Project final report

Scalable, NUMA-Aware Shared-Memory Data Structures

CS550 – Advanced Operating Systems

Apr 27, 2023

Bo Huang A20496327

Qiming Lyu A20480567

1. Introduction

NUMA (non-uniform memory access) is the phenomenon that memory at various points in the address space of a processor has different performance characteristics [1]. Compared with UMA (uniform memory access) architecture, in which all CPUs share the whole memory, NUMA architecture divides CPUs and memory into several “local parts” called nodes and supports each CPU to access the nearest memory location first. Certainly, it decreases the latency for small data. However, it also causes bad performance for shared-memory data structures since different CPUs will have significantly different latencies to the same memory region if they are from different nodes. In this project, we will implement a few NUMA-aware shared-memory data structures to solve the latency problem due to the NUMA architecture.

2. Background Information

The signal path length from the processor to memory plays an important role when evaluating the speed performance. Take a system with NUMA for example (as Figure 1 shows), NUMA architecture brings memory nearer to processors [2]. Access from a single core to remote memory at other nodes has much higher latency compared to local memory within a node because of longer signal paths, interconnections between nodes and remote memory controllers. As the requirement for speed grows, NUMA systems become popular.

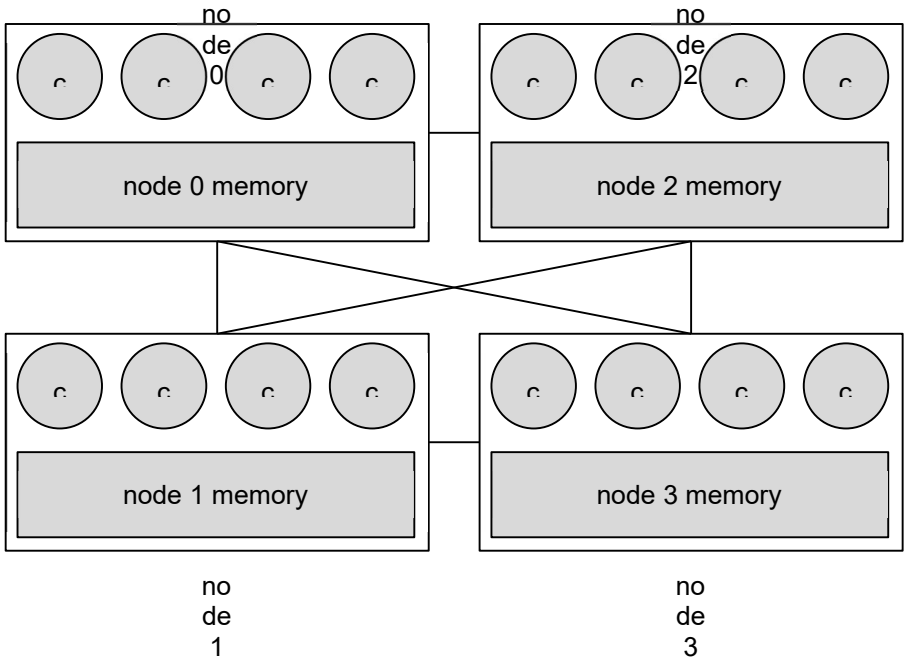


Figure 1, A System with 4 Nodes and 16 Processors

Shared memory under the NUMA architecture can access memory faster, improve system throughput, improve scalability, improve program performance, and improve System reliability. There are many occasions that need to use NUMA shared memory, such as

1. High-performance computing (HPC) environment: In this environment, the use of NUMA shared memory can improve program performance on multi-node systems.

2. Distributed database system: Using NUMA shared memory can speed up data sharing in a distributed database system.

3. Distributed storage system: Using NUMA shared memory can improve the performance of distributed storage systems.

There are many program libraries that use NUMA shared memory, including numactl, Boost, BCL, HCL, etc. NUMA machines provide higher performance when programs run on a single NUMA node. However, when it comes to shared memory, a program may span multiple NUMA nodes, causing performance differences.

3. The design of your NUMA-aware data structures

Data Structure	Operations	Description
singly linked list	Push_front()	Insert the element to the head of the singly linked list.
	Erase()	Erase the element
lock-free singly linked list	Push_front()	Insert the element to the head of the singly linked list.
	Erase()	Erase the element
parallel queue (with locking)	Push()	Add an element to the tail of the queue.
	Pop()	Pops the node at the head of the queue and returns the data stored in the node.
lock-free queue	Push()	Add an element to the tail of the queue.
	Pop()	Pops the node at the head of the queue and returns the data stored in the node.
unordered_map	void emplace(Args &&...args)	insert item into hash table.
	T &get(const Key &key)	find item in table, return val.

	int size(int node)	get size from node
unordered_set	void insert(T val)	insert val into hash set.
	iterator begin()	get set iterator

To ensure that data structures and the data they store are allocated from specific NUMA nodes, a NUMA-aware memory allocator is required. The NUMA-aware memory allocator ensures that memory allocations are allocated from the NUMA node where the process is currently located, avoiding remote access and thus improving performance. In C++, you can use the numa_alloc_onnode() function to allocate on a specific NUMA node.

For the specific implementation of the data structure, you can use a method similar to the following code to allocate:

```
void* ptr = numa_alloc_onnode(size, node);
T* obj = new(ptr) T();
```

Figure 2, allocate memory on the node num

Here, T represents the type of data structure, and the numa_alloc_onnode() function will allocate memory on the node num, and then use the placement new operator to construct objects on this memory block.

In order to determine the NUMA node where the process is currently located, the functions provided by the NUMA API need to be used. Under the Linux system, you can use the numa_get_run_node_mask() function to obtain the NUMA node mask where the current process is located. This function returns a bitmask representing the set of NUMA nodes that the current process can access. In this mask, a 1 in the i-th bit indicates that the i-th NUMA node is available.

```
#include <numa.h>
#include <bitset>

int main() {
  numa_set_localalloc();
  std::bitset<64> node_mask;
  numa_get_run_node_mask(node_mask.data(), node_mask.size());
  int current_node = node_mask._Find_first();
  std::cout << "Current node: " << current_node << std::endl;
  return 0;
}
```

Figure 3, A sample code to get the NUMA node where the current process resides

In the above code, we first use the numa_set_localalloc() function to specify the memory allocator to allocate memory from the local NUMA node. Then use the numa_get_run_node_mask() function to get the NUMA node mask that the current process can access, and convert it to a binary representation via std::bitset. Finally, we use the _Find_first() function of std::bitset to get the position of the first bit that is 1, which is the NUMA node number where the current process is located.

we designed these data structures to improve the reading speed of processes on other nodes by saving a backup to other nodes when inserting data.

```cpp
template<typename T>
class numa_unordered_set {
public:
    numa_unordered_map() {
        int num_numa_nodes = numa_max_node() + 1;
        node_sets_.resize(num_numa_nodes);
    }

    void insert(T val) {
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, 0);
        int cur_node = nearest_numa_node();
        node_sets_[cur_node].insert(val);
        for(int i = 0; i < numa_max_node(); i++) {
            if (cur_node != i) {
                void* ptr = numa_alloc_onnode(size_of (val), i);
                node_sets_[i].insert(ptr);
            }
        }

        MPI_Win_unlock(rank);
    }

    iterator *begin() {
        int cur_node = nearest_numa_node();
        return node_sets_[cur_node];
    }

private:
    int nearest_numa_node() const {
        int cpu = sched_getcpu();
        int node = numa_node_of_cpu(cpu);
        if (node < 0) {
            node = 0;
        }
        return node;
    }

    std::vector<std::unordered_set<T>> node_sets_;
}
```

Figure 4, NUMA unordered set code

Under the NUMA architecture, the latency of each CPU core accessing its own local memory is lower than that of accessing remote memory. Therefore, allocating data structures and the data they store to local NUMA nodes can avoid remote memory accesses and thus improve performance.

4. Experimental Setup (Testbed, software)

Experiment Environment

1. **Chameleon Cloud**
2. **2 NUMA-Node, 16 CPUs**
3. **Ubuntu 20.04 (Linux version 5.4.0-139-generic)**
4. **Boost (libboost 1.71-dev amd64)**
5. **gcc (g++ (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0)**

Software

1. **Vscode**
2. **Clion**

Third party software

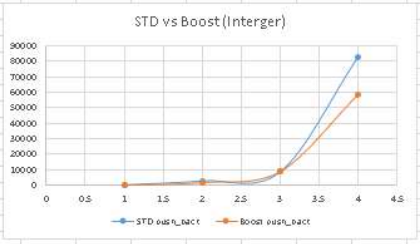
Intel® Memory Latency Checker

5. Experiments

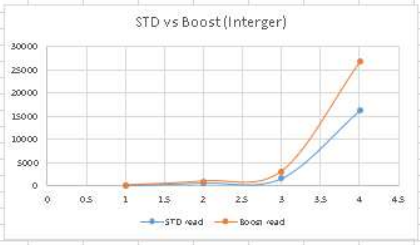
In order to test the impact of numa cross node access on performance, we set up a set of test cases, created a shared memory using the boost::interprocess package, write data in the rank0 process and then performed data reading tests in different processes, in order to make Processes can run on different node nodes we use openmpi. Use mpirun -nq x to start the program and set x processes to run in different node nodes.

5.1 STD vs. Boost

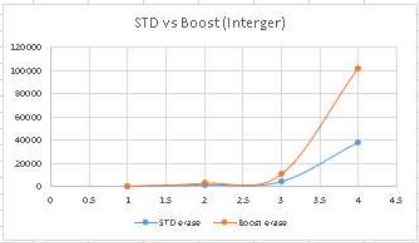
	loop_10C	loop_10C	loop_10C	loop_1000000
STD pusl	297	2873	8618	82354
Boost pu	203	1977	8954	57937



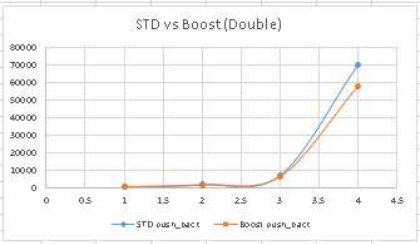
	loop_10C	loop_10C	loop_10C	loop_1000000
STD reac	56	556	1577	16133
Boost rea	99	977	3173	26728



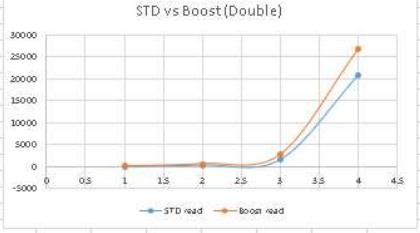
	loop_10C	loop_10C	loop_10C	loop_1000000
STD eras	138	1283	4028	38256
Boost era	374	3039	10547	102012



	loop_10C	loop_10C	loop_10C	loop_1000000
STD pusl	251	1577	7050	70312
Boost pu	199	1376	5935	58140

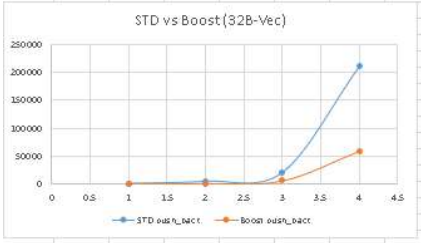


	loop_10C	loop_10C	loop_10C	loop_1000000
STD reac	56	344	1603	20845
Boost rea	99	646	2793	26811

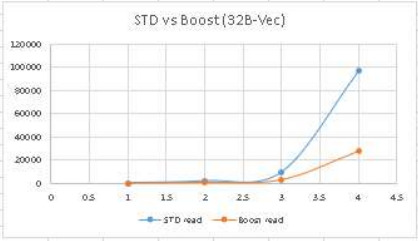


	loop_10C	loop_10C	loop_10C	loop_1000000
STD eras	150	846	3388	41432
Boost era	371	2468	10179	101718

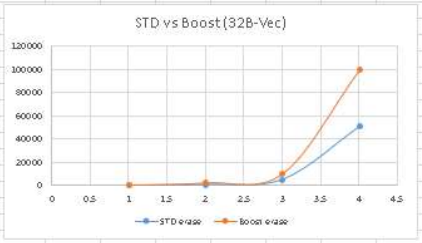
	loop_10C	loop_10C	loop_10C	loop_1000000
STD pusl	755	4708	20953	212562
Boost pu	201	1344	5954	57962



	loop_10C	loop_10C	loop_10C	loop_1000000
STD reac	353	1935	9740	96481
Boost rea	101	659	2837	27393



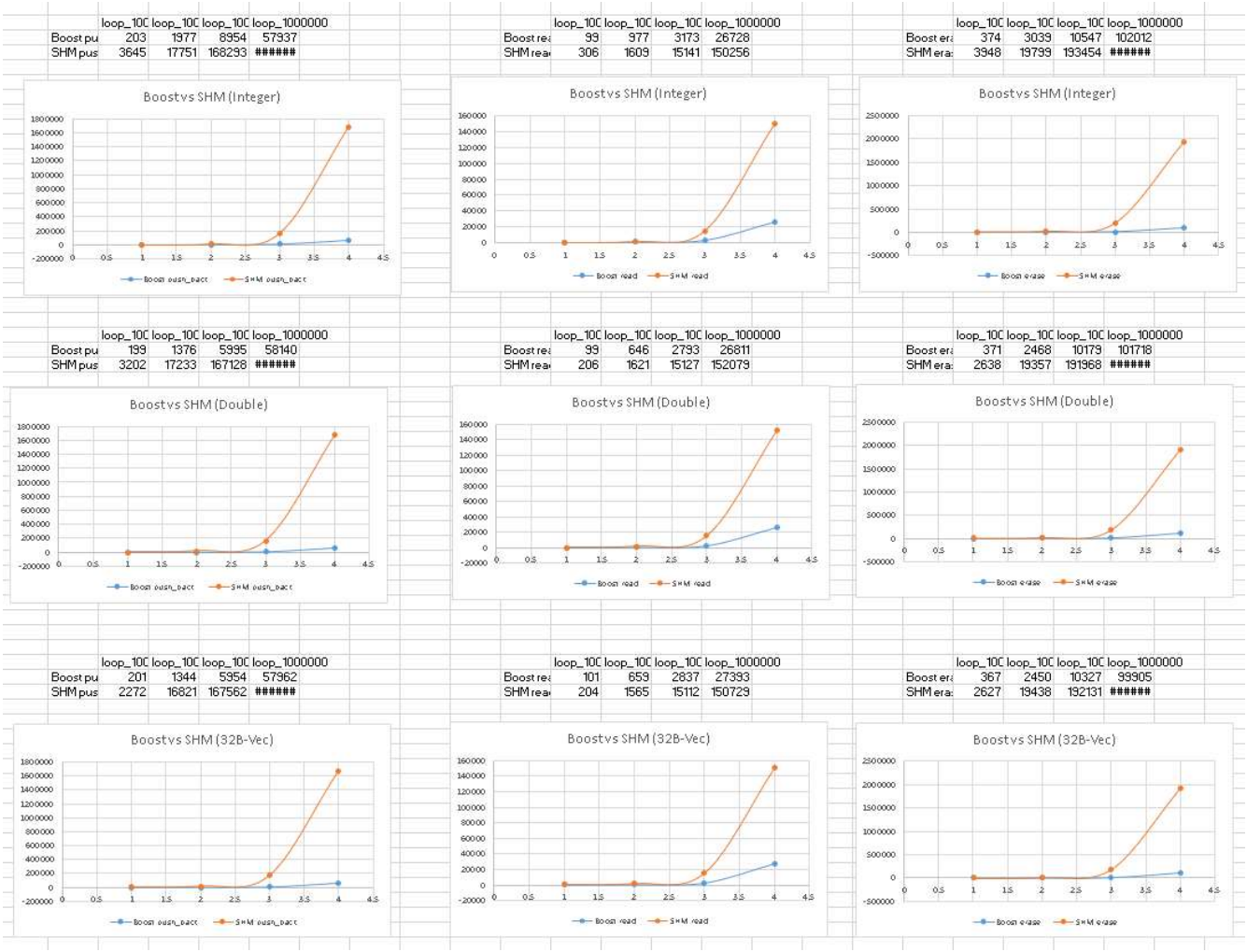
	loop_10C	loop_10C	loop_10C	loop_1000000
STD eras	187	775	5264	51371
Boost era	367	2450	10327	99305



Conclusion

1. **Boost optimizes push_front and read operations (especially push_front)**
2. **The performance of Boost's Erase operation is much worse**
3. **When reading, STD is better in small data, and Boost is dominant in big data**

5.2 Boost local vs. SHM



Conclusion

Shared Memory has much more overhead

5.3 SHM vs. Cross node

5.3.1 How our tests are done.

We use the following function to determine the node id

```
// get the cpu id and node id
int cpu = sched_getcpu();
int node = numa_node_of_cpu(cpu);
```

We first let process 0 delete the old shared memory.

```
if (rank == 0)
{
    bip::shared_memory_object::remove(shm_name.c_str());
}
pComm->barrier();
```

Let process 0 create a new shared memory

```
if (rank == 0)
{
    segment = std::make_unique<boost::interprocess::managed_shared_memory>(
        bip::open_or_create, shm_name.c_str(), MEM_LENGTH);
}
pComm->barrier();
```

Let process 0 build a singly linked list on the shared memory.

```
MySlist *slist;
if (rank == 0)
{
    slist = segment->construct<MySlist>("my_list")(
        segment->get_segment_manager());
}
pComm->barrier();
if (rank != 0)
{
    slist = segment->find_or_construct<MySlist>("my_list")(
        segment->get_segment_manager());
}
```

Then let process 0 perform the insert operation

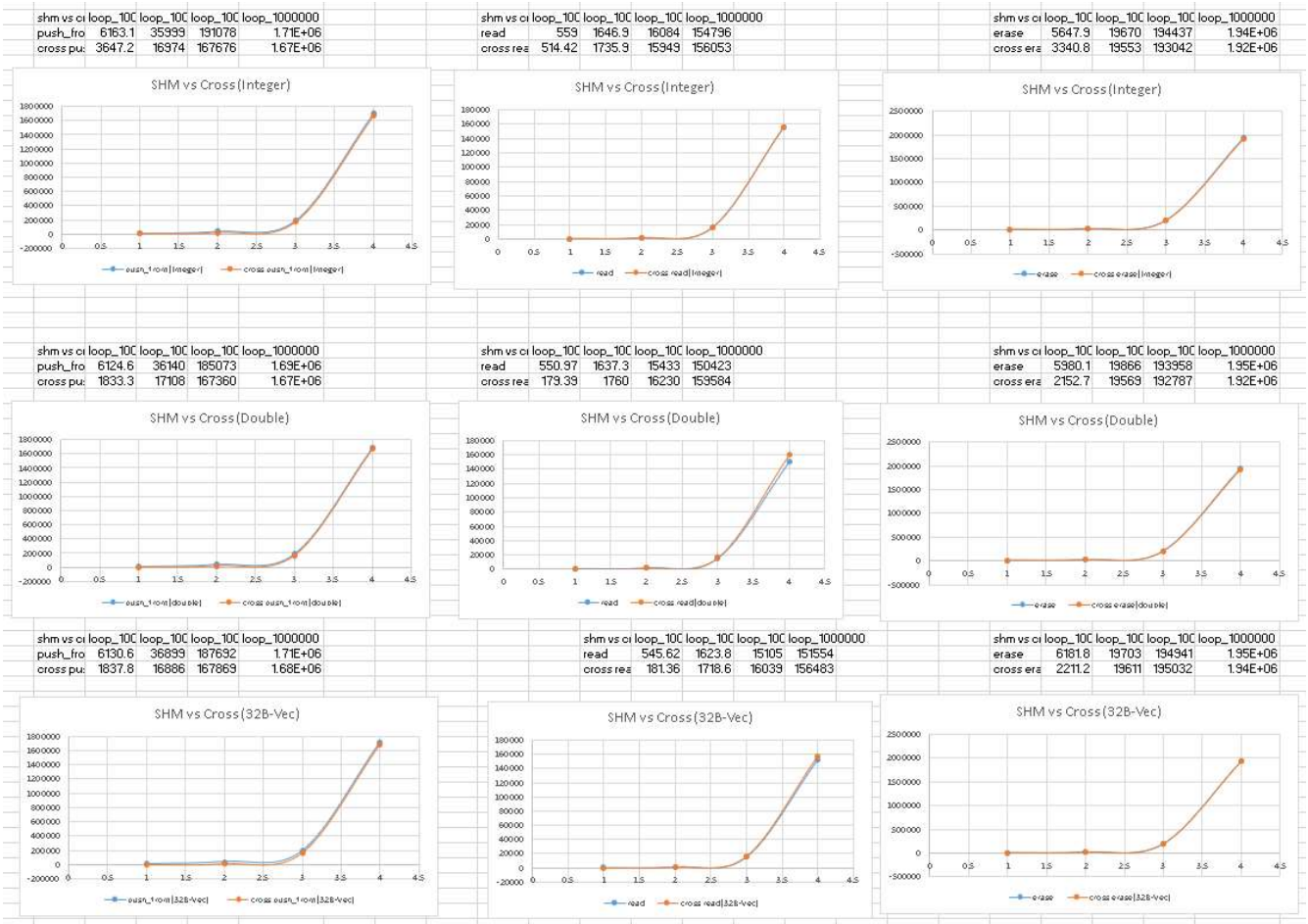
```
if (rank == 0)
{
    std::cout << "[Rank "<< rank << " on node " << node << ":] ";
    std::cout << "build slist on node " << std::endl;
    std::cout << "boost::container::slist on shared_memory push_front ";
    // MEASURE_TIME_MPI(
    start_time = MPI_Wtime();
    for (int i = 0; i < TIMES; ++i)
    {
        slist->push_front(i);
    }
    // );
}
double end_time = MPI_Wtime();
pComm->barrier();
```

Then test the access time of different processes to the singly linked list

```
start_time = MPI_Wtime();
// Distribute get operations across all processes
for (auto it = slist->begin(); it != slist->end(); ++it)
{
    int value = *it;
}
end_time = MPI_Wtime();

elapsed_time = end_time - start_time;
boost::mpi::reduce(*pComm, elapsed_time, max_elapsed_time, boost::mpi::maximum<double>(), 0);
std::cout << "Getting key elapsed time: " << max_elapsed_time << " seconds" << std::endl;
times.push_back(max_elapsed_time * 1000 * 1000);
```

5.3.2 Result



5.3.3 Conclusion

As shown in the above results, almost all the tests obtained almost overlapping curves, which shows that the NUMA-sensitive test was not successful.
I doubt very much that operations based on boost::interprocess shared memory cannot be NUMA-sensitive.

5.4 unordered map

	200,000	100,000	50,000
shared memory insertion	1967.23	993.901	535.574
STL map insertion	92.1762	45.8357	23.3414
STL map read	10.1445	5.03357	2.59301
shared memory local node read 1	90.3509	44.7793	22.7943
shared memory local node read 2	90.1997	44.6488	23.6882
shared memory cross node read 1	93.4369	47.1382	23.6847
shared memory cross node read 2	93.4381	47.1312	23.7509

Figure 10, test result table

Our comparison of 3 sets of data volumes are 200,000, 100,000 and 50,000.

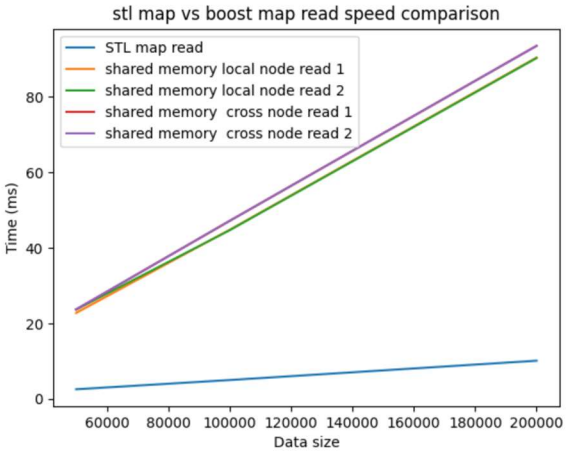


Figure 11, stl vs boost read speed

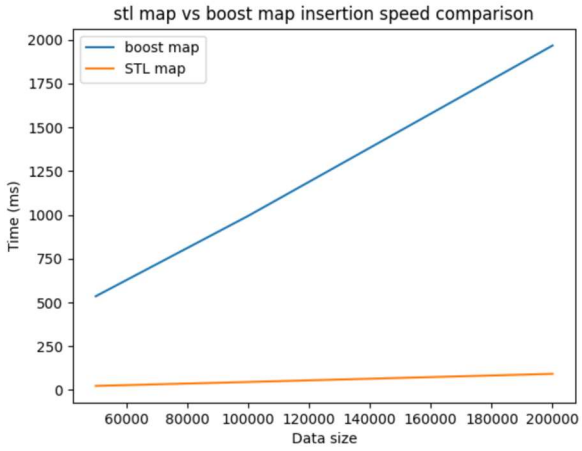


Figure 12, stl vs boost insertion speed

From the test data, we can know that the map in the stl library insertion performance is better than the map in the boost library. And start processes on different nodes, it can be seen from the data that the reading speed of the local node is faster than the cross node.

5.5 unordered set

	500,000	100,000	50,000
shared memory insertion	1917.18	3746.01	7265.25
STL map insertion	140.638	282.446	560.416
STL map read	14.2122	28.257	56.0418
shared memory local node read 1	82.5385	164.529	326.534
shared memory local node read 2	86.1956	170.801	328.254
shared memory cross node read 1	84.2618	167.271	330.468
shared memory cross node read 2	84.2511	167.273	330.481

Figure 13, test result table

We comparison of 3 sets of data volumes are 500,000, 100,000 and 50,000

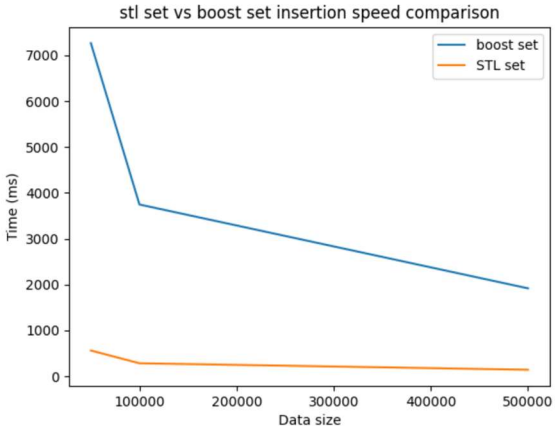


Figure 14, stl vs boost read speed

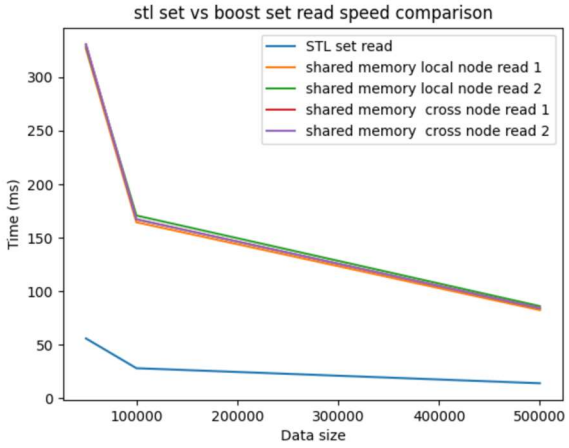


Figure 15, stl vs boost insertion speed

From the data, we can know that the set in the stl library insertion performance is better than the set in the boost library. And start processes on different nodes, it can be seen from the data that the reading speed of the local node is faster than the cross node.

5.6 Third part tools

We use the Intel® Memory Latency Checker process to test the server's environment verification.

```
Measuring idle latencies for random access (in ns)...
                Numa node
Numa node            0       1
       0          92.8   145.1
       1         146.0    90.3
```

Figure 3, MLC result 1

```
Measuring cache-to-cache transfer latency (in ns)...
Local Socket L2->L2 HIT  latency        51.5
Local Socket L2->L2 HITM latency        51.8
Remote Socket L2->L2 HITM latency (data address homed in writer socket)
                  Reader Numa Node
Writer Numa Node     0       1
           0         -   117.3
           1     117.2       -
Remote Socket L2->L2 HITM latency (data address homed in reader socket)
                  Reader Numa Node
Writer Numa Node     0       1
           0         -   117.0
           1     117.2       -
```

Figure 5, MLC result 2

```
Measuring Memory Bandwidths between nodes within system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
                Numa node
Numa node            0       1
       0        106167.4        34456.7
       1        34461.0 106160.7
```

Figure 6, MLC result 3

6. Conclusion

We plan to quantify the impact of NUMA on shared memory data structures. The existing C++ library has very limited support for NUMA; there is no NUMA support for some basic and commonly used data structures. We will optimize some commonly used data structures to support NUMA. Then, by using different test software and test

cases, we will observe the impact of supporting NUMA on the performance of using shared memory.

It is expected to achieve through this project as following,

1. Establish a NUMA performance testing environment and establish a performance baseline
2. Implement some commonly use data structure of C++ library that supports NUMA in anticipation of improved memory access performance
3. Make sure that the new implementation library can run in the real environment.

6.1 Our Thinking

1. Why is there no obvious performance difference between cross nodes?

The problem now is that with the Intel® Memory Latency Checker, it can be seen that there are indeed differences in memory access between different nodes. However, our programs on shared memory did not see such a difference.

Then the possible reason is that Boost's shared memory library boost::interprocess itself may not divide memory according to nodes, but access in strips, or average access. Therefore, all based on boost::interprocess It is impossible for the program to obtain NUMA-aware.

2. What are the possible future directions?

a. Instead of using boost::interprocess, implement data structures based on other NUMA-aware libraries.

b. Implement node memory application by yourself, for example, use mmap() to apply directly, or use functions in boost to directly specify nodes to apply for memory

7. References

[1] Christoph Lameter, “NUMA (Non-Uniform Memory Access): An Overview,” Acmqueue Aug 9, 2013.

[2] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu, Tsinghua University, “Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes,” USENIX Association 15th USENIX Symposium on Operating Systems Design and Implementation.