Project 3 All-pairs Shortest Path

Group members

- Abdelrahman Kamel (ak883)
- Hee Jung Ryu (hr99)

Timing Plots

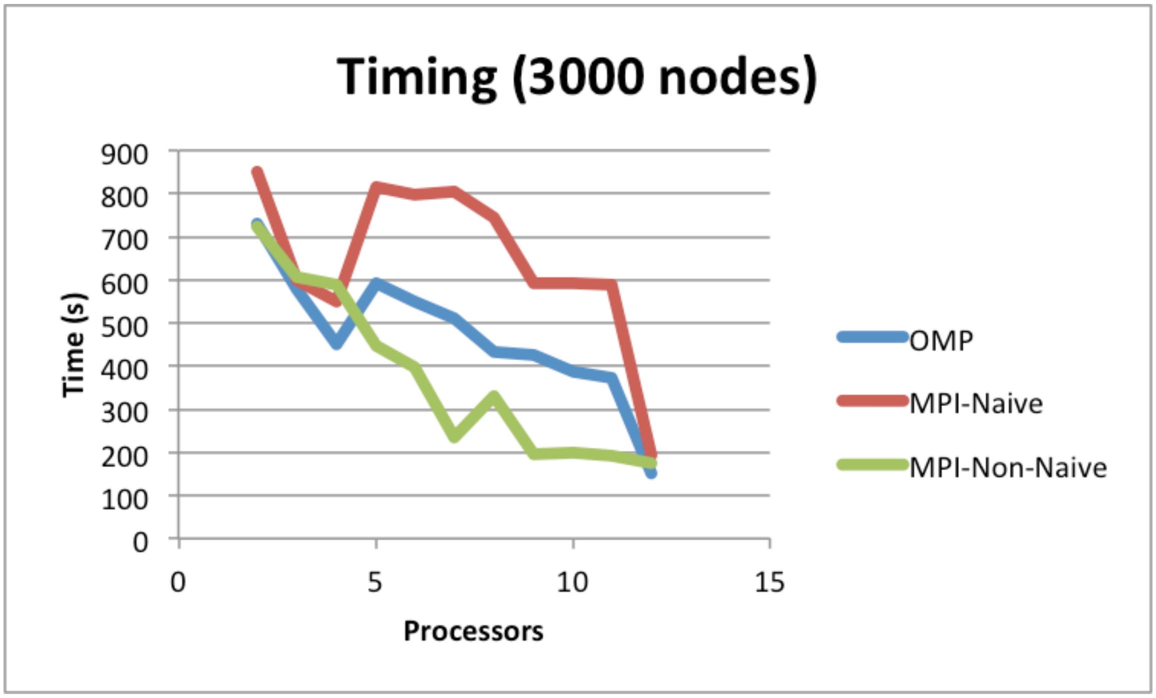


Figure 1: In the above plot we show how each implementation, OMP, MPI (naive), and MPI (non naive) preform by varying the number of processors for a fixed problem size of 3000 nodes.

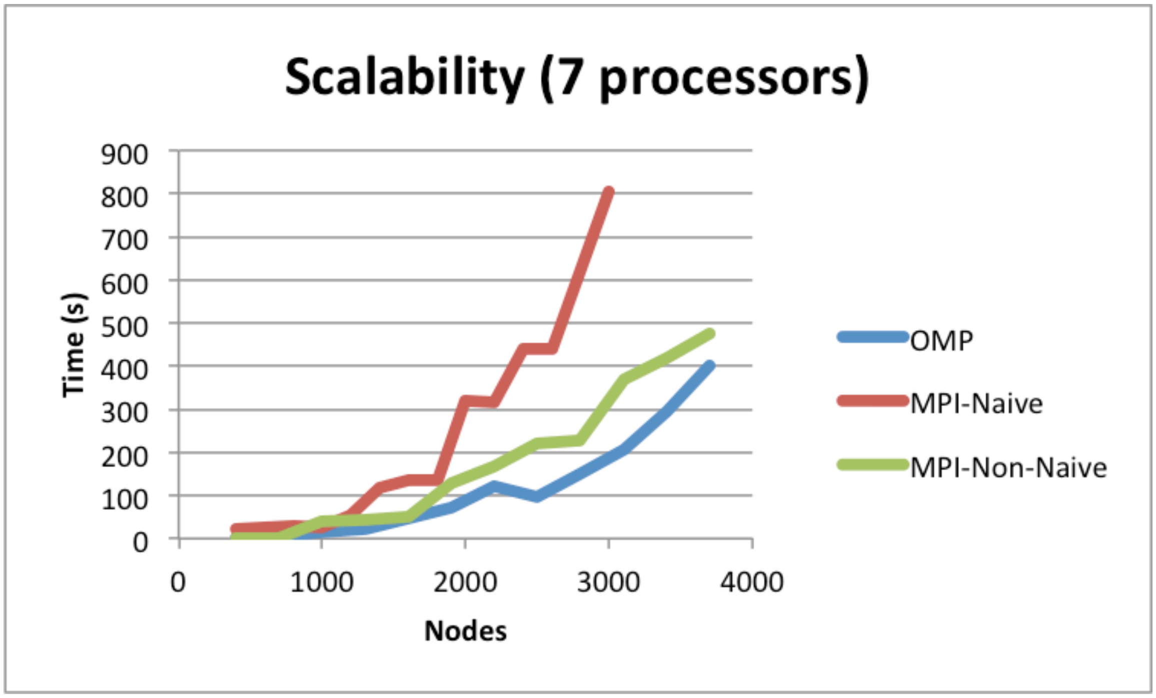


Figure 2: In this graph we show a weak study of all three implementations by varying the number of nodes. The number of processors is fixed at 7 processors. We chose 7 processors because we saw the best performance from all three implementations at that point.

Timing Plot Conclusion

In figure 1 we see that the best preforming implementation is non-naive MPI. The worst performance is obtained from the naive version. The reason why the naive version preforms badly is due to the memory scalability of the implementation. The naive implementation stores the entirety of the distance matrix at every processor and then preforms the calculations. Storing the entire matrix at every processor is very costly in terms of cache efficiency. A matrix of that size results in many cache hit/misses which would explain the performance we are seeing for the naive implementation.

Alternatively instead of keeping the entire distance array at every processor we rotate sections of the entire distance matrix around every processor in a 1D ring fashion. In this implementation we see in figure 1 that the work being done at every processor is much less, which results in better performance.

In figure 2 we can see the cost of communication as the node size increases. This becomes very obvious in the naive MPI implementation. By storing a section of the matrix we see a drastic improvement. Given more time to analyze the code we can also conclude that the non-naive MPI implementation will out preform OMP. After 4000 nodes this affect becomes noticeable.

Speedup Plots

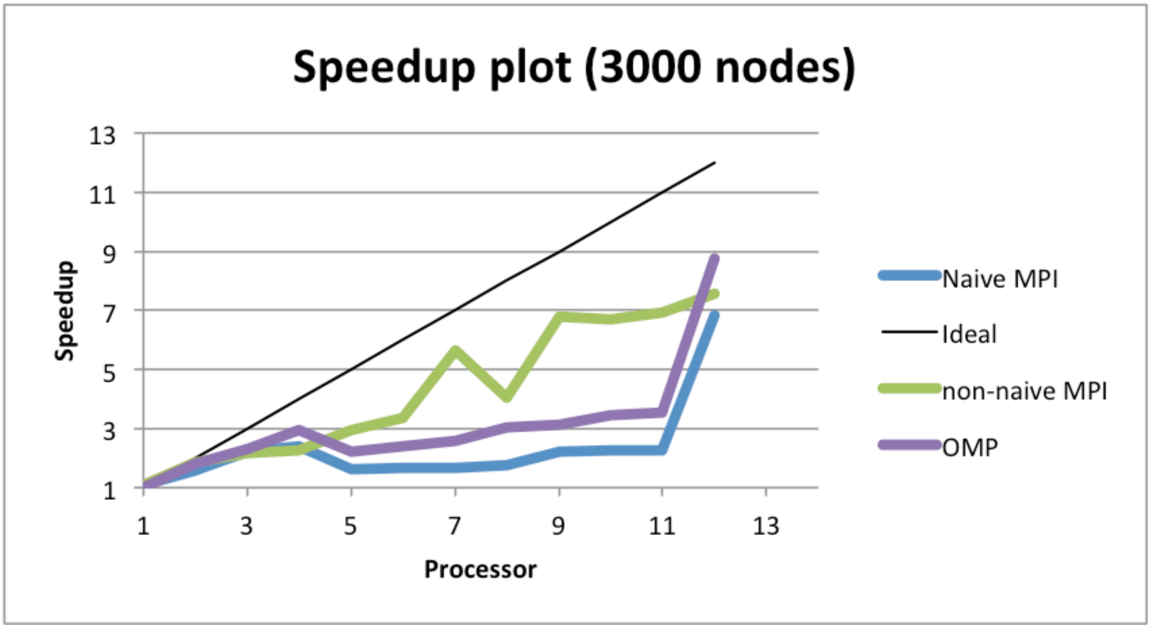


Figure 3: In this plot we graph the speedup of all three implementations. The best performance is seen in the non-naive MPI implementation. OMP and the naive implementation preform very closely in terms of speedup.

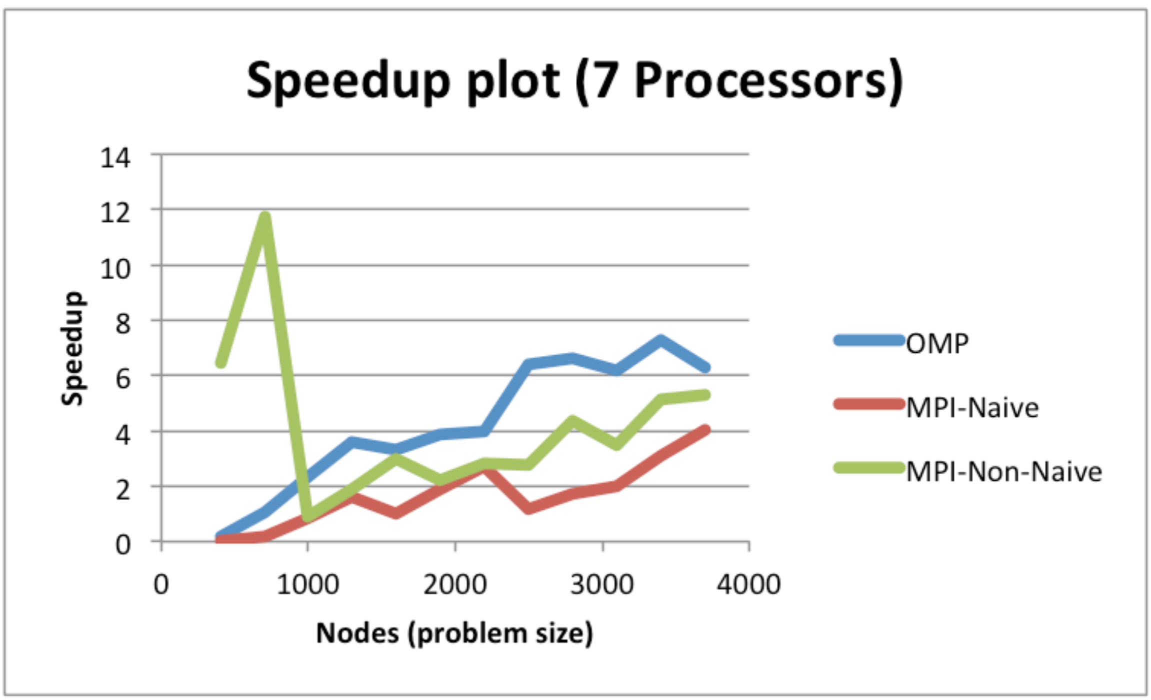


Figure 4: In this plot we see something very interesting; Non-naive MPI has a speedup of almost 12 for nodes less then 1000. I was unable to add a "ideal" trendline since the problem size is in the thousands.

Speedup plot Conclusion

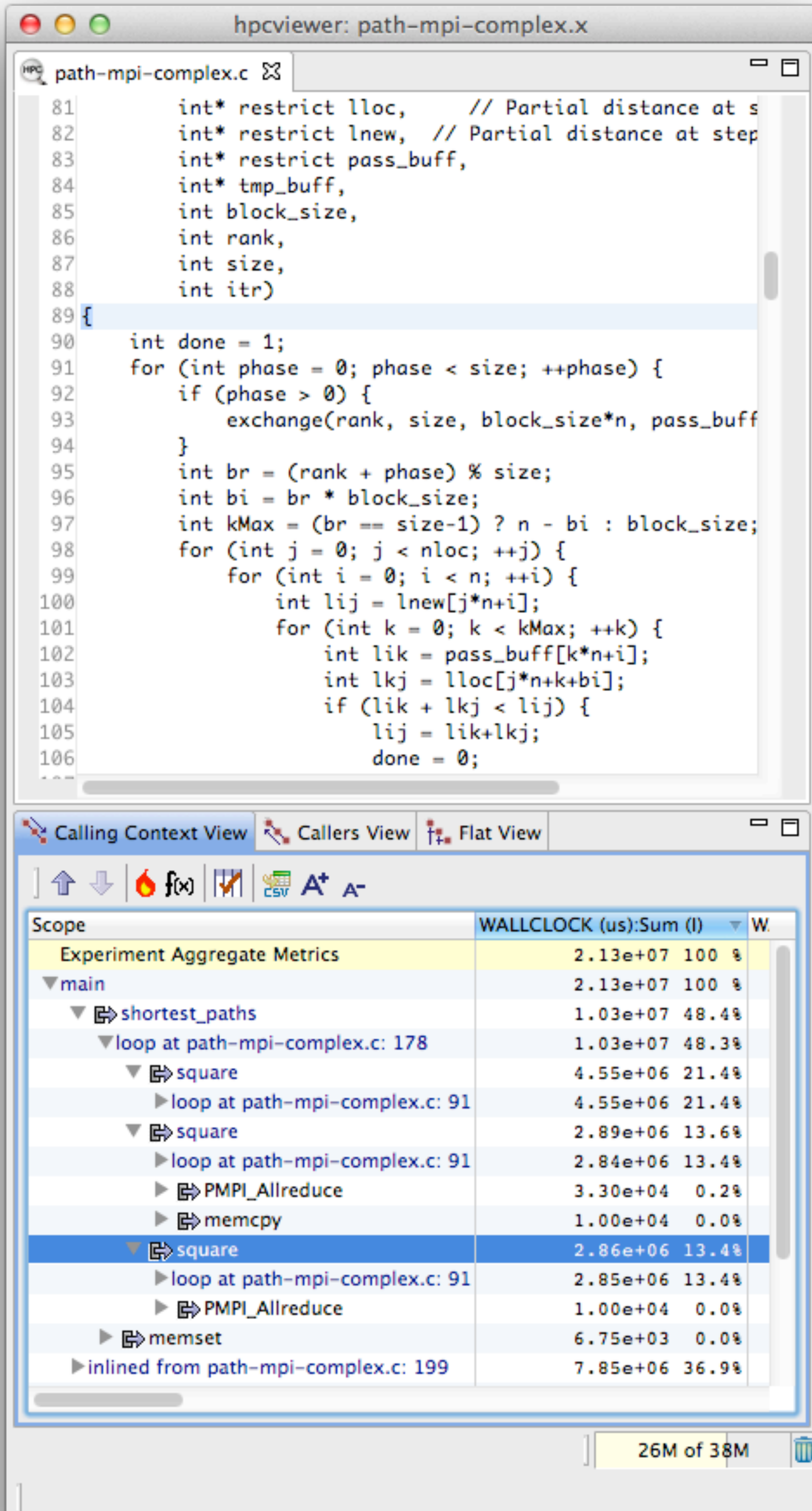
In figure 3, we see that the non-naive MPI implementation approaches the ideal p-time until speedup of 7. After 7 we start to see the gap between all three implementions. In the second graph it is infesting to know that the non-naive implementation preforms well for problem sizes lower then 1000. Its also interesting to note that after 4000 nodes we see OMP preforming worse than the non-naive MPI implementation.

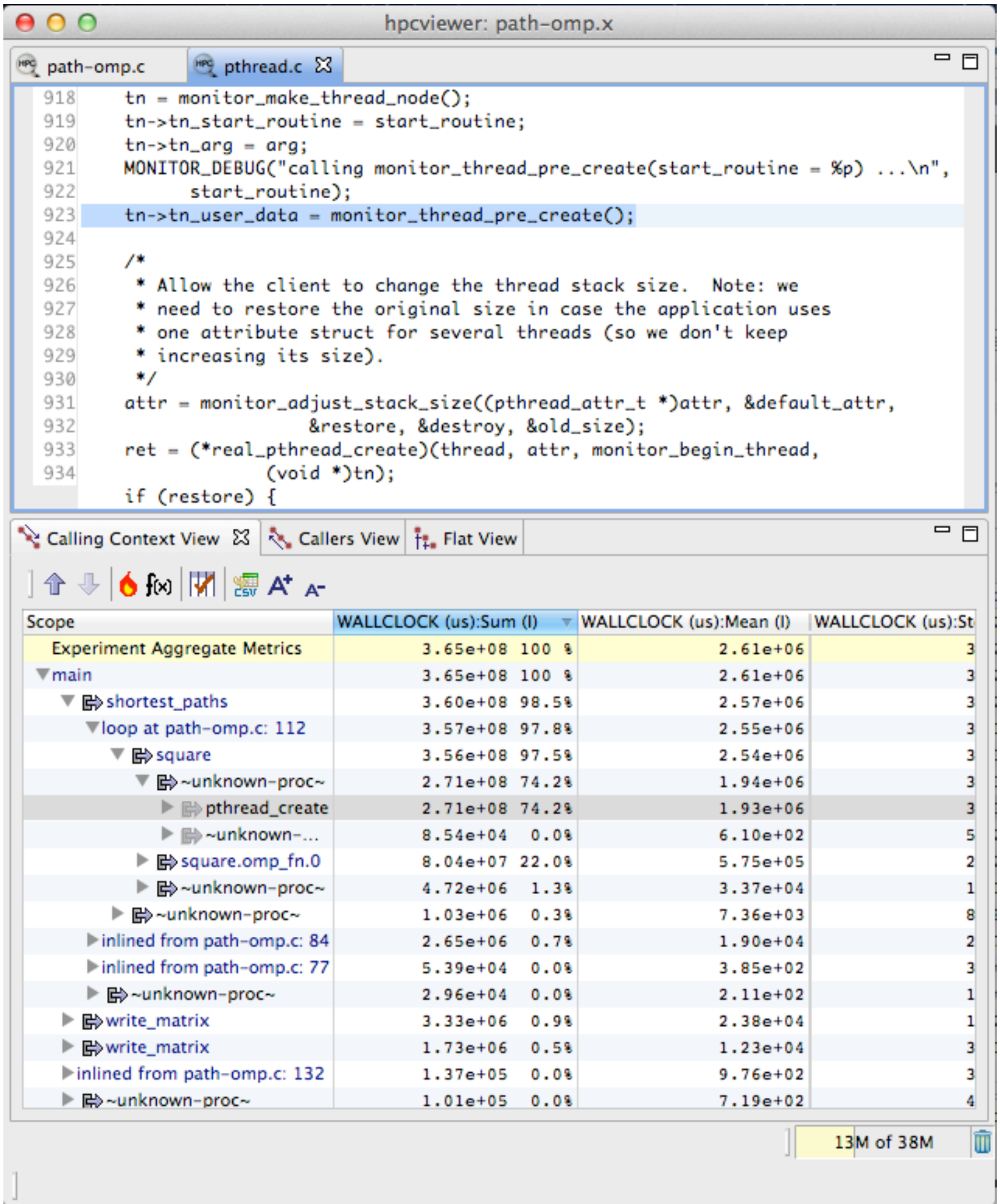
Code breakdown

HPCVIEWER

After using HPCToolkit to profile our code we see that the non-naive version has a bottle neck at square function as expected. Using 7 processes, we see that the communication cost is completely overlapped by the computational time. We also preformed an analysis to compare our results with the naive implementation and noticed that approximately 30% of the total time is spent in communication.

From the graphs and profiling we see that as the problem size increases for the naive version, half total time being spent is in computation and the other half is in communication. As for OMP approximately 22% of the time goes into actual computation and the rest goes into thread creation.


hpcviewer: path-mpi-complex.x
path-mpi-complex.c
int* restrict lloc,   // Partial distance at s
int* restrict lnew, // Partial distance at step
int* restrict pass_buff,
int* tmp_buff,
int block_size,
int rank,
int size,
int itr)
{
int done = 1;
for (int phase = 0; phase < size; ++phase) {
if (phase > 0) {
exchange(rank, size, block_size*n, pass_buff
}
int br = (rank + phase) % size;
int bi = br * block_size;
int kMax = (br == size-1) ? n - bi : block_size;
for (int j = 0; j < nloc; ++j) {
for (int i = 0; i < n; ++i) {
int lij = lnew[j*n+i];
for (int k = 0; k < kMax; ++k) {
int lik = pass_buff[k*n+i];
int lkj = lloc[j*n+k+bi];
if (lik + lkj < lij) {
lij = lik+lkj;
done = 0;
Calling Context View
Callers View
Flat View
Scope
WALLCLOCK (us):Sum (I)
Experiment Aggregate Metrics 2.13e+07 100 %
main 2.13e+07 100 %
shortest_paths 1.03e+07 48.4%
loop at path-mpi-complex.c: 178 1.03e+07 48.3%
square 4.55e+06 21.4%
loop at path-mpi-complex.c: 91 4.55e+06 21.4%
square 2.89e+06 13.6%
loop at path-mpi-complex.c: 91 2.84e+06 13.4%
PMPI_Allreduce 3.30e+04 0.2%
memcpy 1.00e+04 0.0%
square 2.86e+06 13.4%
loop at path-mpi-complex.c: 91 2.85e+06 13.4%
PMPI_Allreduce 1.00e+04 0.0%
memset 6.75e+03 0.0%
inlined from path-mpi-complex.c: 199 7.85e+06 36.9%
26M of 38M


hpcviewer: path-omp.x
path-omp.c
pthread.c
tn = monitor_make_thread_node();
tn->tn_start_routine = start_routine;
tn->tn_arg = arg;
MONITOR_DEBUG("calling monitor_thread_pre_create(start_routine = %p) ...\n",
start_routine);
tn->tn_user_data = monitor_thread_pre_create();
/*
* Allow the client to change the thread stack size. Note: we
* need to restore the original size in case the application uses
* one attribute struct for several threads (so we don't keep
* increasing its size).
*/
attr = monitor_adjust_stack_size((pthread_attr_t *)attr, &default_attr,
&restore, &destroy, &old_size);
ret = (*real_pthread_create)(thread, attr, monitor_begin_thread,
(void *)tn);
if (restore) {
Calling Context View
Callers View
Flat View
Scope
WALLCLOCK (us):Sum (I)
WALLCLOCK (us):Mean (I)
WALLCLOCK (us):St
Experiment Aggregate Metrics 3.65e+08 100 % 2.61e+06
main 3.65e+08 100 % 2.61e+06
shortest_paths 3.60e+08 98.5% 2.57e+06
loop at path-omp.c: 112 3.57e+08 97.8% 2.55e+06
square 3.56e+08 97.5% 2.54e+06
~unknown-proc~ 2.71e+08 74.2% 1.94e+06
pthread_create 2.71e+08 74.2% 1.93e+06
~unknown-... 8.54e+04 0.0% 6.10e+02
square.omp_fn.0 8.04e+07 22.0% 5.75e+05
~unknown-proc~ 4.72e+06 1.3% 3.37e+04
~unknown-proc~ 1.03e+06 0.3% 7.36e+03
inlined from path-omp.c: 84 2.65e+06 0.7% 1.90e+04
inlined from path-omp.c: 77 5.39e+04 0.0% 3.85e+02
~unknown-proc~ 2.96e+04 0.0% 2.11e+02
write_matrix 3.33e+06 0.9% 2.38e+04
write_matrix 1.73e+06 0.5% 1.23e+04
inlined from path-omp.c: 132 1.37e+05 0.0% 9.76e+02
~unknown-proc~ 1.01e+05 0.0% 7.19e+02
13M of 38M

Navie MPI Implementation Analysis (path-mpi.c)

MPI APIs used for actual computation:
1. MPI_Allgatherv()
2. MPI_Allreduce()

Both functions have an implicit barrier. Thus, by using the two at the end of the square function ensured that all the processes were on the same page before moving on the next iteration of the square computation.

However, in regards to OMP, it was very inefficient in terms of memory because each process kept the whole matrix l and lnew. In an amortized version with infinite number of nodes with a lot of processes, this implementation is also very inefficient . This is partially due to the two MPI_APIs used requireing all-to-all communication, which is an expensive network communication method.

Non-Naive MPI Implementation Analysis (path-mpi-complex.c)

MPI APIs used for actual computation besides sanity check:
1. MPI_Sendrecv()
2. MPI_Allreduce()

They have an implicit barrier. Thus, by using MPI_Allreduce() at the end of the square function ensured that all the processes were on the same page before moving on the next iteration of the square computation. We also used MPI_Sendrecv() within the nested-loop before copying over the received data to update our reference matrix called pass_buff.