

CSC4005_A1_Report

119010108_HuangPengxiang

10/13/2021

Contents

Introduction	4
How to run my program	4
The sample output screen shot	5
Basic background	6
Parallel Odd-Even Transposition Sort	6
What is Parallel Odd-Even Transposition	6
The complexity of Parallel Odd-Even Transposition Sort && Some declarations in my program	7
MPI Programming	7
What is MPI Programming and Parallel Computing	7
What is the MPI's Topologies in my program	8
MPI function I have used in my program	9
What is Slurm Sever and How Does it Influence My Program	10
What Is Node and How it Inlunce My Program	11
What Does Internet Connection Influence My Program && Some Issues	11
The Design of This Program	12
Program Design	12
Data Structure Design	12
Design of Experiment	13
The Experiment Data And Analysis	14
Declarations	14
Parallel vs. Sequential	14
For the large size array	14
For the small size array	16

How Array size affect Performance	18
How number of cores affect Performance.	19
How Node allocations affect the performance	21
The Regression Model for Parallel Time Complexity	22
Conclusion	24
The performance Summary	24
Limitaiton of My Project	24
Conclutions	24
Appendix	24

Introduction

In this project, I designed a parallel odd-even transposition sort by using MPI. I run my project in the Slurm server, and collect the data of running program time duration. Also I separate those data based on different variables, for each variable, I analysis the program performance and check whether this variable have the significant influence for parallel odd-even sort. Moreover, This project also cover the comparison of the performance between the parallel program and sequential program.

How to run my program

```
#In order to run my program, you can run it on Slurm or you can also run it in local, and you need to make sure you already set the environment for mpi
```

```
And I use "salloc" to test my program in slurm
```

```
# It contains two parts
```

```
# first you need to open the /src/generatenum.c to generate numbers
```

```
$ gcc -o generate generate.c && ./generate # use this to generate the in.txt
```

```
# second to test the program
```

```
$ cd ..
```

```
$ mkdir build && cd build
```

```
$ cmake .. -DCMAKE_BUILD_TYPE=Debug # please modify this to Release if you want to benchmark your program
```

```
$ cmake --build . -j4 # make the program
```

```
$ salloc -N4 -n128 -t5
```

```
$/gtest_sort # first test
```

```
$ mpirun -np /* num of core */ ./main ../src/in.txt ../src/out.txt
```

```
# which allows you from the in.txt read the random number and write it orderly in out.txt
```

The sample output screen shot

The sample output is showing below:

```
huangpengxiang@huangpengxiangdeMacBook-Pro Project_1 % rm -rf build
huangpengxiang@huangpengxiangdeMacBook-Pro Project_1 % mkdir build && cd build
huangpengxiang@huangpengxiangdeMacBook-Pro build % cmake .. -DCMAKE_BUILD_TYPE=Debug
-- The C compiler identification is AppleClang 11.0.3.11030032
-- The CXX compiler identification is AppleClang 11.0.3.11030032
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /Library/Developer/CommandLineTools/usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found MPI_C: /usr/local/Cellar/open-mpi/4.1.1_2/lib/libmpi.dylib (found version "3.1")
-- Found MPI_CXX: /usr/local/Cellar/open-mpi/4.1.1_2/lib/libmpi.dylib (found version "3.1")
-- Found MPI: TRUE (found version "3.1")
-- Found Python: /Library/Frameworks/Python.framework/Versions/3.8/bin/python3.8 (found version "3.8.1") found
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/huangpengxiang/Desktop/Project_1/build
huangpengxiang@huangpengxiangdeMacBook-Pro build % cmake --build . -j4
[ 14%] Building CXX object CMakeFiles/odd-even-sort.dir/src/odd-even-sort.cpp.o
[ 14%] Building CXX object googletest/googletest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
[ 21%] Linking CXX shared library libodd-even-sort.dylib
[ 21%] Built target odd-even-sort
[ 28%] Building CXX object CMakeFiles/main.dir/src/main.cpp.o
[ 35%] Linking CXX executable main
[ 35%] Built target main
[ 42%] Linking CXX static library ../lib/libgtestd.a
[ 42%] Built target gtest
[ 64%] Building CXX object googletest/googletest/CMakeFiles/gmock.dir/src/gmock-all.cc.o
[ 64%] Building CXX object googletest/googletest/CMakeFiles/gtest_main.dir/src/gtest_main.cc.o
[ 64%] Building CXX object CMakeFiles/gtest_sort.dir/src/tests.cpp.o
[ 71%] Linking CXX static library ../lib/libgtest_maind.a
[ 71%] Built target gtest_main
[ 78%] Linking CXX executable gtest_sort
[ 78%] Built target gtest_sort
[ 85%] Linking CXX static library ../lib/libgmockd.a
[ 85%] Built target gmock
[ 92%] Building CXX object googletest/googletest/CMakeFiles/gmock_main.dir/src/gmock_main.cc.o
[100%] Linking CXX static library ../lib/libgmock_maind.a
[100%] Built target gmock_main
huangpengxiang@huangpengxiangdeMacBook-Pro build %
```

```
220891762032286702->9221184289651987980->9221778573293102101->92220261
222593291902989477->9222647378244943416->9222659829180917671->92229985
[ OK ] OddEvenSort.Random (7613 ms)
[-----] 2 tests from OddEvenSort (7615 ms total)
```

```
[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (7615 ms total)
[ PASSED ] 2 tests.
```

```
huangpengxiang@huangpengxiangdeMacBook-Pro build %
```

```
>2002774122~>2002937961~>2003474017~>2003574053~>2003611123~>2003759947~>2004663404~>2004835813~>2004932536~>2004972227~>2005210940~>2005268848~>2005308235~>2005972063~>2005981969~>2006249684~>2006499892~>2006935703~>2006960290~>2007000893~>2007089150~>2007302883~>2007527349~>2007618534~>2007622210~>2007829448~>2008197085~>2008431615~>2008533886~>2008644228~>2008683933~>20089373028~>2009318083~>2009438191~>2009541144~>2010099520~>2010199732~>2010216300~>2010495544~>2010574607~>2010856452~>2011276116~>2011524171~>2011607784~>2011774447~>2012057331~>2012154629~>2012216045~>2012455656~>2012589110~>2012633670~>2012864877~>2012938539~>2012938987~>2013107345~>2013314670~>2013316708~>2013967542~>2014029572~>2014087743~>2014267713~>2014709393~>2014089943~>2015146809~>2015155719~>2015280856~>2015606977~>2015627602~>2015913233~>2016172640~>2016268702~>2016351765~>2016364503~>2016473105~>2016644441~>2016857117~>2017300159~>2018092200~>2018401459~>2018409256~>2018459732~>2018512087~>2018624142~>2018699097~>2019270329~>2019030373~>2019993065~>2020410415~>2020510834~>2020606073~>2021339032~>2021673950~>2021749040~>2021869620~>2021967950~>2021969071~>2022000317~>2022207349~>2022277264~>20223285930~>20223707783~>2023017491~>2023034031~>2023306352~>2023152900~>2023793401~>2023383606~>2023954471~>2024043928~>2024131046~>2024214397~>2024379334~>2024381252~>2024385089~>2024555190~>2024606239~>2024712946~>2025041096~>2025415911~>2026167389~>2026191667~>2026252900~>2026580998~>2026976162~>2027316558~>2027642865~>2028097652~>2028271139~>2028355304~>2028387740~>2028406123~>2028506691~>2028642133~>2028840454~>2029136629~>2029275315~>2029384916~>2029500400~>2029895646~>2029966412~>2030200039~>2030455140~>2030463252~>2030713232~>2030851242~>2031094982~>2031717175~>2031735952~>2031756090~>2032271604~>2032346312~>2032369765~>2032434890~>2032538640~>2032862653~>2033179107~>2033218595~>2033789353~>2033822300~>2033979053~>2033990200~>2034050650~>2035270894~>2035855796~>2035913206~>2036042866~>2036355427~>2036386606~>2036897053~>2037819497~>2037224820~>2037608520~>2038504874~>2038712208~>2038918354~>2039706591~>2039843805~>2039968100~>2039974289~>2040341906~>2040528334~>2041303513~>2041453952~>2041878362~>2042167767~>2042374309~>2042422061~>2042455890~>2042618359~>2043002551~>2043114113~>2043255345~>2043314971~>2044045926~>2044282116~>2044355172~>2044416523~>2044465243~>2044468703~>2044799993~>2045143752~>2045166634~>2045528895~>2045616714~>2045618351~>2045824731~>20460771261~>2047133876~>2047350816~>2047380203~>2047858808~>2048398709~>2048411292~>2048426314~>2048553029~>2048698395~>2048736561~>2048756192~>2049240302~>2049461118~>20496689421~>2049852446~>2049931411~>2049967104~>2050032514~>2050322646~>2050328520~>2050555079~>2050595200~>2051567121~>2051847983~>2051882893~>2052011909~>2052061085~>2052066128~>2052153400~>2052384452~>2052605697~>2052532322~>2053403021~>2053617691~>2054105918~>2054372047~>2054821455~>2054040401~>2054049969~>2055347953~>2055406011~>2056271300~>2056361375~>2056450411~>2056548570~>2056589624~>2056914777~>2057085416~>2057178577~>205717736~>20577991224~>2058383373~>2058718128~>2058768669~>2059158182~>2059333289~>2059750621~>2059850483~>2059888004~>2060302371~>2060353300~>2060497196~>2060578360~>2060762124~>20608085711~>20608621529~>2061264262~>2061343572~>2061569224~>2062067064~>2062094307~>2062105889~>2062182017~>2062524349~>2062545753~>2062800690~>2062815640~>2063016806~>2063429973~>2064133026~>2064141109~>2064340601~>2064421192~>2064573350~>2064930395~>2064957506~>2065077352~>2065043818~>2065055417~>2065567040~>2065753308~>2065762179~>2067009578~>2067314012~>2067709442~>2067846044~>2068087850~>2068258043~>2068277549~>2068760419~>2068949433~>2068960591~>2068997202~>2069178451~>2069185189~>2069504653~>2069616492~>2069640431~>2069683440~>2069817542~>2070062318~>2070254257~>2070265596~>2070671144~>2070642179~>2070664131~>20707878910~>2071041249~>2071858112~>2071209574~>2071359896~>2071744338~>2072003695~>2072011601~>2072067987~>2072316799~>2072928322~>2073113083~>2073145900~>2073626013~>2074083176~>2074599955~>2074818599~>2074911691~>2075042228~>2075120305~>2075318685~>2075367044~>2075519902~>2075799070~>2075940937~>2076423165~>2076877811~>2076966517~>2077612811~>2077389356~>2077578766~>2077510082~>2077522631~>2077735120~>2078065845~>2079568715~>2079683734~>2079632066~>2079632218~>2079666498~>2079699435~>2079778337~>2079992257~>2080049538~>2080031152~>2080395900~>2080521333~>2080572925~>2081205023~>2081222575~>2081418831~>2081600330~>2081786044~>2081791188~>2081974525~>2082333081~>2082903472~>2082935284~>2083042839~>2083170627~>2083258431~>2083263763~>2083286936~>2083370278~>2083870294~>2083968597~>2084319012~>2084410578~>2084406066~>2085000094~>2085054924~>2085320115~>2085380395~>2085705804~>2086044503~>2086101107~>2086442370~>2086577926~>2087045420~>2087085660~>2087146326~>2087870015~>2088255899~>2088631926~>2088780843~>2088782149~>2089031820~>2089805365~>2090147176~>2090195691~>2090285282~>2090610677~>2090626669~>2091099168~>2091234169~>2091581391~>2091832111~>2091965349~>2092125076~>2092178276~>2092506351~>2092655965~>2092884749~>2092981200~>2093340564~>2093380672~>2093975362~>2094102979~>2094683830~>2095083727~>209521141~>2095792967~>2096445620~>2097085916~>2097041119~>2098041683~>2098044197~>2099000321~>2099003210~>2099321506~>2099506309~>2099613074~>2100116400~>2100210404~>2101149144~>2101350222~>2101489440~>2101712065~>2101966993~>2102088117~>210209163~>2102218044~>2102408567~>2102542833~>2102586906~>2102650633~>2102712041~>2102846050~>2102933959~>2103190260~>2103196467~>2103506383~>2103832899~>2104077754~>2104444089~>2104973896~>2105172541~>2105223570~>2105355871~>2105456305~>2105525811~>2105551192~>2105644100~>2106043093~>2106122361~>2106484530~>2106567198~>2106589370~>210677832~>2106850016~>2107042940~>2107370954~>210745378~>2107472744~>21080069910~>2108131658~>2108593013~>2109525087~>2109636170~>2109700644~>2110195082~>2110280440~>2110408704~>2110440429~>2111238033~>2111707200~>2112164004~>2112535091~>2112654252~>2112663367~>2112950714~>2113524597~>2113551405~>2114713201~>2114778652~>2115257668~>2115295509~>2115584685~>2116507494~>2116693191~>2117005527~>2117389420~>2117847176~>2117994441~>2118108594~>2118387028~>2119241103~>2119299744~>2119403924~>2119441601~>2119504356~>2119879604~>2120122086~>2120455742~>2120679647~>2120770309~>2120827352~>2121568329~>2121570000~>2121676154~>2121740653~>2121840271~>212192939~>2121967409~>2122114322~>2122325411~>2122236021~>2122675230~>2122794330~>2123007744~>2124000671~>2124054027~>2124569570~>2124941083~>2125067832~>2125262106~>2125337254~>2125416900~>2125529299~>2125537541~>2126626942~>2126775896~>2126804360~>2127376504~>2127649319~>2127782925~>2129126986~>2129582648~>2129657388~>2129834071~>2129933166~>2130172487~>2130678381~>2131225910~>2131441235~>2131526483~>2131687579~>2132085911~>213216763~>2132347268~>2132381630~>2132508327~>2132697153~>2133439958~>2133581065~>2133830902~>2133949377~>2134034045~>2134076242~>2134165809~>2134640904~>2134680473~>2134808308~>2135020079~>2135006654~>2135425287~>2135774771~>2135803763~>2136270835~>2136572396~>2136802166~>2137427108~>2137463317~>2137697537~>2137922355~>2138415406~>2138779626~>2139540807~>2139575860~>2139752428~>2140208969~>2140221777~>2140358078~>2140911697~>2140925466~>2140942326~>2140954550~>2141122697~>2141274353~>2141402227~>2141737895~>2141800150~>2142056027~>2142170568~>2142718783~>2143117851~>2143258564~>2143932550~>2144430709~>2144553965~>2144664328~>2144695264~>2145104342~>2145155274~>2145315239~>2145300726~>2145530556~>2146130401~>2146538447~>2146672612~>2147050602
```

```
input size: 10000
proc number: 4
duration (ns): 128032400
throughput (gb/s): 0.000581929
huanmengenv1andhuanmeng1anddeMarBook-Pro build 0
```

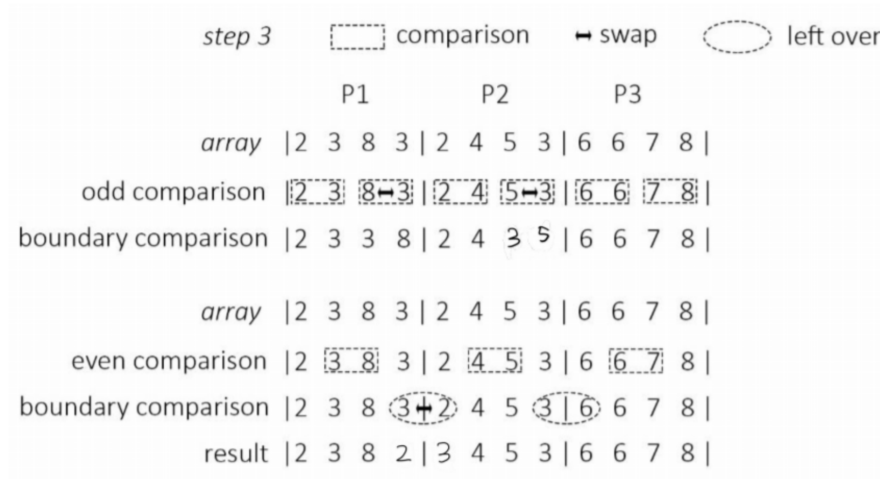
Basic background

Parallel Odd-Even Transposition Sort

What is Parallel Odd-Even Transposition

Odd-Even Transposition Sort is a parallel sorting algorithm. It is based on the Bubble Sort technique, which compares every 2 consecutive numbers in the array and swap them if first is greater than the second to get an ascending order array. First step.Insides each process, compare the odd element with the posterior even element in odd iteration, or the even element with the posterior odd element in even iteration respectively. Swap the elements if the posterior element is smaller. Second step, If the current process rank is P, and there some elements that are left over for comparison in step 1, Compare the boundary elements with process with rank P-1 and P+1. If the posterior element is smaller, swaps them. Repeat 1-2 until the numbers are sorted. You can clear understand it in the below figure

comparisons; swap; left over (boundary elements that should be compared).



The complexity of Parallel Odd-Even Transposition Sort && Some declarations in my program

The approach of Parallel Odd-Even Transposition Sort is basically the same as Bubble sort, which is always keep $O(N^2)$. And in my program, I didn't do the check when the array is already sorted, so the complexity of my sort is always keep $O(N^2)$ in every case. **And MOST IMPORTANTLY, I think we can not do the check step inside the program in order to uniquely control variable.** For instance, if we generate the array randomly, it may appear that The already sorted array (best case) for 1 core, 100 array size to run, it only need to run 1 time to check every element in it then return, which is $O(1)$. However, if we generate the array again, and this time is the worst case, for 4 core to run, the time duration is obviously longer than the previous one, which is much less than $O(N^2)$ but still bigger than $O(1)$. So, in order to control the unique variable and avoid accident case, I refuse this case happens. It means that no matter what array is, the complexity in my program is $O(N^2)$.

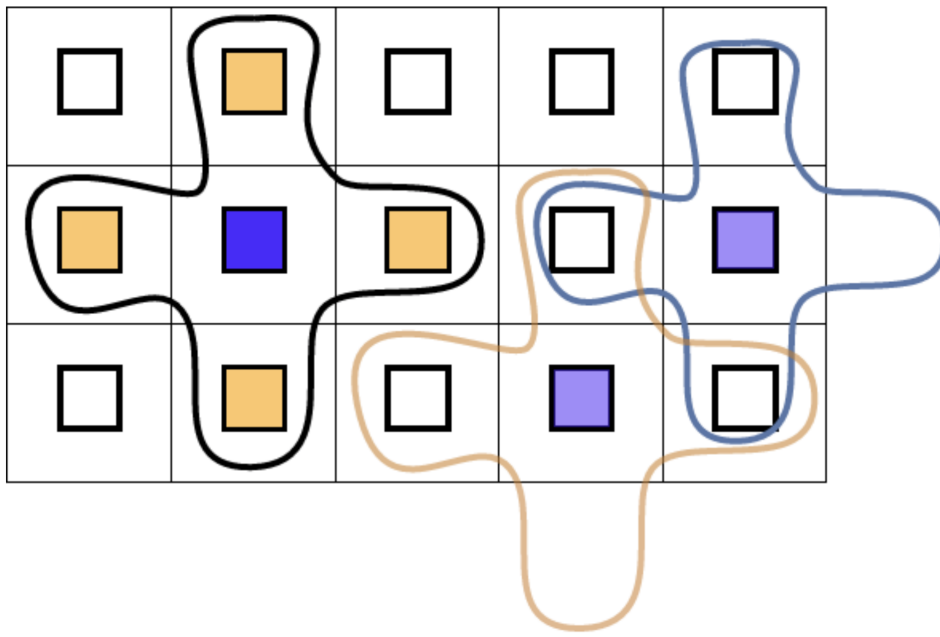
MPI Programming

What is MPI Programming and Parallel Computing

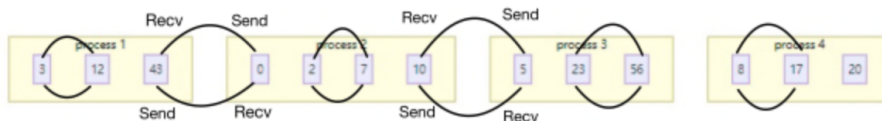
Message Passing Interface (MPI) is a communication protocol for parallel programming. MPI is specifically used to allow applications to run in parallel across a number of separate computers connected by **network** (may have some influence on the sort performance, will introduce later). MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

What is the MPI's Topologies in my program

In standard MPI topology, A communicator describes a group of processes, but the structure of your computation may not be such that every process will communicate with every other process. For instance, in a computation that is mathematically defined on a Cartesian 2D grid, the processes themselves act as if they are two-dimensional ordered and communicate with N/S/E/W neighbors. If MPI had this knowledge about your application, it could conceivably optimize for it, for instance by renumbering the ranks so that communicating processes are closer together physically in your cluster. you may understand in the following figure.



But in my program, I just view all of core in different rank as a mathematically a line, one-dim. each rank represent a graphic element which can process the data, can also contains one ore more cpu in it. They can communicate with the adjacent rank element by specific functions. The way they communicate each other in order to implement the odd even sort is like the follwing figure.



MPI function I have used in my program

I have used MPI_Bcast function to board cast the root inforamtion to the other rank.

```
#include "mpi.h"

int MPI_Bcast(
    void          *buffer,
    int           count,
    MPI_Datatype  datatype,
    int           root,
    MPI_Comm      comm
);
```

MPI_Scatterv function to Scatters data from one member across all members of a group.

```
int MPI_Scatterv(
    _In_ void      *sendbuf,
    _In_ int       *sendcounts,
    _In_ int       *displs,
    MPI_Datatype  sendtype,
    _Out_ void     *recvbuf,
    int          recvcount,
    MPI_Datatype  recvtype,
    int          root,
    MPI_Comm      comm
);
```

MPI_Gatherv to Gathers variable data from all members of a group to one member.

```
int MPI_Gatherv(
    _In_ void      *sendbuf,
    int          sendcount,
    MPI_Datatype  sendtype,
    _Out_opt_ void *recvbuf,
    _In_opt_ int  *recvcounts[],
```

```

    _In_opt_ int          *displs[],
        MPI_Datatype recvtype,
        int          root,
        MPI_Comm      comm
);

```

also use `Send` and `Recv` to send and receive the information from the adjacent rank

```

int MPI_Send(
    void* msg_buf_p,
    int msg_size,
    MPI_Datatype msg_type,
    int dest,
    int tag,
    MPI_Comm communicator
);

int MPI_Recv(
    void* msg_buf_p,
    int buf_size,
    MPI_Datatype buf_type,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status_p
);

```

What is Slurm Server and How Does it Influence My Program

The Slurm Workload Manager, formerly known as Simple Linux Utility for Resource Management (SLURM), or simply Slurm, is a free and open-source job scheduler for Linux and Unix-like kernels, used by many of the world's supercomputers and computer clusters. Slurm has many ways to submit the homework. But in my program, I just used one way named `salloc`, which is an interactive way. The command is like that

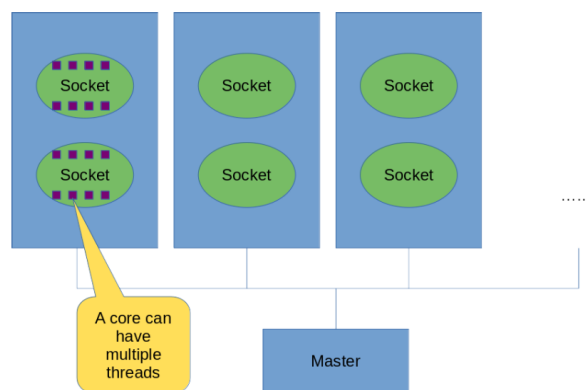
```

$ salloc -N2 -n48 -t10
# for example, it will prepare a session with cores distributed on two
nodes and the session can last for mins:
$ scancel --user=$(whoami)
# cancel my application

```

What Is Node and How it Influence My Program

A node is a connection point inside a network that can receive, send, create, or store data. Each node requires you to provide some form of identification to receive access, like an IP address. A few examples of nodes include computers, printers, modems, bridges, and switches. A node is any physical device within a network of other tools that's able to send, receive, or forward information. A personal computer is the most common node. It's called the computer node or internet node. In our server, it may look like:



And in the following report, I found that **the time duration will be influenced by the number of nodes and the allocations of node**. Since the communication between each node will consume time, and it will have latency for the algorithm, so it will cause some difference even if you use the same number of cores but will not the same number of node. It will be shown and proofed in the next with detailed data.

What Does Internet Connection Influence My Program && Some Issues

During the experiment, I found that the internet connection is not stable when the people are too crowded in server. It may have more possibility to have higher latency. So I am afraid it may have an influence my data and further analysis. **It is also a limitation of this project**. And also, I have encountered this issue

when I do the experiment. My WLAN network is still good but I found that the connection with server is not stable and shown the issue below and I can not run program. And the solution for me is to wait, after I wait couple seconds or even more, then I can run my program.

```
bash-4.2$ mpirun -np 2 ./main ../src/in.txt ../src/out.txt
```

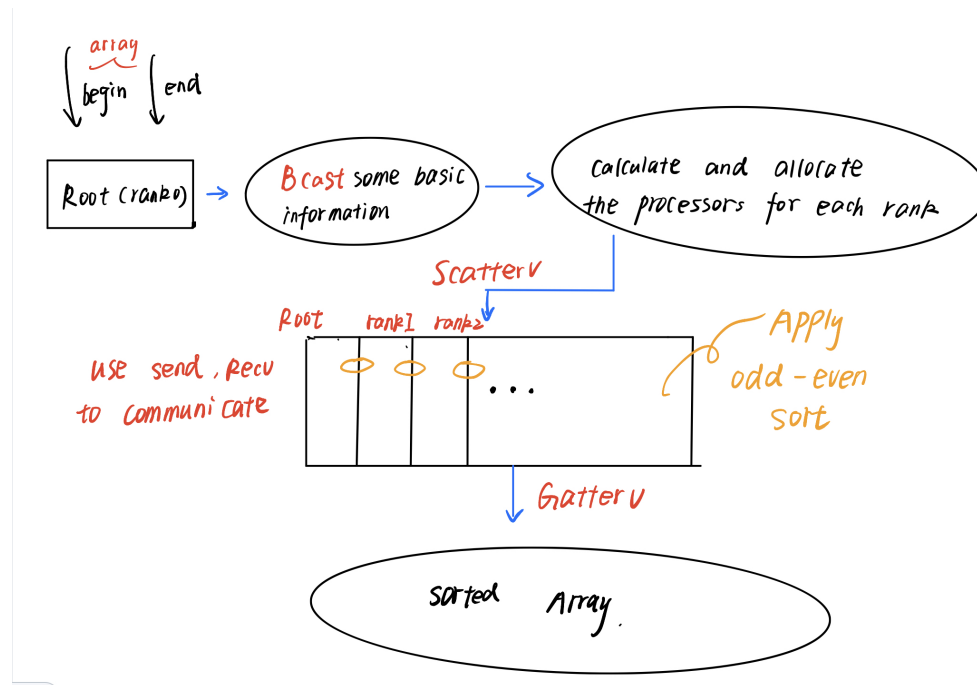
```
An ORTE daemon has unexpectedly failed after launch and before communicating back to mpirun. This could be caused by a number of factors, including an inability to create a connection back to mpirun due to a lack of common network interfaces and/or no route found between them. Please check network connectivity (including firewalls and network routing requirements).
```

```
bash-4.2$ █
```

The Design of This Program

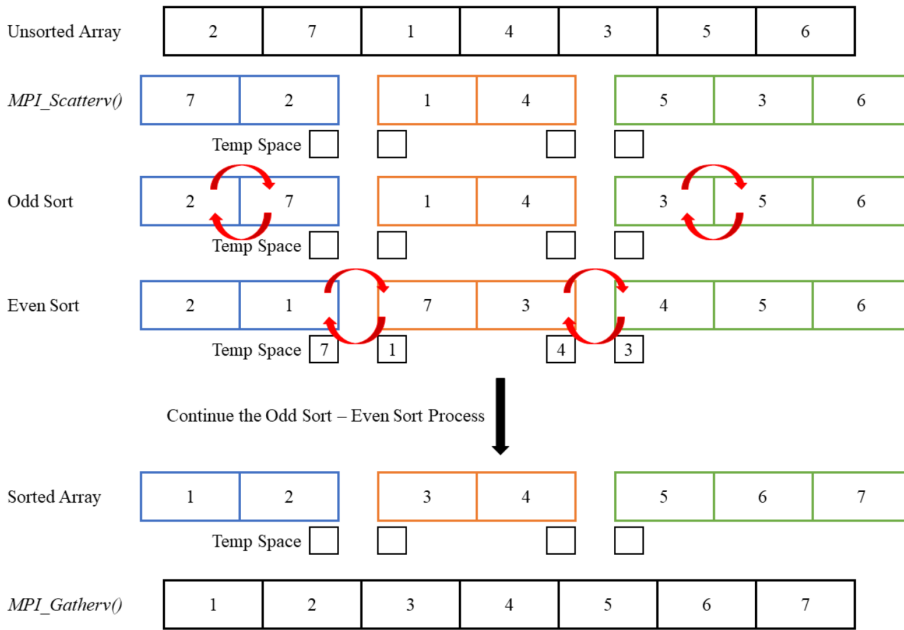
Program Design

The program flow is showing below:



Data Structure Design

To avoid change the original data, I use the array to store the parameter passed in the program and sort the array, finally send the data back to the place where it should be. It may looks like that:



Design of Experiment

As we reported before, The complexity of sequential $O(n^2)$ obviously, and I am curious about the Time complexity of Parallel computing, and how does other variable affect the time duration. During the background information, I predict that the follwing element may have a influence for the sort performance.

- Array size
- Number of Core
- Number of Node and Its Allocation

The outline of my experiment is that:

- First we compare the performance between Parallel and Sequential roughly, and find their apparantly difference. (**Notice:** need to control the same node,which is one, and the same array size.)
- Second we compare the array size does what influence for parallel sort. (Node 1, and we have 32 core to run this program)
- Third we compare the number of cores will do what affect (set array size is 10000, and increase the core from 2 to 128)
- then we find how number of node affect our performance

- finally, we combine those together to find out the time complexity for parallel computing in this sort and DO the regression to find out the model of Parallel sort.

The Experiment Data And Analysis

As reported in Design of Experiment, The experiment analysis is separated into 5 parts.

Declarations

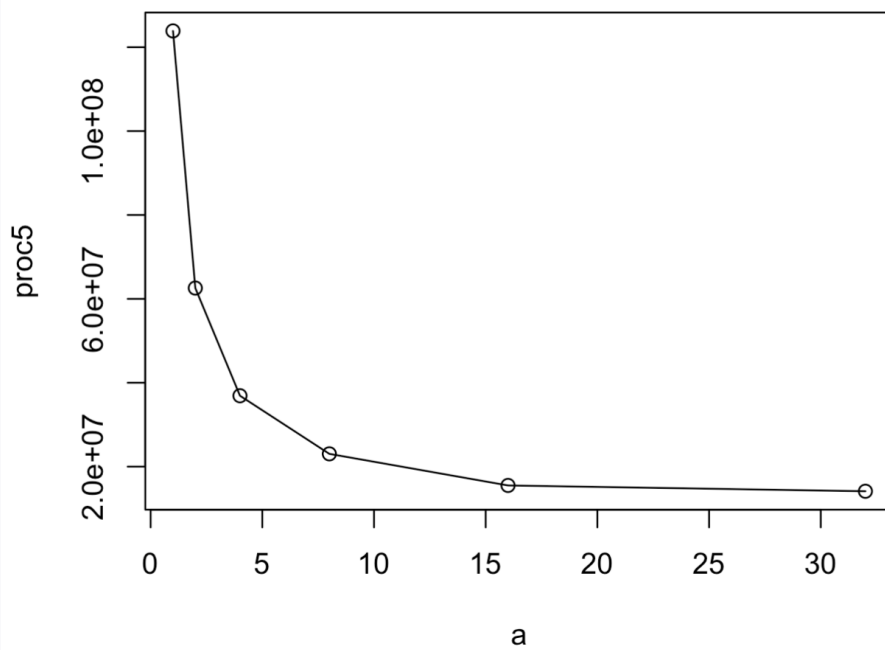
We only use Time Duration to measure the performance. If this condition could sort the array in a less time compared to another, we say this one has better performance

Parallel vs. Sequential

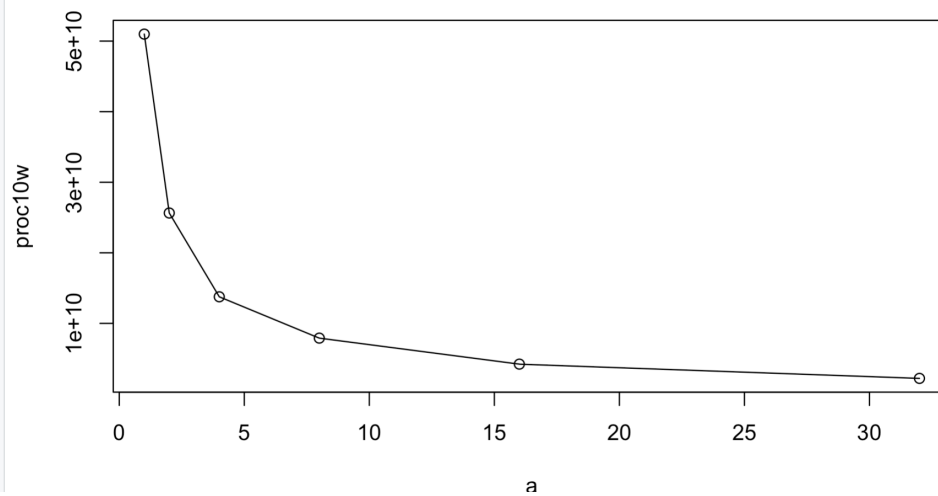
To compare the performance between the Parallel sort and Sequential sort, **First we need to concer the array size.** It means that we have to compare the performance in different length of array.

For the large size array

For the lagrge array, i first pick 5 thousand, we found that the Paralle sort is much better than Sequential one. **The figure showing below is the Node 1, and the x-axis is the number of core, which range in (1,2,4,8,16,32). the y-axis is the time duration**



Then I pick 10 thousand data size, and I found almost the same shape figure as 5 thousand.



Analysis

For input size is 5000, the time is (ns)

time_dur_core1 = 123879672

time_dur_core2 = 62573025

```
time_dur_core4 = 36900207
time_dur_core8 = 22312568
time_dur_core16 = 15509580
time_dur_core32 = 14126808
```

we first compare the Sequential one (time_dur_core1) with the Parallel one (time_dur_core32) and we found that: The parallel one almost save 8.7 times of Sequential one. which means, only In the time point of view, **Parallel sort has much less time consumption with Sequential one.**

```
print(time_dur_core1/time_dur_core32)
```

```
## [1] 8.76912
```

However, I found an another fact, that. time_dur_core1/time_dur_core2 is almost 2, and time_dur_core16/time_dur_core32 is almost 1. Logically, in ideal odd even sort, the core 2 should have 2 speed than core 1, and similarly, core 32 should have 2 times speed than core 16. but core 32 is almost the same as core 16. what affect that? **It means that when data size is large, the commnucation between each core in the same node also time consuming**

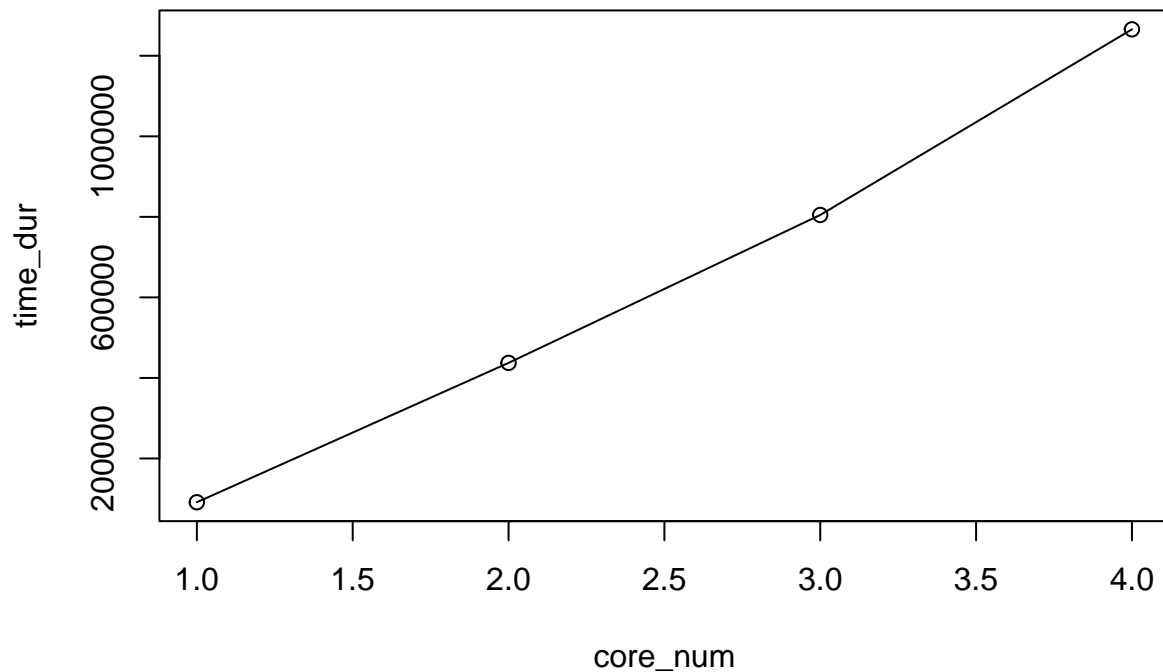
```
print(time_dur_core16/time_dur_core32)
```

```
## [1] 1.097883
```

For the small size array

The data is showing below

```
# the size of array is 12, time is (ns)
time_dur <- c(91357,437356,804708,1265617)
core_num <- c(1,2,3,4)
plot(core_num,time_dur,type ="o")
```

we can easily found that, for the small size data, The sequential one is much better than Parallel one, which also shows that the commnucation between each core in the same node also time consuming. In this case, the communication time even larger than sort time.

one extreme case:

If the core number even exceed the size of array, of course the program may have some fault. But it is ok when I run it in my mac. My program wiil allow you to allocate the 0 cores in a rank. Anyway, **Don't test my parallel program when array size even smaller than cores.**

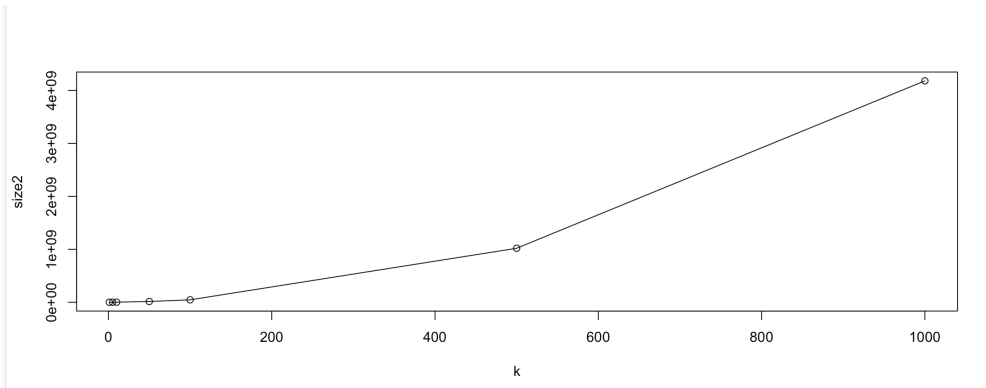
```
throughput (gb/s): 0.000400004
huangpengxiang@huangpengxiangdeMacBook-Pro build % mpirun -np 4 ./main ../src/in.txt ../src/out.txt
The rank 0 have: 1 processors.
The rank 2 have: 1 processors.
The rank 1 have: 1 processors.
The rank 3 have: 0 processors.
the Sorted array is: 1->2->321
input size: 3
proc number: 4
duration (ns): 608743
throughput (gb/s): 3.67179e-05
huangpengxiang@huangpengxiangdeMacBook-Pro build %
```

Connect to Host in New Window

How Array size affect Performance

To test the how array size affect the performance, I first set the core number is 16. And to avoid the small size bring more significant latency between communication, I set the data size range in (100, 1k, 5k, 1w, 5w,10w). And Then I plot the graph for that.

the x-axis represent the data size from 100~100000, the y-axis represent the time duration



Analysis

we can easily found that, with the increase data size with 10 times, the during time will also increse almost 10 times.

```
time_size1w = 45930978
time_size10w = 4180986908
time_size5w = 1020206504
time_size500 = 1354113
time_size5k = 14868410

print(time_size10w/time_size1w)
```

```
## [1] 91.0276
```

```
print(time_size5w/time_size5k)
```

```
## [1] 68.61571
```

```
print(time_size5k/time_size500)
```

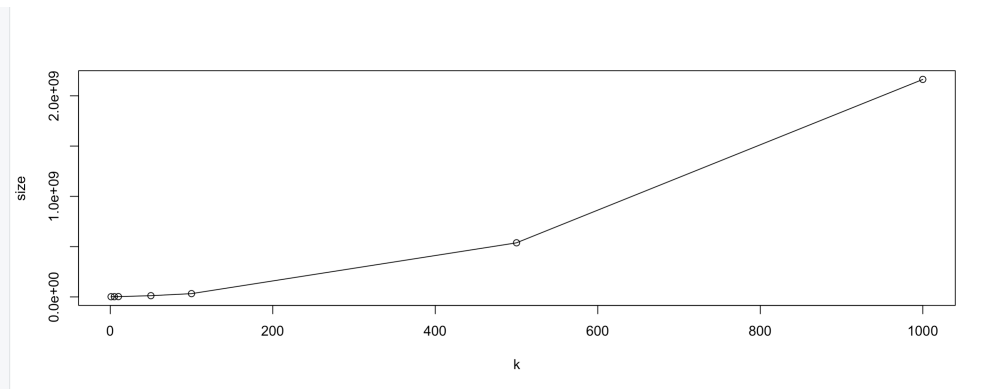
```
## [1] 10.98018
```

This data is mostly used for compute the time complexity. However, when I compute the size 5k/500, I get the number close to 10. which means that: The complexity for Parallel is $O(N)$. However, when I do the another comparison, which is 5w/5k, it get almost 7. Even, 10w/1w is almost get 100! It is totally don't obey the idea rule for parallel computing. The reason for that may be the fowlling:

- The communication for large data is very time consuming for each rank in mpi.
- The communication for large data is also time consuming for each node
- The case will be even worse if the data need to communicate with each rank, and then still need to transport to another Node.

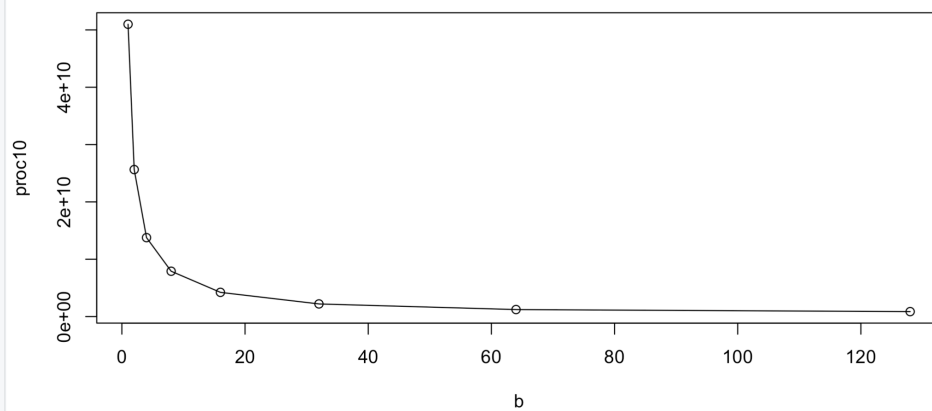
It means that the degree of communication latency will be Square, intuitively.

And I test another data to reproof my analysis. (32 core, data size range in (100, 1k, 5k, 1w, 5w,10w)). It have the same shape with the previous one.



How number of cores affect Performance.

For this case, I design it very in a simple way. First I make the size of array to 10w. the number of core is range in (1,2,4,8,16,32,64,128) And I observed the follwing result.



We can observe that when the number of core decrease, the time will also increase.

```
core16_sie10w = 4221353766
core32_size10w = 2207320400
core64_size10w = 1226279018
core128_size10w = 866243730

print(core16_sie10w/core32_size10w)
```

```
## [1] 1.912434
```

```
print(core32_size10w/core64_size10w)
```

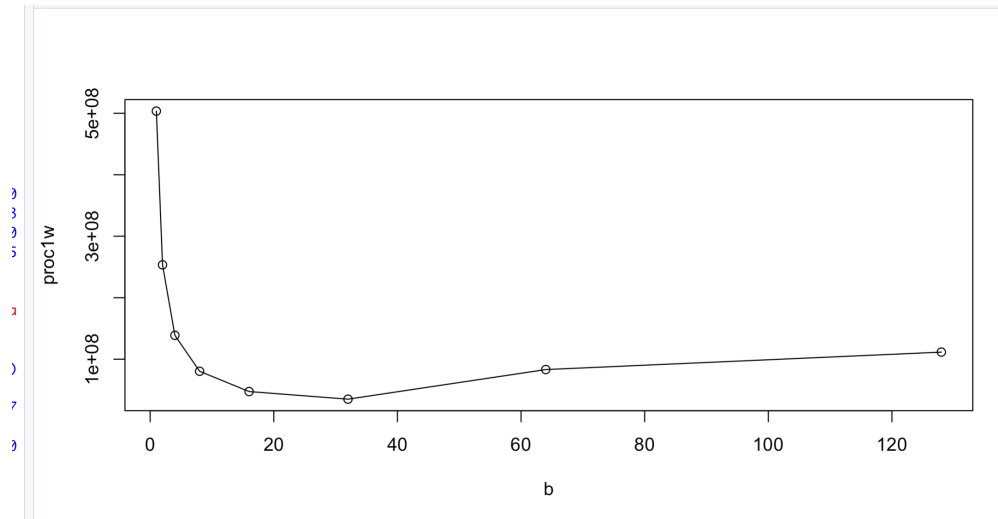
```
## [1] 1.800015
```

```
print(core64_size10w/core128_size10w)
```

```
## [1] 1.415628
```

But stil, with the node and core increase, the value will decrease, which idealy will be 2.

And moreover, with the core increase, the time even increase. Another data: (1w array size)



The reason for that is: When core increase from 32 to 64, it means that we use 2 node fully(in this case). and from 64 core to 128 core, we use 4 node fully. The communication time cost may have more influence than its performance improvement by the increase of the core. And we also design a experiment to check whether this hypothesis is right.

How Node allocations affect the performance

We design a experiment with core number 32 totally, data size is 10000. And we have 3 allocations, which is (32,0,0,0),(16,16,0,0),(8,8,8,8). If 3 allocation have the same performance, we could say, Node allocation have no influence on performance, however, we have to admit the Node allocation will affect the performance.

```
# can be designed by following steps
$ salloc -N1 -n32 --ntasks-per-node=32
$ salloc -N2 -n32 --ntasks-per-node=16
$ salloc -N4 -n32 --ntasks-per-node=8
```

The time for (16,16,0,0)

```
47->2146672612->2147056062
input size: 10000
proc number: 32
duration (ns): 76316070
throughput (gb/s): 0.000976279
bash-4.2$
```

The time for (32,0,0,0)

```

47->2146672612->2147056062
input size: 10000
proc number: 32
duration (ns): 31311693
throughput (gb/s): 0.00237949
bash-4.2$ █

```

The time for (8,8,8,8)

```

965->2144664328->2144695264->2145104342->2145155274->
47->2146672612->2147056062
input size: 10000
proc number: 32
duration (ns): 114873625
throughput (gb/s): 0.000648589
bash-4.2$ █

```

Then I found that the $(32,0,0,0) > (16,16,0,0) > (8,8,8,8)$, with almost 2 times of the other. It means that the allocation of node have influence on the performance. Also, we also can conclude that: **the time communication latency between each node is longer than the latency between each core in the same node.** Hence, If we would better let the core in one node as much as possible to improve the performance.

The Regression Model for Parallel Time Complexity

To compute the Time complexity for the Parallel, We need to build a model for it. From the previous analysis, we can divide the Time complexity into 2 parts, first is computation part, second is communication part. $T_p = T_{compute} + T_{communicate}$. For the communication part.

$$T_{communication} = T_{startup} + n_1 T_{coretrans} + n_2 T_{nodetrans}$$

- $T_{startup}$ represent the Message latency (assume constant)
- $T_{coretrans}$ represent the latency from one core to another, within a node.
- $T_{nodetrans}$ represent the latency from one node to another
- n_1 represent the message number need to transfer to another core
- n_2 represent the message number need to transfer to another node

For computation part, which is much easier. If we assume the latency of computation is constant, then

The computation part is $O(N)$, which is linear with the array size. **Then, we can make a assumption that the Big(O) for Parallel is $O(N)$.** because the communication is $O(n)$ and computation is also $O(n)$, theoretically.

The model for time complexity is linear, theoretically. Then I use the my data to try to prove it.

```
# I use the data in the "Array size analysis" (100,250, 500,1000, 5k, 1w, 5w,10w)
mysize <- c(100, 250,500,1000 ,5000,10000,50000,100000)
mytime <- c(992233,1293,1354113,14023141,1486410,45930978,1020206504,4180986908)
data <- data.frame(mysize,mytime)
myfit <- lm(mytime~mysize,data)
summary(myfit)
```

```
##
## Call:
## lm(formula = mytime ~ mysize, data = data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -784307560  -70977296  140460424  154163792  409685168
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -162273612  161459680  -1.005    0.354
## mysize      39336      4064    9.679 6.98e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 388700000 on 6 degrees of freedom
## Multiple R-squared:  0.9398, Adjusted R-squared:  0.9298
## F-statistic: 93.68 on 1 and 6 DF,  p-value: 6.975e-05
```

We found that the Multiple R-squared: 0.9398 nad Adjusted R-squared: 0.9298, which is high. Then we can say the time complexity for this model Linear.

Conclusion

The performance Summary

From the previous analysis, The performance mainly is depend on array size. When the size is very small (such as 20), The Sequential performs better than parallel. When the size is at medium, not very large and not very small, paralle sort with less nodes perform better, since we conclude that the communication between nodes is kind of time consuming. For the very large array, then since we only have 32 cores per node, then we have to choose more node to satisfy computing requirements, but we also need to put as much cores in less nodes as possible.

Limitaiton of My Project

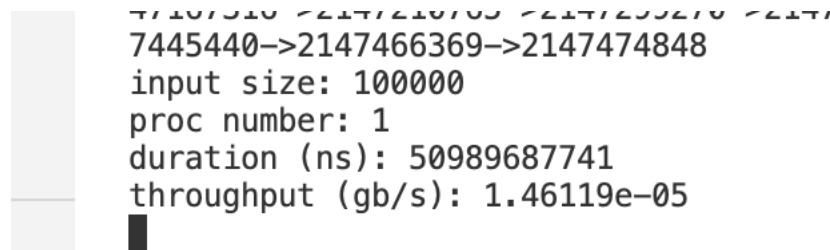
The biggest limitation is not enough data for further analysis. the data size is not enough for the regression proof. without much data, the analysis will also be influenced by accident data. But due to the time strict on server, I only get those data. Another Limitation is that there are few approach to know how the communication between nodes and cores affect performance. cause I think the number of data which needed to be trasnported to another is random. so maybe the model is not fair enough to prove linear time complexity.

Conclutions

Based on MPI, This project recover the parallel odd-even transposition sort and analyze its performance for each case.

Appendix

there are few sreenshots of my running results to show my data authenticity.



```
7445440-2147466369-2147474848
input size: 100000
proc number: 1
duration (ns): 50989687741
throughput (gb/s): 1.46119e-05
```

7445440->2147466369->2147474848
input size: 100000
proc number: 2
duration (ns): 25641204119
throughput (gb/s): 2.90571e-05
■

0

/445440->2147466369->2147474848
input size: 100000
proc number: 4
duration (ns): 13768673071
throughput (gb/s): 5.41126e-05
■

/445440->2147466369->2147474848
input size: 100000
proc number: 8
duration (ns): 7910378049
throughput (gb/s): 9.41874e-05
■

/445440->2147466369->2147474848
input size: 100000
proc number: 16
duration (ns): 4221353766
throughput (gb/s): 0.000176497
■

input size: 100000
proc number: 32
duration (ns): 2207320400
throughput (gb/s): 0.00033754
■

7445440->2147466369->2147474848
input size: 100000
proc number: 64
duration (ns): 1226279018
throughput (gb/s): 0.000607576
■

0

```
7445440->2147466369->2147474848
input size: 100000
proc number: 128
duration (ns): 866243730
throughput (gb/s): 0.000860102
█
```

```
46259938->2146460955->2146554066->2146710114->2
input size: 10000
proc number: 1
duration (ns): 503505796
throughput (gb/s): 0.000147974
bash-4.2$ █
```

```
46259938->2146460955->2146554066->214671
input size: 10000
proc number: 2
duration (ns): 253466029
throughput (gb/s): 0.000293948
bash-4.2$ █
```

```
47->2146672612->2147056062
input size: 10000
proc number: 32
duration (ns): 76316070
throughput (gb/s): 0.000976279
bash-4.2$ █
```

```
47->2146672612->2147056062
input size: 10000
proc number: 32
duration (ns): 76316070
throughput (gb/s): 0.000976279
bash-4.2$ █
```