miniVite

ShanghaiTechU GeekPie_ HPC

Establishing baseline performance

We use spack and NFS to setup environments and share data between compute nodes and the head node.

How are these two input graphs different?

Orkut dataset is from a online social network, which allows users form a group that other members can join in.

Webbase dataset has been obtained from the 2001 crawl performed by the WebBase crawler.

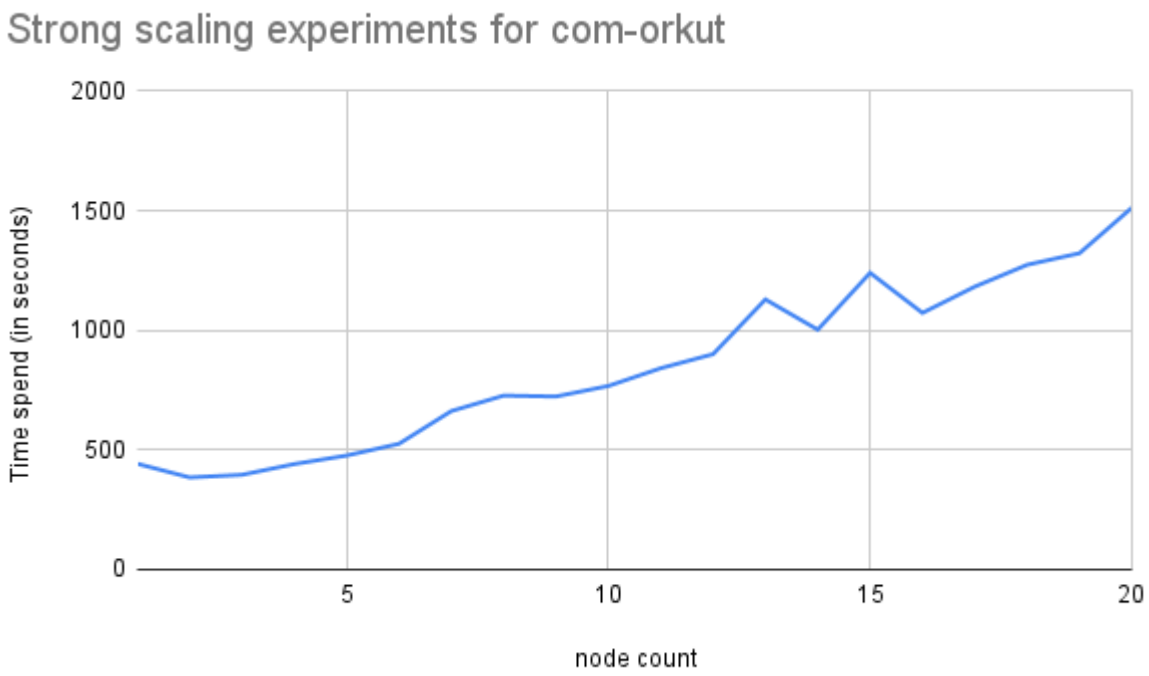
Orkut dataset has fewer vertices and edges compared to the webbase dataset. Orkut dataset has high modularity, which means there are dense connections between the nodes within modules but sparse connections between nodes in different modules(Wikipedia).

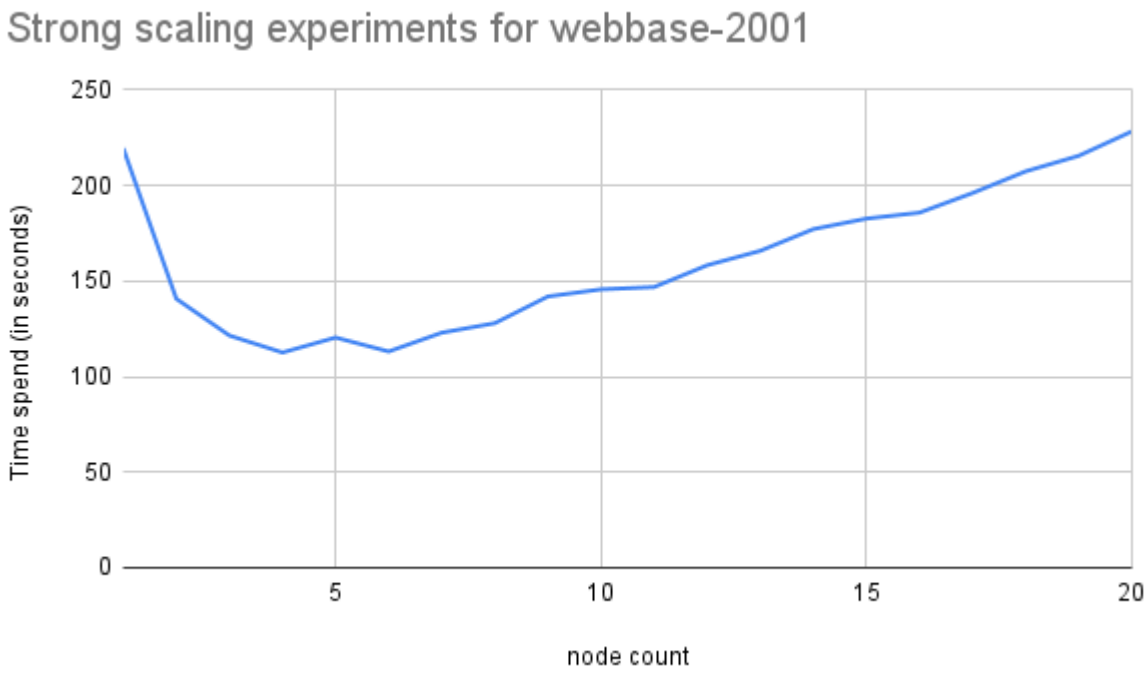
	`com-orkut.ungraph.bin`	`webbase-2001.bin`
Size	3.6G	31G
Modularity(miniVite output)	0.633749	0.458129
Running time	Longer	Shorter
Number of vertices	3072441	118142155
Number of edges	234370166	1985689782
Maximum number of edges	21807018	125244164
Average number of edges	1.17185e+07	9.92845e+07
Expected value of X^2	1.62778e+14	1.01941e+16
Variance	2.54547e+13	3.36641e+14
Standard deviation	5.04526e+06	1.83478e+07

What arguments did you choose to run miniVite

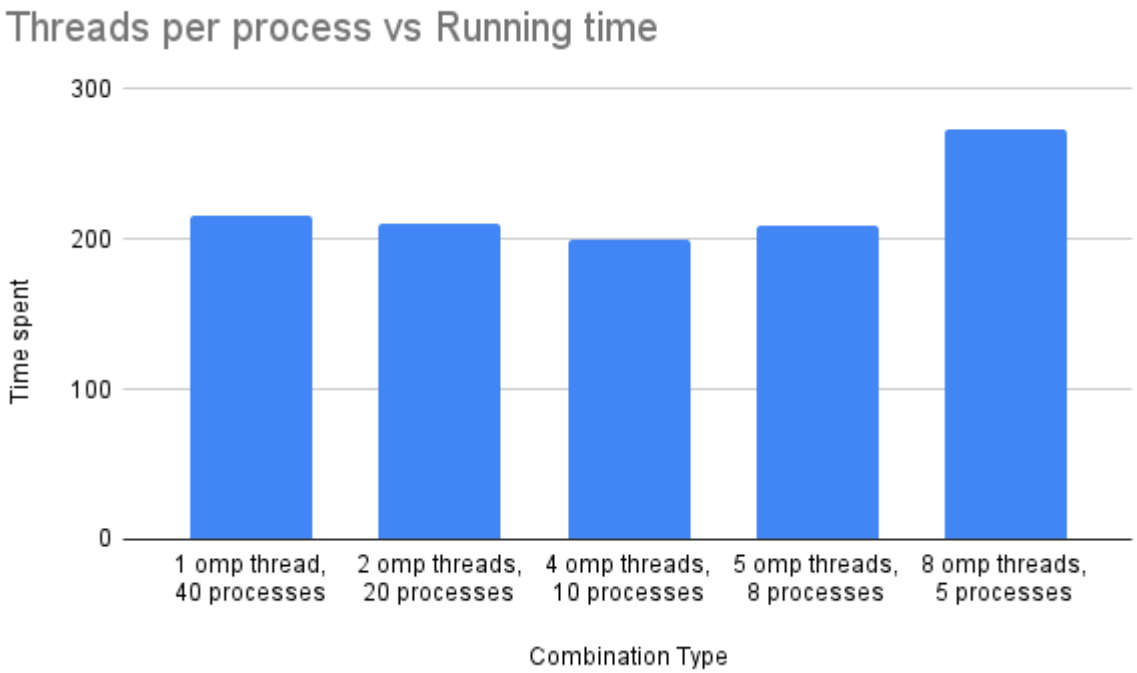
We use `spack install minivite`. The default MPI is openMPI. The default compiler is gcc-8 on CentOS.

```
#!/bin/fish
touch output;
for i in (seq 1 20);
    rm host$i;
    touch host$i;
    for j in (seq 1 $i);
        echo "node-$j slots=20" >> "host$i"
    end;
    echo "webbase-$i running...";
    echo "webbase-$i running..." >> output;
    mpirun --hostfile host$i -n (math 20 \* $i) miniVite -f /share/webbase-2001.bin >>
output;
    echo "ungraph-$i running...";
    echo "ungraph-$i running..." >> output;
    mpirun --hostfile host$i -n (math 20 \* $i) miniVite -f /share/com-
orkut.ungraph.bin >> output;
end;
```





Does increasing the number of OpenMP threads help the performance?



In our case, increasing the OpenMP threads to 3 maximizes our performance. The phenomenon may be caused by the imbalance of jobs between MPI processes. When an MPI process hits the MPI_Barrier or MPI_Waitall, it has finished its calculation and has to wait for other MPI processes, then it can donate its CPU time to other processes. By increasing OpenMP threads, the MPI process which has the heaviest work load among all MPI processes can utilize as many

free CPU resources as possible. However, setting OpenMP threads too high may cause overhead, such as context switch, which will reduce the performance.

Performing further optimizations

Compare baseline performance with the improved version

Open MPI

We use infinity band to get better performance.

OpenMPI variant: `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`

miniVite

We have tried `-DUSE_32_BIT_GRAPH`, but it crashed.

```
terminate called after throwing an instance of 'std::length_error'
  what():  vector::_M_fill_insert
```

Furthermore, we have tried `-DUSE_MPI_ACCUMULATE` and the other two options, but the performance is not improved (see the spack build script).

```
# Copyright 2013-2022 Lawrence Livermore National Security, LLC and other
# Spack Project Developers. See the top-level COPYRIGHT file for details.
#
# SPDX-License-Identifier: (Apache-2.0 OR MIT)

from spack.package import *


class Minivite(MakefilePackage):
    """miniVite is a proxy application that implements a single phase of
    Louvain method in distributed memory for graph community detection.
    """

    tags = ["proxy-app", "ecp-proxy-app"]

    homepage = "https://hpc.pnl.gov/people/hala/grappolo.html"
    git = "https://github.com/Exa-Graph/miniVite.git"

    version("develop", branch="master")
    version("1.0", tag="v1.0")
    version("1.1", tag="v1.1")

    variant("openmp", default=True, description="Build with OpenMP support")
    variant("opt", default=True, description="Optimization flags")
    variant("mode",default='default',description="mode",values=
('collective','sendrecv','rma','default','rma_accu'))
    variant("omp_schedule", default=False, description="Enable OMP schedule")
    variant("use_32_bit_graph", default=False, description="Use 32bit graph")

    depends_on("mpi")

    @property
    def build_targets(self):
        targets = []
        cxxflags = ["-std=c++11 -g -DCHECK_NUM_EDGES -DPRINT_EXTRA_NEDGES"]
        ldflags = []

        if "+openmp" in self.spec:
            cxxflags.append(self.compiler.openmp_flag)
            ldflags.append(self.compiler.openmp_flag)
        if "+opt" in self.spec:
            cxxflags.append(" -O3 ")
        if self.spec.variants['mode'].value == 'collective':
            cxxflags.append("-DUSE_MPI_COLLECTIVES")
        elif self.spec.variants['mode'].value == 'sendrecv':
            cxxflags.append("-DUSE_MPI_SENDRECV")
        elif self.spec.variants['mode'].value == 'rma':
            cxxflags.append("-DUSE_MPI_RMA")
```

```
        elif self.spec.variants['mode'].value == 'rma_accu':
            cxxflags.append("-DUSE_MPI_RMA -DUSE_MPI_ACCUMULATE ")

        if "+omp_schedule" in self.spec:
            cxxflags.append("-DOMP_SCHEDULE_RUNTIME")
        if "+use_32_bit_graph" in self.spec:
            cxxflags.append("-DUSE_32_BIT_GRAPH")

        targets.append("CXXFLAGS={0}".format(" ".join(cxxflags)))
        targets.append("OPTFLAGS={0}".format(" ".join(ldflags)))
        targets.append("CXX={0}".format(self.spec["mpi"].mpicxx))

        return targets

    def install(self, spec, prefix):
        mkdirp(prefix.bin)
        if self.version >= Version("1.1"):
            install("miniVite", prefix.bin)
        elif self.version >= Version("1.0"):
            install("dspl", prefix.bin)
```

Final minVite variant: `minivite@1.1+openmp+opt mode=rma`

Run

We generate the appfile and the rankfile based on the `hostfile` by the python script mentioned above. We assume that the average load of the MPI process in every NUMA is close. We want to find the number of MPI processes on one node (which has two NUMAs) that maximize the performance. We also find that with the USE_MPI_RMA flag open, the performance grows with the increase of nodes, so we will use 20 nodes to find the best PPN. After some trials, we found PPN = 14 is best for webbase-2001, 20 nodes. We generated a series of rankfiles and appfiles from 1 node to 20 nodes.

`rankfile`, `appfile` generator

```
ranks = []
apps = []
rank_id = 0
PPN = 14
with open('hostfile') as h:
    data = h.read()
    data = data.strip().split('\n')

while rank_id < len(data) * PPN:
    ranks.append(f"rank {rank_id}={data[rank_id // PPN]} slot={0 if rank_id % PPN <
PPN / 2 else 1}:0-9")
    apps.append(f"-np 1 /share/run.sh {0 if rank_id % PPN < PPN / 2 else 1} {PPN}")
    rank_id += 1
with open('rankfile', 'w') as f:
    f.write('\n'.join(ranks))

with open('appfile', 'w') as f:
    f.write('\n'.join(apps))
```

/share/run.sh

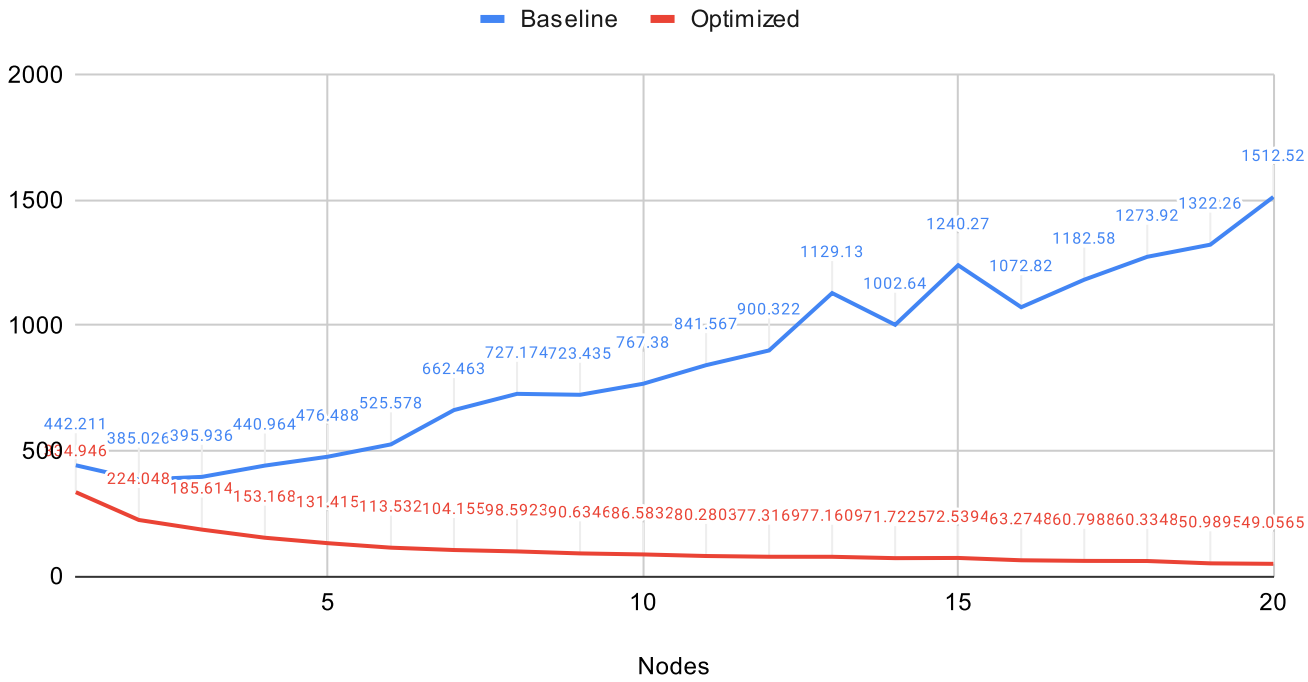
```
#!/bin/bash
export OMP_PLACES="sockets"
export OMP_PROC_BIND="close"
export OMP_NUM_THREADS=3

numactl --membind=$1 miniVite -f /share/webbase-2001.bin -b -r $2
```

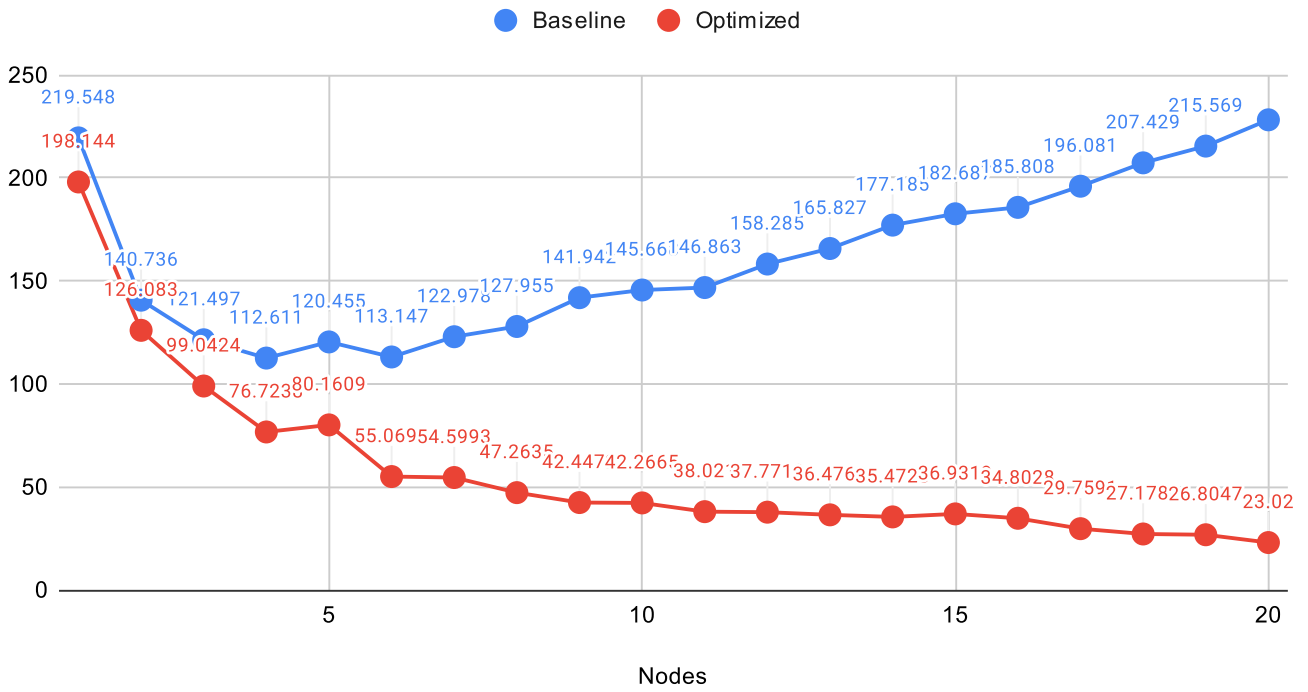
Command:

```
mpirun --report-bindings --mca mpi_leave_pinned 1 --hostfile hostfile --rankfile
rankfile --app appfile
```

Baseline and Optimized (orkut)



Baseline and Optimized (webbase)



Best performance: `webbase-2001` 23.02s, `com-orkut.ungraph` 49.05s

Does your set of options affect the output quality (expressed via modularity and MODS) in any way?

We found that there may be different numbers of iterations in different runs, so the modularity may be different.

Our modifications do not affect the output quality, because these modifications still produce the same intermediate result, which means our modifications are deterministic. For example, we enable the `USE_MPI_RMA` macro, which enables the RMA operations, although different MPI calls are used, the result stays the same.