#pragma omp parallel for**.

- Tried to play with **the omp calls** and added a **schedule** to **#pragma omp parallel for**.
  - Neither **static nor dynamic scheduling** did not seem to help with the speedup.
  - Playing with **the number of threads** made it even worse in terms of our speedup.

- Completed the parallelization by using **tasks** and parallelized the body block of the outermost for loop in **first_layer_transform**.
  - Since tasks will be created on request, I thought that this might be better to use.
  - However, the speedup was around the same with just **#pragma omp parallel for**.

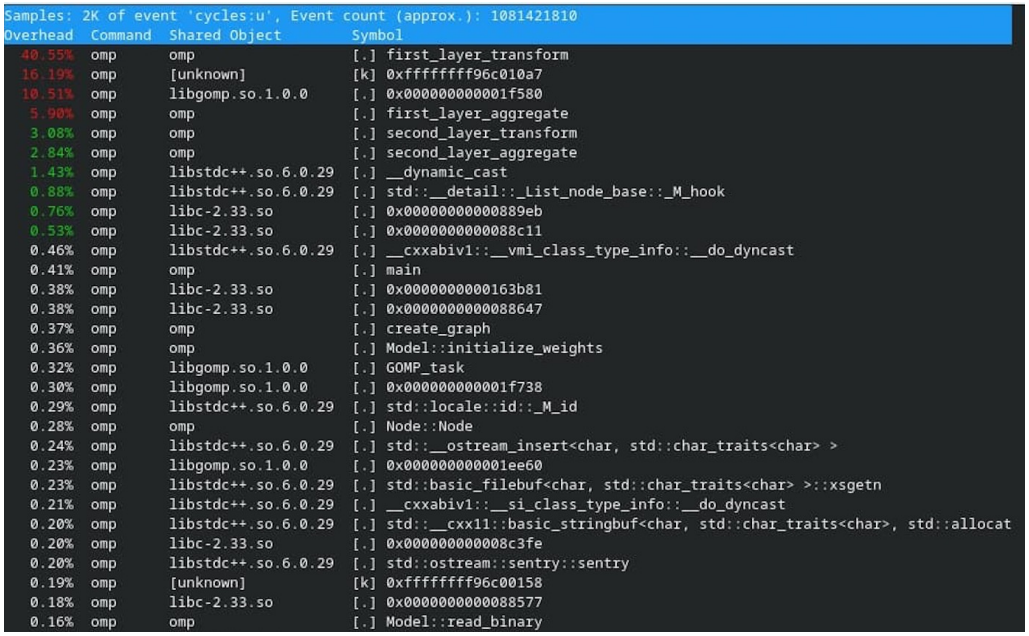
- Also tried **tasks** in the other functions, but it only made things worse.
  - Creates **overhead** since all those functions must wait for each other before starting their computations with **omp taskwait**.

- Observed a small speedup by parallelizing the outermost loop in all of these functions by **#pragma omp parallel for** *without specifying any schedule or number of threads.*

OpenMP - Intermediate Speed-up results, profiling

This implementation gets a speedup around **25x**.

Perf results (2K samples):



```
Samples: 2K of event 'cycles:u', Event count (approx.): 1081421810
Overhead  Command  Shared Object        Symbol
  40.55%  omp      omp                  [.] first_layer_transform
  16.19%  omp      [unknown]            [k] 0xffffffff96c010a7
  10.51%  omp      libgomp.so.1.0.0     [.] 0x000000000001f580
   5.90%  omp      omp                  [.] first_layer_aggregate
   3.08%  omp      omp                  [.] second_layer_transform
   2.84%  omp      omp                  [.] second_layer_aggregate
   1.43%  omp      libstdc++.so.6.0.29  [.] __dynamic_cast
   0.88%  omp      libstdc++.so.6.0.29  [.] std::__detail::_List_node_base::_M_hook
   0.76%  omp      libc-2.33.so         [.] 0x00000000000889eb
   0.53%  omp      libc-2.33.so         [.] 0x0000000000088c11
   0.46%  omp      libstdc++.so.6.0.29  [.] __cxxabiv1::__vmi_class_type_info::__do_dyncast
   0.41%  omp      omp                  [.] main
   0.38%  omp      libc-2.33.so         [.] 0x0000000000163b81
   0.38%  omp      libc-2.33.so         [.] 0x0000000000088647
   0.37%  omp      omp                  [.] create_graph
   0.36%  omp      omp                  [.] Model::initialize_weights
   0.32%  omp      libgomp.so.1.0.0     [.] GOMP_task
   0.30%  omp      libgomp.so.1.0.0     [.] 0x000000000001f738
   0.29%  omp      libstdc++.so.6.0.29  [.] std::locale::id::_M_id
   0.28%  omp      omp                  [.] Node::Node
   0.24%  omp      libstdc++.so.6.0.29  [.] std::__ostream_insert<char, std::char_traits<char> >
   0.23%  omp      libgomp.so.1.0.0     [.] 0x000000000001ee60
   0.23%  omp      libstdc++.so.6.0.29  [.] std::basic_filebuf<char, std::char_traits<char> >::xsgetn
   0.21%  omp      libstdc++.so.6.0.29  [.] __cxxabiv1::__si_class_type_info::__do_dyncast
   0.20%  omp      libstdc++.so.6.0.29  [.] std::__cxx11::basic_stringbuf<char, std::char_traits<char>, std::allocat
   0.20%  omp      libc-2.33.so         [.] 0x000000000008c3fe
   0.20%  omp      libstdc++.so.6.0.29  [.] std::ostream::sentry::sentry
   0.19%  omp      [unknown]            [k] 0xffffffff96c00158
   0.18%  omp      libc-2.33.so         [.] 0x0000000000088577
   0.16%  omp      omp                  [.] Model::read_binary
```

OpenMP - Final Implementation improvements and new speed-up

- Also parallelized the other for loops (in the **main** and **create_graph** functions) also with **#pragma omp parallel for** and each of those gave a small speedup to the problem.

- Set **the number of threads** of omp by **omp_set_num_threads**.
  - Tested some values on the Submission Server, and found a sweet spot when the number of threads is set to **16**, and in doing so, this gave us speedup of **50-52.5x**.

Amdahl's law: **7.53x * 7x ≈ 53x**.

The speedup I got: **50-52.5x**.

OpenMP - Final Implementation improvements and new speed-up

Perf results (10K samples):

```
Samples: 10K of event 'cycles:u', Event count (approx.): 6024009648
Overhead  Command  Shared Object        Symbol
  87.65%  omp      omp                  [.] first_layer_transform
   2.10%  omp      libgomp.so.1.0.0     [.] 0x000000000001f738
   1.37%  omp      [unknown]            [k] 0xffffffff96c010a7
   1.33%  omp      omp                  [.] first_layer_aggregate
   0.49%  omp      omp                  [.] second_layer_aggregate
   0.40%  omp      omp                  [.] second_layer_transform
   0.29%  omp      libstdc++.so.6.0.29  [.] __dynamic_cast
   0.17%  omp      libc-2.33.so         [.] 0x00000000000889eb
   0.16%  omp      libc-2.33.so         [.] 0x000000000008c347
   0.15%  omp      libstdc++.so.6.0.29  [.] std::__detail::_List_node_base::_M_hook
   0.14%  omp      libgomp.so.1.0.0     [.] 0x000000000001f73a
   0.12%  omp      libc-2.33.so         [.] 0x0000000000088c11
   0.10%  omp      omp                  [.] Node::Node
   0.10%  omp      libgomp.so.1.0.0     [.] 0x000000000001f580
   0.09%  omp      libstdc++.so.6.0.29  [.] __cxxabiv1::__vmi_class_type_info::__do_dyncast
   0.08%  omp      libstdc++.so.6.0.29  [.] std::__ostream_insert<char, std::char_traits<char> >
   0.08%  omp      libc-2.33.so         [.] 0x000000000008c3fe
   0.08%  omp      libc-2.33.so         [.] 0x0000000000088647
   0.07%  omp      omp                  [.] Node::get_prediction
   0.07%  omp      libc-2.33.so         [.] 0x000000000008d220
   0.07%  omp      libstdc++.so.6.0.29  [.] std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<
   0.07%  omp      libc-2.33.so         [.] 0x000000000008d23a
   0.07%  omp      libstdc++.so.6.0.29  [.] __cxxabiv1::__si_class_type_info::__do_dyncast
   0.06%  omp      libc-2.33.so         [.] 0x0000000000088885
   0.06%  omp      omp                  [.] create_graph
```

OpenMP - Errors found after the submission

The data that we were working with in the OpenMP submission was faulty. My submission was working with it, but when I tried it after the deadline, it didn't work because the faulty data was corrected after the submission deadline.

How I fixed the error (If the data was correct at that time, it'd have taken me less than 2 minutes to solve it):

gcn_omp.cpp — View file @a8fb421a

```
@@ -84,7 +84,7 @@ void first_layer_aggregate(Node** nodes, int num_nodes, Model& model) {
            // aggregate normalized message and add bias
            for (int c = 0; c < node->dim_hidden; ++c) {
-               node->hidden[c] += message[c] / norm + model.bias_1[c];
+               node->hidden[c] += message[c] / norm + model.bias_1[c] / node->degree;
            }
        }

@@ -134,7 +134,7 @@ void second_layer_aggregate(Node** nodes, int num_nodes, Model& model) {
            // aggregate normalized message and add bias
            for (int c = 0; c < node->num_classes; ++c) {
-               node->logits[c] += message[c] / norm + model.bias_2[c];
+               node->logits[c] += message[c] / norm + model.bias_2[c] / node->degree;
            }
        }
    }
```

OpenMP - Further improvements

Working with the next MPI submission, I found out another improvement to the code.

With this change, I saw an enormous speedup of **90x**.

```
/*****************************************************************************/
void first_layer_transform(Node** nodes, int num_nodes, Model& model) {
    // transform
    for (int n = 0; n < num_nodes; ++n) {
        #pragma omp task default(none) firstprivate(nodes, n, model)
        {
            Node* node = nodes[n];

            for (int c_in = 0; c_in < node->dim_features; ++c_in) {
                float x_in = node->x[c_in];
                float* weight_1_start_idx = model.weight_1 + (c_in * node->dim_hidden);

                // if the input is zero, do not calculate the corresponding hidden values
                if (x_in == 0) {
                    continue;
                }

                for (int c_out = 0; c_out < node->dim_hidden; ++c_out) {
                    node->tmp_hidden[c_out] += x_in * *(weight_1_start_idx + c_out);
                }
            }
        }
    }

    #pragma omp taskwait
}
/*****************************************************************************/
```

```
/*****************************************************************************/
// computation in second layer
void second_layer_transform(Node** nodes, int num_nodes, Model& model) {
    // transform
    #pragma omp parallel for
    for (int n = 0; n < num_nodes; ++n) {
        Node* node = nodes[n];

        for (int c_in = 0; c_in < node->dim_hidden; ++c_in) {
            float h_in = node->hidden[c_in];
            float* weight_2_start_idx = model.weight_2 + (c_in * node->num_classes);

            // if the input is zero, do not calculate the corresponding logits
            if (h_in == 0) {
                continue;
            }

            for (int c_out = 0; c_out < node->num_classes; ++c_out) {
                node->tmp_logits[c_out] += h_in * *(weight_2_start_idx + c_out);
            }
        }
    }
}
/*****************************************************************************/
```

OpenMP - Further improvements

Perf results (2K samples):

```
Samples: 2K of event 'cycles:u', Event count (approx.): 938266307
Overhead  Command  Shared Object        Symbol
  43.86%  omp      omp                  [.] first_layer_transform
   7.36%  omp      [unknown]            [k] 0xffffffff96c010a7
   6.63%  omp      libgomp.so.1.0.0     [.] 0x000000000001f738
   6.43%  omp      omp                  [.] first_layer_aggregate
   3.60%  omp      omp                  [.] second_layer_transform
   2.97%  omp      omp                  [.] second_layer_aggregate
   1.93%  omp      libstdc++.so.6.0.29  [.] __dynamic_cast
   1.01%  omp      libc-2.33.so         [.] 0x000000000008c3fe
   0.89%  omp      libc-2.33.so         [.] 0x0000000000088c11
   0.75%  omp      libc-2.33.so         [.] 0x00000000000889eb
   0.65%  omp      libgomp.so.1.0.0     [.] GOMP_task
   0.62%  omp      libc-2.33.so         [.] 0x0000000000088647
   0.55%  omp      libgomp.so.1.0.0     [.] 0x000000000001f580
   0.51%  omp      omp                  [.] Node::Node
   0.47%  omp      omp                  [.] create_graph
   0.46%  omp      libc-2.33.so         [.] 0x000000000008c347
   0.44%  omp      omp                  [.] Model::initialize_weights
   0.37%  omp      libc-2.33.so         [.] 0x0000000000088b91
   0.34%  omp      libc-2.33.so         [.] 0x000000000008d220
   0.32%  omp      libstdc++.so.6.0.29  [.] std::basic_filebuf<char, std::char_traits<char> >::xsgetn
   0.31%  omp      libstdc++.so.6.0.29  [.] __cxxabiv1::__si_class_type_info::__do_dyncast
   0.30%  omp      libstdc++.so.6.0.29  [.] std::__detail::_List_node_base::_M_hook
   0.30%  omp      libstdc++.so.6.0.29  [.] std::__ostream_insert<char, std::char_traits<char> >
   0.29%  omp      libc-2.33.so         [.] 0x000000000008302c
   0.28%  omp      [unknown]            [k] 0xffffffff96c00158
   0.28%  omp      libstdc++.so.6.0.29  [.] __cxxabiv1::__vmi_class_type_info::__do_dyncast
   0.27%  omp      omp                  [.] Node::get_prediction
```

Outline

1. Sequential code analysis
2. Improved sequential code analysis
3. OpenMP
4. **Message Passing Interface (MPI)**
5. Hybrid (OpenMP + MPI)
6. Bonus
7. Conclusion

MPI - Parallelized implementation and approach

Implementing a basic MPI framework would give me a speedup of **7x**.

I tried to investigate the data a bit and saw that the neighbours were not uniformly distributed and the first chunk would process **4-5 times more neighbours** than the others. So, I thought I could try **Dynamic Scheduling**:

```
if (rank == 0) { // Master
    for (; start < model.num_nodes; start += chunk_size) {
        provide_work_to_workers(&start);
    }

    send_kill_signal_to_workers(size);
} else { // Workers
    while (request_and_receive_work_from_master(rank, &start)) {
        const int end = std::min(start + chunk_size, model.num_nodes);

        // perform actual computation in network
        perform_computation(start, end, nodes, model);

        // compute the accuracy
        compute_accuracy(start, end, nodes, model, acc);
    }
}
```

MPI - Intermediate Speed-up results, profiling

The dynamic scheduling approach gave me a speedup of **15x**.

Perf results (2K samples):

```
Samples: 2K of event 'cycles:u', Event count (approx.): 1017000941
Overhead  Command  Shared Object                Symbol
  39.51%  mpi      mpi                          [.] first_layer_transform
  17.98%  mpi      [unknown]                    [k] 0xffffffff96c010a7
   5.06%  mpi      mpi                          [.] first_layer_aggregate
   4.17%  mpi      mpi                          [.] second_layer_transform
   2.53%  mpi      mpi                          [.] second_layer_aggregate
   1.62%  mpi      ld-2.33.so                   [.] do_lookup_x
   1.53%  mpi      libstdc++.so.6.0.29          [.] __dynamic_cast
   0.71%  mpi      ld-2.33.so                   [.] strcmp
   0.67%  mpi      libstdc++.so.6.0.29          [.] std::__detail::_List_node_base::_M_hook
   0.60%  mpi      libc-2.33.so                 [.] 0x000000000008c347
   0.52%  mpi      libc-2.33.so                 [.] 0x0000000000088647
   0.49%  mpi      libc-2.33.so                 [.] 0x00000000000889eb
   0.46%  mpi      libc-2.33.so                 [.] 0x0000000000076b19
   0.43%  mpi      libc-2.33.so                 [.] 0x0000000000076b2d
   0.40%  mpi      mpi                          [.] Model::initialize_weights
   0.38%  mpi      libstdc++.so.6.0.29          [.] __cxxabiv1::__si_class_type_info::__do_dyncast
   0.34%  mpi      libc-2.33.so                 [.] 0x000000000008c3fe
   0.33%  mpi      mpi                          [.] Node::get_prediction
   0.31%  orted    [unknown]                    [k] 0xffffffff96c010a7
   0.29%  mpi      libstdc++.so.6.0.29          [.] std::__ostream_insert<char, std::char_traits<char> >
   0.29%  mpi      ld-2.33.so                   [.] _dl_relocate_object
   0.29%  mpi      mpi                          [.] create_graph
   0.28%  mpi      libc-2.33.so                 [.] 0x0000000000163b81
   0.28%  mpi      libc-2.33.so                 [.] 0x0000000000088c11
   0.26%  orted    ld-2.33.so                   [.] strcmp
   0.25%  mpi      libc-2.33.so                 [.] 0x000000000008b18b
   0.24%  orted    ld-2.33.so                   [.] do_lookup_x
```

MPI - Intermediate Speed-up results, profiling

- Tried the optimization described in the final OpenMP slide also in this submission, and this got me a speedup around **28x.**

- I was happy to have the 1st place in the leaderboard.

However, I did not know something important!

MPI - Final Implementation improvements and new speed-up

There were 2 important points (I fixed them in the end!):

1. I had missed the statement in the PDF stating that the input files may only be found on RANK 0, i.e. we may only read files on RANK 0.
2. The dataset we were given was faulty and it got fixed. So, our implementation wasn't working anymore and there must have been a mistake.

- Handled these 2 points above and barely passed the speedup threshold.
  - First made sure that only **the master** would read the files.
  - Making only this change dropped our speedup by a great margin to **12x**.

- Reinvestigated the dataset to find out that now the neighbors are more uniformly distributed.
  - Got rid of my **Dynamic Scheduling** approach.
  - Wanted to have a clearer approach and tried working with **Gathers** and **Scatters**, and came up with a solution using those (I omitted using **Scatters** in the end).

At this point, the speedup I was getting was **19x**.

MPI - Final Implementation improvements and new speed-up

Some final changes that improved the speedup to **23x**:

- At first, only **the master** was doing **the aggregate** and **accuracy computation** operations. It was because I wanted to keep the **communication overhead** as small as possible.
- But we have the same **overhead** when broadcasting the output of **first layer transform** instead of the output of **first layer aggregate**!
- So, I did that and parallelized also **first layer aggregate.** Also did the same thing to **second layer aggregate** and **accuracy computation** operations because I wanted to have consistency. As this did not decrease our speedup, and I think it gave it a small jump, I decided to keep it.

Amdahl's law: **7.53x * 7x ≈ 53x**.
The speedup I got: **23x**.

MPI - Final Implementation improvements and new speed-up

Perf results (4K samples):

```
Samples: 4K of event 'cycles:u', Event count (approx.): 1182428464
Overhead  Command  Shared Object                  Symbol
  45.47%  mpi      mpi                            [.] first_layer_transform
  15.40%  mpi      [unknown]                      [k] 0xffffffff96c010a7
   4.77%  mpi      mpi                            [.] first_layer_aggregate
   3.57%  mpi      mpi                            [.] second_layer_transform
   2.05%  mpi      mpi                            [.] second_layer_aggregate
   1.34%  mpi      ld-2.33.so                     [.] do_lookup_x
   1.29%  mpi      libstdc++.so.6.0.29            [.] __dynamic_cast
   0.74%  mpi      ld-2.33.so                     [.] strcmp
   0.65%  mpi      libc-2.33.so                   [.] 0x000000000008c3fe
   0.57%  mpi      libc-2.33.so                   [.] 0x00000000000889eb
   0.51%  mpi      libc-2.33.so                   [.] 0x000000000008c347
   0.45%  mpi      libstdc++.so.6.0.29            [.] __cxxabiv1::__si_class_type_info::__do_dyncast
   0.42%  mpi      mpi                            [.] create_graph
   0.40%  mpi      libc-2.33.so                   [.] 0x0000000000163b81
   0.40%  mpi      libstdc++.so.6.0.29            [.] std::__detail::_List_node_base::_M_hook
   0.38%  orted    [unknown]                      [k] 0xffffffff96c010a7
   0.38%  mpi      libstdc++.so.6.0.29            [.] __cxxabiv1::__vmi_class_type_info::__do_dyncast
   0.37%  mpi      mpi                            [.] Model::initialize_weights
   0.37%  orted    ld-2.33.so                     [.] do_lookup_x
   0.36%  mpi      libc-2.33.so                   [.] 0x0000000000076b19
   0.36%  mpi      libc-2.33.so                   [.] 0x0000000000076b2d
   0.34%  mpi      mpi                            [.] Node::get_prediction
   0.33%  mpi      libstdc++.so.6.0.29            [.] std::basic_filebuf<char, std::char_traits<char> >::xsgetn
   0.32%  mpi      libc-2.33.so                   [.] 0x0000000000088647
   0.31%  mpi      libc-2.33.so                   [.] 0x000000000008d23a
   0.29%  mpi      libc-2.33.so                   [.] __memcpy_avx_unaligned_erms
   0.28%  mpi      libc-2.33.so                   [.] 0x0000000000088c11
   0.28%  mpi      mpi                            [.] Node::Node
```

Outline

1. Sequential code analysis
2. Improved sequential code analysis
3. OpenMP
4. Message Passing Interface (MPI)
5. **Hybrid (OpenMP + MPI)**
6. Bonus
7. Conclusion

Hybrid - Parallelized implementation and approach

I already had tried lots of things trying to improve my **OpenMP** and **MPI** submissions.

My solution is basically the combination of our **MPI** and **OpenMP** implementations. I build my **OpenMP** parallelization on my **MPI** implementation and tried to make use of *the shared memory of each compute node in themselves*.

Hybrid - Intermediate Speed-up results, profiling

Successfully building this framework gave me a speedup of **40x**.

Perf results (7K samples):

```
Samples: 7K of event 'cycles:u', Event count (approx.): 2959302280
Overhead  Command  Shared Object                Symbol
  36.75%  hybrid   hybrid                       [.] first_layer_transform
  11.08%  hybrid   [unknown]                    [k] 0xffffffff96c010a7
   9.67%  hybrid   libgomp.so.1.0.0             [.] 0x000000000001f580
   3.52%  hybrid   hybrid                       [.] first_layer_aggregate
   3.05%  hybrid   libc-2.33.so                 [.] __memcpy_avx_unaligned_erms
   2.62%  hybrid   libgomp.so.1.0.0             [.] 0x000000000001f738
   2.18%  hybrid   hybrid                       [.] second_layer_transform
   1.99%  hybrid   hybrid                       [.] second_layer_aggregate
   1.83%  hybrid   libgomp.so.1.0.0             [.] 0x000000000001f582
   0.92%  hybrid   ld-2.33.so                   [.] do_lookup_x
   0.86%  hybrid   [unknown]                    [k] 0xffffffff96c00158
   0.79%  hybrid   libstdc++.so.6.0.29          [.] __dynamic_cast
   0.47%  hybrid   hybrid                       [.] create_graph
   0.47%  hybrid   libc-2.33.so                 [.] 0x00000000000889eb
   0.45%  hybrid   libc-2.33.so                 [.] 0x0000000000085d24
   0.44%  hybrid   libc-2.33.so                 [.] 0x0000000000088c11
   0.41%  hybrid   libstdc++.so.6.0.29          [.] std::__detail::_List_node_base::_M_hook
   0.40%  hybrid   ld-2.33.so                   [.] strcmp
   0.38%  hybrid   libc-2.33.so                 [.] 0x000000000008c347
   0.35%  hybrid   libuct.so.0.0.0              [.] uct_tcp_iface_progress
   0.33%  hybrid   libucs.so.0.0.0              [.] ucs_event_set_wait
   0.31%  hybrid   libc-2.33.so                 [.] 0x0000000000076b19
   0.30%  hybrid   libc-2.33.so                 [.] 0x0000000000085daa
   0.29%  hybrid   libc-2.33.so                 [.] 0x000000000008c3fe
   0.29%  hybrid   libc-2.33.so                 [.] 0x0000000000076b2d
   0.26%  hybrid   libc-2.33.so                 [.] 0x0000000000163b81
   0.26%  hybrid   mca_pml_ucx.so               [.] mca_pml_ucx_recv
   0.24%  hybrid   libc-2.33.so                 [.] 0x000000000008897d
   0.22%  hybrid   hybrid                       [.] Node::get_prediction
```

Hybrid - Final Performance Results

Changed the scheduling of the OMP parallel calls to **Dynamic Scheduling** because:

- In the transform functions, some nodes can have more 0s in their inputs!
- In the aggregation functions, neighbors might not be uniformly distributed across the nodes!

The **Dynamic Scheduling** here gave me a speedup of **44x**.

Amdahl's law: **7.53x * 7x ≈ 53x**.
The speedup I got: **44x**.

Hybrid - Final Performance Results

Perf results (27K samples):

```
Samples: 27K of event 'cycles:u', Event count (approx.): 5154519850
Overhead  Command  Shared Object                Symbol
  21.13%  hybrid   hybrid                       [.] first_layer_transform
   8.89%  hybrid   [unknown]                    [k] 0xffffffff96c00158
   6.37%  hybrid   [unknown]                    [k] 0xffffffff96c010a7
   5.50%  hybrid   libgomp.so.1.0.0             [.] 0x000000000001f580
   5.08%  hybrid   libc-2.33.so                 [.] 0x0000000000085d24
   4.46%  hybrid   libc-2.33.so                 [.] 0x0000000000085daa
   4.22%  hybrid   libucs.so.0.0.0              [.] ucs_event_set_wait
   4.00%  hybrid   libuct.so.0.0.0              [.] uct_tcp_iface_progress
   2.03%  hybrid   hybrid                       [.] first_layer_aggregate
   1.99%  hybrid   libucp.so.0.0.0              [.] ucp_worker_progress
   1.96%  hybrid   libc-2.33.so                 [.] __memcpy_avx_unaligned_erms
   1.65%  hybrid   libc-2.33.so                 [.] 0x0000000000085d91
   1.60%  hybrid   libc-2.33.so                 [.] 0x00000000001006a9
   1.59%  hybrid   libopen-pal.so.40.30.0       [.] opal_progress
   1.47%  hybrid   libgomp.so.1.0.0             [.] 0x000000000001f738
   1.25%  hybrid   hybrid                       [.] second_layer_transform
   1.24%  hybrid   libuct.so.0.0.0              [.] 0x0000000000016239
   1.12%  hybrid   hybrid                       [.] second_layer_aggregate
   1.11%  hybrid   libgomp.so.1.0.0             [.] 0x000000000001f582
   0.86%  hybrid   libuct.so.0.0.0              [.] 0x0000000000016320
   0.79%  hybrid   libc-2.33.so                 [.] 0x0000000000085d89
   0.63%  hybrid   ld-2.33.so                   [.] do_lookup_x
   0.47%  hybrid   libstdc++.so.6.0.29          [.] __dynamic_cast
   0.47%  hybrid   libopen-pal.so.40.30.0       [.] 0x0000000000075547
   0.34%  hybrid   libc-2.33.so                 [.] 0x00000000000889eb
   0.33%  hybrid   libstdc++.so.6.0.29          [.] std::__detail::_List_node_base::_M_hook
```

Hybrid - Further improvements

Achieved a speedup of **76x (*0.27244s*)** and secured the **1st place in the leaderboard**!

- Decreased *the communication overhead* by delegating the **first layer transform** operation only to **the master**.

- Delegating the **first layer transform** operation only to **the master**, and then have **the master** broadcast the output of the **first layer transform** operation (**big_tmp_hidden - *tmp_hidden for the full graph***) to **the other processes**, I decreased the communication overhead caused by *communicating the **X** values* and achieve a speedup of **76x**.

However, I decided not to use this solution here and used it in the bonus submission instead!

Hybrid - Further improvements

Perf results (7K samples):

```
Samples: 7K of event 'cycles:u', Event count (approx.): 2701310003
Overhead  Command  Shared Object                  Symbol
  39.71%  hybrid   hybrid                         [.] first_layer_transform
   8.97%  hybrid   [unknown]                      [k] 0xffffffff96c010a7
   7.92%  hybrid   libgomp.so.1.0.0              [.] 0x000000000001f580
   3.87%  hybrid   hybrid                         [.] first_layer_aggregate
   2.77%  hybrid   libgomp.so.1.0.0              [.] 0x000000000001f738
   2.39%  hybrid   hybrid                         [.] second_layer_transform
   1.70%  hybrid   hybrid                         [.] second_layer_aggregate
   1.30%  hybrid   libgomp.so.1.0.0              [.] 0x000000000001f582
   1.22%  hybrid   [unknown]                      [k] 0xffffffff96c00158
   1.19%  hybrid   ld-2.33.so                     [.] do_lookup_x
   0.97%  hybrid   libstdc++.so.6.0.29            [.] __dynamic_cast
   0.57%  hybrid   libc-2.33.so                   [.] 0x0000000000085d24
   0.57%  hybrid   libc-2.33.so                   [.] 0x0000000000085daa
   0.57%  hybrid   libc-2.33.so                   [.] 0x000000000008c347
   0.54%  hybrid   libc-2.33.so                   [.] 0x000000000008c789
   0.50%  hybrid   libc-2.33.so                   [.] 0x0000000000088c11
   0.49%  hybrid   libucs.so.0.0.0                [.] ucs_event_set_wait
   0.48%  hybrid   libc-2.33.so                   [.] 0x000000000008c3fe
   0.44%  hybrid   hybrid                         [.] create_graph
   0.38%  hybrid   libuct.so.0.0.0                [.] uct_tcp_iface_progress
   0.33%  hybrid   libc-2.33.so                   [.] 0x0000000000076b2d
   0.32%  hybrid   ld-2.33.so                     [.] strcmp
   0.31%  hybrid   libc-2.33.so                   [.] 0x00000000000889eb
   0.31%  hybrid   libstdc++.so.6.0.29            [.] std::__detail::_List_node_base::_M_hook
   0.30%  hybrid   libc-2.33.so                   [.] 0x0000000000076b19
```

Outline

1. Sequential code analysis
2. Improved sequential code analysis
3. OpenMP
4. Message Passing Interface (MPI)
5. Hybrid (OpenMP + MPI)
6. **Bonus**
7. Conclusion

Bonus - Parallelized implementation and approach

Decrease the communication overhead and apply **SIMD** on top of the **Hybrid** implementation!

```
void inner_first_layer_transform_simd(const int n, const int start, Node* node, const int c_in, float* weight_1, float* chunk_tmp_hidden) {
    const int last_chunk_idx = node->dim_hidden - (node->dim_hidden % 8) - 1;
    __m256 x_in = _mm256_set1_ps(node->x[c_in]); // node->x[c_in]

    // vectorized loop
    for (int c_out = 0; c_out <= last_chunk_idx; c_out += 8) {
        __m256 partial_sum = _mm256_loadu_ps(node->tmp_hidden + c_out);

        __m256 w_out = _mm256_loadu_ps(weight_1 + c_in * node->dim_hidden + c_out); // model.weight_1[c_in * node->dim_hidden + c_out]
        __m256 hidden_mult = _mm256_mul_ps(x_in, w_out); // node->x[c_in] * model.weight_1[c_in * node->dim_hidden + c_out]

        partial_sum = _mm256_add_ps(partial_sum, hidden_mult); // += node->x[c_in] * model.weight_1[c_in * node->dim_hidden + c_out]

        _mm256_storeu_ps(node->tmp_hidden + c_out, partial_sum);
        _mm256_storeu_ps(chunk_tmp_hidden + (n - start) * node->dim_hidden + c_out, partial_sum);
    }

    // the remainder chunk is processed
    for (int c_out = last_chunk_idx + 1; c_out < node->dim_hidden; ++c_out) {
        float tmp_hidden_item = node->x[c_in] * weight_1[c_in * node->dim_hidden + c_out];

        node->tmp_hidden[c_out] += tmp_hidden_item;
        chunk_tmp_hidden[(n - start) * node->dim_hidden + c_out] += tmp_hidden_item;
    }
}
```

Bonus - Final Performance Results

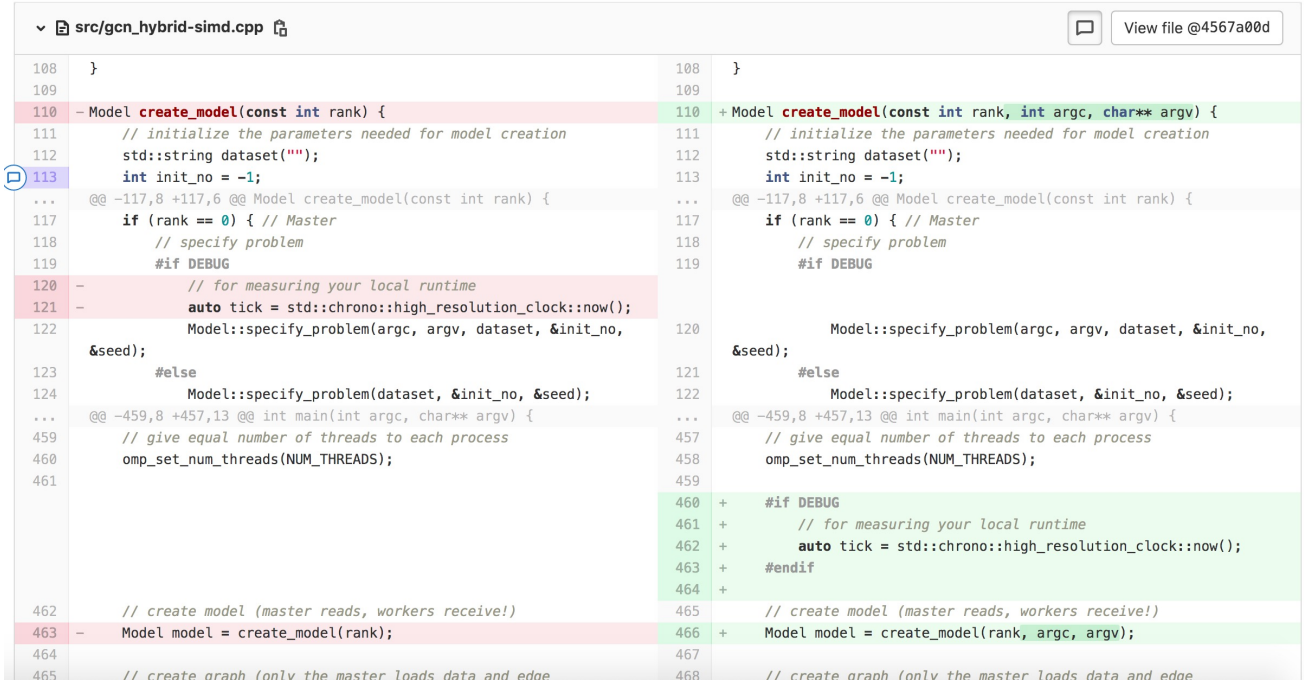
The final speedup achieved is **44x**.

Perf results (17K samples):

```
Samples: 17K of event 'cycles:u', Event count (approx.): 3294313483
Overhead  Command      Shared Object                Symbol
  10.14%  hybrid-simd  hybrid-simd                  [.] first_layer_transform
   7.87%  hybrid-simd  [unknown]                    [k] 0xffffffff96c00158
   7.87%  hybrid-simd  libgomp.so.1.0.0             [.] 0x000000000001f580
   7.30%  hybrid-simd  [unknown]                    [k] 0xffffffff96c010a7
   4.42%  hybrid-simd  libc-2.33.so                 [.] 0x0000000000085d24
   3.93%  hybrid-simd  hybrid-simd                  [.] inner_first_layer_transform_simd
   3.89%  hybrid-simd  libucs.so.0.0.0              [.] ucs_event_set_wait
   3.89%  hybrid-simd  libc-2.33.so                 [.] 0x0000000000085daa
   3.40%  hybrid-simd  libuct.so.0.0.0              [.] uct_tcp_iface_progress
   2.17%  hybrid-simd  hybrid-simd                  [.] first_layer_aggregate
   2.17%  hybrid-simd  hybrid-simd                  [.] inner_second_layer_transform_simd
   1.97%  hybrid-simd  libgomp.so.1.0.0             [.] 0x000000000001f738
   1.88%  hybrid-simd  libucp.so.0.0.0              [.] ucp_worker_progress
   1.59%  hybrid-simd  libopen-pal.so.40.30.0       [.] opal_progress
   1.48%  hybrid-simd  hybrid-simd                  [.] second_layer_aggregate
   1.37%  hybrid-simd  libc-2.33.so                 [.] 0x00000000001006a9
   1.36%  hybrid-simd  libc-2.33.so                 [.] 0x0000000000085d91
   1.13%  hybrid-simd  libuct.so.0.0.0              [.] 0x0000000000016239
   1.12%  hybrid-simd  libgomp.so.1.0.0             [.] 0x000000000001f582
   1.04%  hybrid-simd  hybrid-simd                  [.] inner_first_layer_aggregate_simd
   1.00%  hybrid-simd  ld-2.33.so                   [.] do_lookup_x
   0.93%  hybrid-simd  hybrid-simd                  [.] second_layer_transform
   0.85%  hybrid-simd  libuct.so.0.0.0              [.] 0x0000000000016320
   0.75%  hybrid-simd  libstdc++.so.6.0.29          [.] __dynamic_cast
   0.68%  hybrid-simd  libc-2.33.so                 [.] 0x0000000000085d89
   0.40%  hybrid-simd  libstdc++.so.6.0.29          [.] std::__detail::_List_node_base::_M_hook
   0.40%  hybrid-simd  libc-2.33.so                 [.] 0x00000000000088c11
   0.39%  hybrid-simd  hybrid-simd                  [.] inner_second_layer_aggregate_simd
   0.38%  hybrid-simd  libopen-pal.so.40.30.0       [.] 0x0000000000075547
   0.38%  hybrid-simd  libc-2.33.so                 [.] 0x00000000000889eb
   0.36%  hybrid-simd  libc-2.33.so                 [.] 0x000000000008c347
   0.31%  hybrid-simd  hybrid-simd                  [.] create_graph
```

Why the code cannot be run with DEBUG = 1

The changes that need to be made to fix this issue:



src/gcn_hybrid-simd.cpp — View file @4567a00d

```
108   }                                                    108   }
109                                                        109
110 - Model create_model(const int rank) {                 110 + Model create_model(const int rank, int argc, char** argv) {
111       // initialize the parameters needed for model creation   111       // initialize the parameters needed for model creation
112       std::string dataset("");                         112       std::string dataset("");
113       int init_no = -1;                                113       int init_no = -1;
...   @@ -117,8 +117,6 @@ Model create_model(const int rank) {   ...   @@ -117,8 +117,6 @@ Model create_model(const int rank) {
117       if (rank == 0) { // Master                       117       if (rank == 0) { // Master
118           // specify problem                           118           // specify problem
119           #if DEBUG                                    119           #if DEBUG
120 -             // for measuring your local runtime
121 -             auto tick = std::chrono::high_resolution_clock::now();
122               Model::specify_problem(argc, argv, dataset, &init_no,   120               Model::specify_problem(argc, argv, dataset, &init_no,
      &seed);                                                  &seed);
123           #else                                        121           #else
124               Model::specify_problem(dataset, &init_no, &seed);   122               Model::specify_problem(dataset, &init_no, &seed);
...   @@ -459,8 +457,13 @@ int main(int argc, char** argv) {   ...   @@ -459,8 +457,13 @@ int main(int argc, char** argv) {
459       // give equal number of threads to each process  457       // give equal number of threads to each process
460       omp_set_num_threads(NUM_THREADS);                458       omp_set_num_threads(NUM_THREADS);
461                                                        459
                                                           460 +     #if DEBUG
                                                           461 +         // for measuring your local runtime
                                                           462 +         auto tick = std::chrono::high_resolution_clock::now();
                                                           463 +     #endif
                                                           464 +
462       // create model (master reads, workers receive!)   465       // create model (master reads, workers receive!)
463 -     Model model = create_model(rank);                466 +     Model model = create_model(rank, argc, argv);
464                                                        467
465       // create graph (only the master loads data and edge   468       // create graph (only the master loads data and edge
```

This issue is related to MPI, Hybrid and Bonus submissions!

Outline

1. Sequential code analysis
2. Improved sequential code analysis
3. OpenMP
4. Message Passing Interface (MPI)
5. Hybrid (OpenMP + MPI)
6. Bonus
7. **Conclusion**

Conclusion

- ***OMP*** gives a speedup almost identical to the Amdahl‘s law: **50-52.5x**. With the further improvements to the code, it‘s possible to get a speedup of **90x**.

- ***MPI*** decreases the speedup. However, we are testing this with a single compute node. We would see the benefits on a real cluster of different compute nodes. Still, the communication overhead is a problem that must be addressed. The speedup achieved is **23x**. With the further improvements, it is possible to achieve a higher speedup of **28x**.

- With the ***Hybrid*** implementation, I gained some speedup by making use of the shared memory of each compute node in themselves: **44x**. Decreasing the communication overhead, a speedup of **75x** could be achieved!

- Applying ***SIMD*** actually decreases the speedup to **20x**. However, we compensated this by decreasing the communication overhead! The final speedup achieved is **44x**.

Q & A

Should you have any questions, please do not hesitate to ask ☺