miniVite Strong Scaling

GeekPie_HPC, ShanghaiTech

Processes-per-node Threads-per-process

Before reading the source code, we use a black-box autotuning program to find the best processes-per-node and threads-per-process. To run faster, we set the threshold to 0.15 so that miniVite would stop after 2 iterations. The result is shown in the following table.

PPN	OMP_NUM_THREADS	Clustering
20	1	100.445
20	2	102.023
18	4	108.072
20	4	108.56
18	1	108.96
19	4	111.123
14	3	112.578
17	1	113.688
13	4	114.414
12	3	132.041
14	1	133.937
11	4	135.394
13	1	144.941
10	2	146.281
12	1	157.053
11	1	178.026
10	1	189.759

After digging into the source code, we find that in every iteration, miniVite:

1. Call `fillRemoteCommunities` to get information about communities on remote processes.
2. Call `distExecuteLouvainIteration` for every vertex.
3. Call `distUpdateLocalCinfo` to update the information of communities on the local process.
4. Call `updateRemoteCommunities` to update the information of communities on remote processes.

5. Check whether the clustering is converged.

OpenMP parallelism is used in `distExecuteLouvainIteration` and `distUpdateLocalCinfo`. So we have to use `omp atomic` to update the information of communities on local process. This may be inefficient.

For MPI parallelism, we use `fillRemoteCommunities` and `updateRemoteCommunities` to synchronize between processes. There is no synchronization in `distExecuteLouvainIteration` and `distUpdateLocalCinfo`.

Therefore, if we set PPN to 20 and set OMP_NUM_THREADS to 1, every core will have 1 process to work on, and there will be no synchronization cost in distExecuteLouvainIteration and distUpdateLocalCinfo. So miniVite will run faster.

Because we have assigned 20 processes to each node, and each process needs to store a complete list of nodes of the graph and a list of the edges which belong to its communities, the program requires a large amount of memory, especially when the node cluster size is small.

In our experiments, miniVite crashes at startup when the number of nodes is less than 13. So we choose 13 as the startup node number.

Then we add nodes one by one until the node number reaches 20.

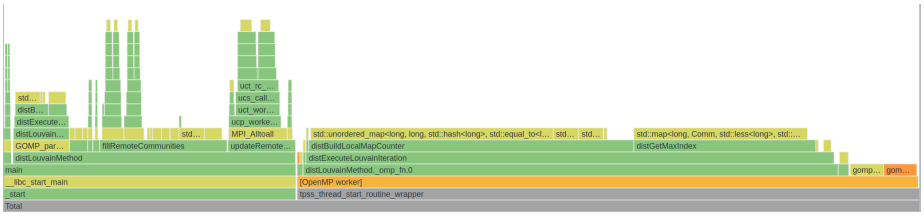
In conclusion, we use 20 processes per node and 1 thread per process to run miniVite from 13 nodes to 20 nodes.

Profiling and Performance Improvements

Before we run strong-scaling experiments, we profile miniVite to find the performance bottleneck.

We use `vtune` to profiling the original miniVite program.

The following figures show the performance analysis of running miniVite on `com-orkut.ungraph.bin` on a single node. (Graph `com-orkut.ungraph.bin` and graph `com-friendster.ungraph.bin` are both social network graphs, and `orkut` is much smaller than `friendster`. So we profile it first.)



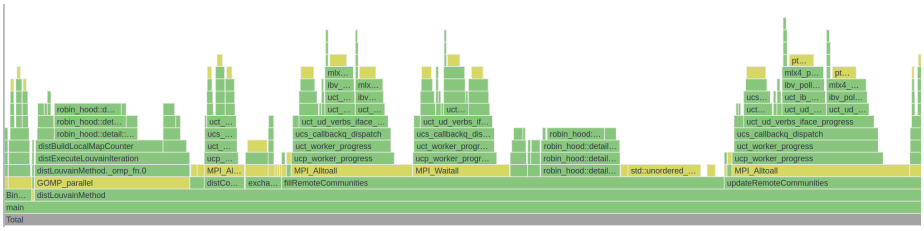
Function / Call Stack	CPU Time ▼	Module	Function (Full)	Sourc
std::unordered_map<long, long, std::hash<long>, std::equal_to<long>, st	1218.343s	miniVite	std::unordered_map<long, long, std::hash<long>, std::equal_to<long>, std::allocator<s…	unordered
std::map<long, Comm, std::less<long>, std::allocator<std::pair<long cons	869.893s	miniVite	std::map<long, Comm, std::less<long>, std::allocator<std::pair<long const, Comm>>>:…	stl_map.h
pthread_spin_lock	244.261s	libpthread.so.0	pthread_spin_lock	
gomp_team_barrier_wait_end	160.719s	libgomp.so.1	gomp_team_barrier_wait_end	bar.c
gomp_simple_barrier_wait	153.000s	libgomp.so.1	gomp_simple_barrier_wait	simple-bar
distBuildLocalMapCounter	134.418s	miniVite	distBuildLocalMapCounter(long, long, std::unordered_map<long, long, std::hash<long…	dspl.hpp
std::unordered_map<long, long, std::hash<long>, std::equal_to<long>, st	128.247s	miniVite	std::unordered_map<long, long, std::hash<long>, std::equal_to<long>, std::allocator<s…	unordered
std::unordered_map<long, long, std::hash<long>, std::equal_to<long>, st	120.096s	miniVite	std::unordered_map<long, long, std::hash<long>, std::equal_to<long>, std::allocator<s…	unordered
std::unordered_map<long, long, std::hash<long>, std::equal_to<long>, st	103.412s	miniVite	std::unordered_map<long, long, std::hash<long>, std::equal_to<long>, std::allocator<s…	unordered
mlx4_poll_cq	82.521s	libmlx4-rdmav34.so	mlx4_poll_cq	cq.c
_int_malloc	81.689s	libc.so.6	_int_malloc	

We can see that most CPU time was spent on `std::map` and `std::unordered_map`.

And we know that the STL implementations of the two maps are very slow, so we replaced them with a faster implementation: robin-hood-hashing.

This change gives miniVite a huge performance boost. When using a 20-node cluster to analyze the graph `com-friendster.ungraph.bin`, the runtime improves from about 460 seconds to about 110 seconds.

Then we profile miniVite again with a 20-node cluster and the graph `com-friendster.ungraph.bin`. The following figure shows the result:



Now we can see that most CPU time is used for cross-process synchronization of MPI, and the efficiency has reached the ideal value.

Our modifications involve only the replacement of the data structure implementation. We did not modify the Louvain Method itself, so the correctness of the algorithm is not affected. The Modularity before and after the modification is also unchanged.

Arguments

OpenMPI compile arguments

```
openmpi@4.1.4~atomics~cuda~cxx~cxx_exceptions~gpfs~internal-hwloc~java
~legacylaunchers~lustre~memchecker+romio+rsh~singularity+static+vt+wrapper-rpath
fabrics=ucx schedulers=non
```

We use `fabrics=ucx` to enable the InfiniBand support.

MiniVite compile commands

```
mpic++ -std=c++17 -g -DCHECK_NUM_EDGES -DPRINT_EXTRA_NEDGES \
       -march=native -ffast-math -mprefer-vector-width=256 -fopenmp -Ofast \
       -c -o main.o main.cpp
mpic++ main.o -fopenmp -o miniVite
```

We have made some changes to the MiniVite code described in the `Profiling and Performance Improvements` section.

We use `-march=native -ffast-math -mprefer-vector-width=256 -Ofast` for efficiency.

Our running command

```
mpirun --hostfile ./hostfile -n 400 -map-by core --bind-to core \
       miniVite -f com-friendster.ungraph.bin -b -t 0.0015
```

`-map-by core --bind-to core` binds one process to one core.

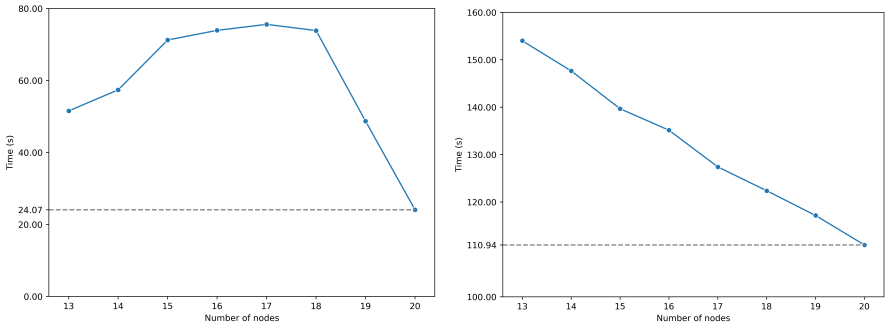
`-b` attempts to distribute approximately equal number of edges among processes. So the workload of each process can be relatively close.

`-t 0.0015` reduces the number of iterations without significantly affecting the modularity.

Results

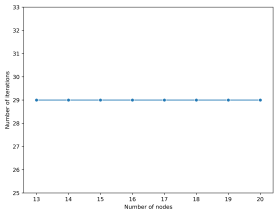
The minimum runtime we get is 110.941 seconds.

- Modularity 0.588858 remains unchanged in all experiments.
- Iterations 29 remains unchanged in all experiments.

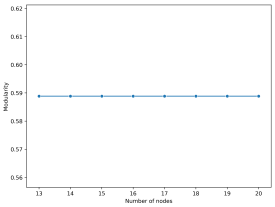


(a) Loading (b) Clustering

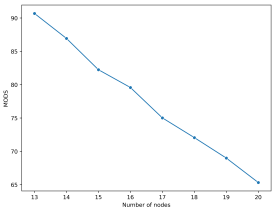
Figure 1: Strong Scaling Running Time



(a) Number of iterations



(b) Modularity



(c) MODS