Big Data sets on High Performance Computing

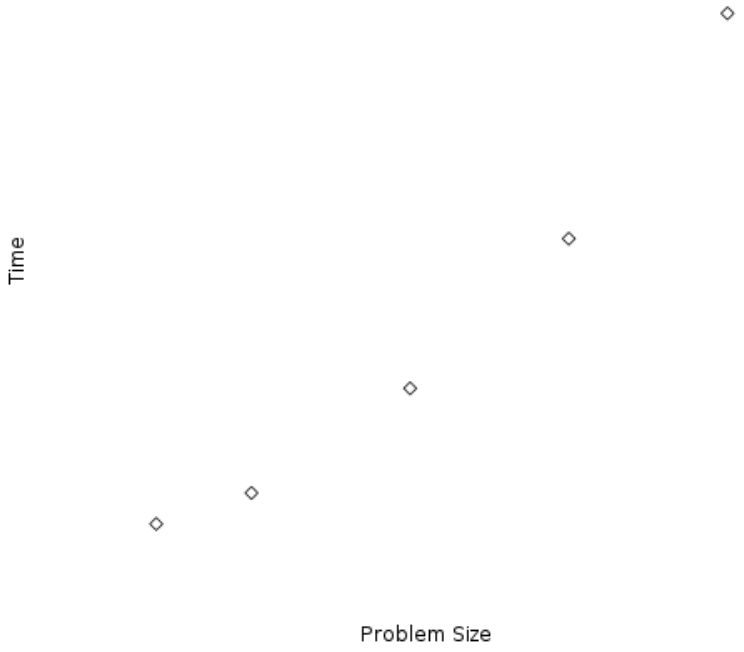
Michael Carlise

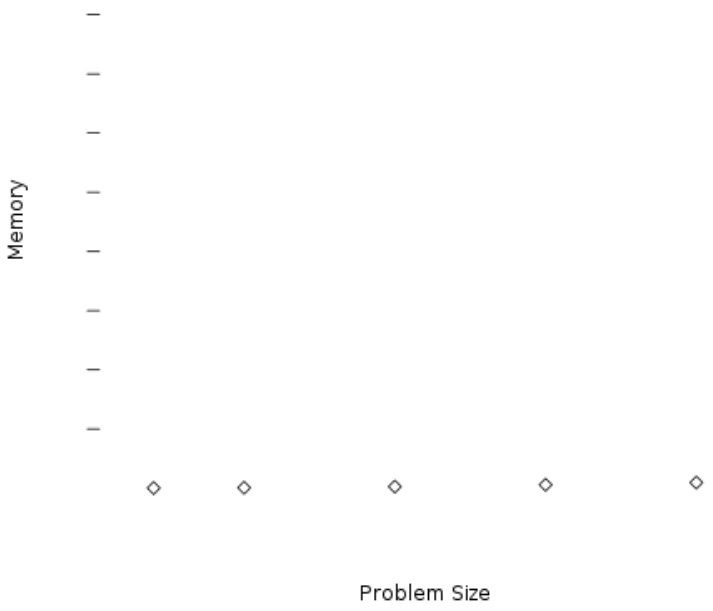
West Virginia University

mcarlise@mail.wvu.edu

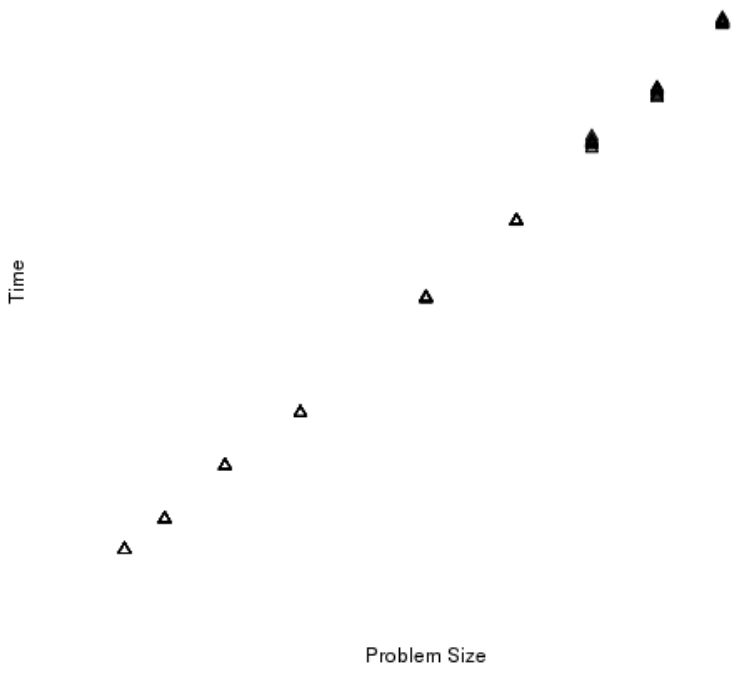
March 31, 2016

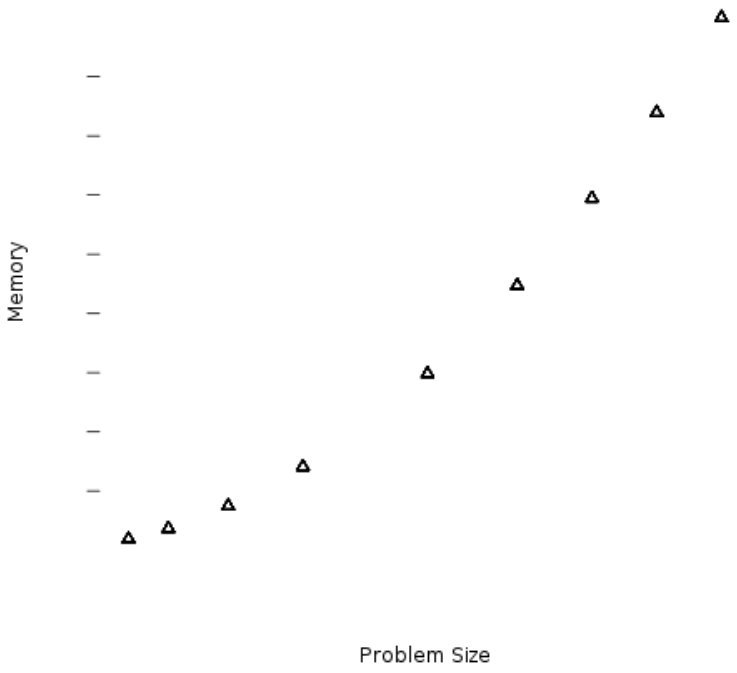
Problem Size/Compute Time Correlation





Problem Size/Compute Time Correlation





Combinations

Limitation	**Solution**
Compute time only	Parallel Computing
Memory only	Distributed Computing
Compute time and Memory	Distributed Computing

Table: Method needed given Limitation

Algorithm Efficiency

```
#!/usr/bin/Rscript

# Schema for Data field representation
mort.schema <- c ( .X0=19, ResidentStatus=1, .X1=40, Education1989=2,
Education2003=1, EducationFlag=1, MonthofDeath=2, .X2=2, Sex=1, AgeDetail=4,
AgeSubstitution=1, AgeRecode52=2, AgeRecode27=2, AgeRecode12=2,
AgeRecodeInfant22=2, PlaceOfDeath=1, MaritalStatus=1, DayOfWeekofDeath=1,
.X3=16, CurrentDataYear=4, InjuryAtWork=1, MannerOfDeath=1,
MethodOfDisposition=1, Autopsy=1, .X4=34, ActivityCode=1, PlaceOfInjury=1,
ICDCode=4, CauseRecode358=3, .X5=1, CauseRecode113=3, CauseRecode130=3,
CauseRecode39=2, .X6=1, Conditions=281, .X8=1, Race=2, BridgeRaceFlag=1,
RaceImputationFlag=1, RaceRecode3=1, RaceRecode5=1, .X9=33,
HispanicOrigin=3,.X10=1, HispanicOriginRecode=1 )

# Function returns a list and skips over any field name that begins with .X
unpack.line <- function ( data, schema ) {
        filter.func <- function ( x ) { substr (x,1,2) != ".X" }
        data.pointer <- 1
        output.data <- list ()
        for ( i in 1:length(schema) ) {
                if ( filter.func ( names(schema)[i] ) ) {
                        output.data [[ names(schema)[i] ]] <- type.convert (
                                substr ( data, data.pointer, data.pointer+schema[i] - 1 ),
                                as.is=TRUE
                        )
                }

                data.pointer <- data.pointer + schema[i]
        }

        output.data
}

# Output wanted


# Creates hash/key value of sex of record
sex.map.fn <- function (v) {
        record <- unpack.line (v, mort.schema)

        key <- ifelse ( record[["Sex"]]==FALSE, "female", "male" )
        key
}

# Iterate through data

male <- 0
female <- 0
while ( length (current <-
                                readLines('data', n = 1, warn = FALSE )) > 0 ) {

        sex <- sex.map.fn ( current )

        if ( sex == "male" ) {
                male <- male + 1
        } else {
                female <- female + 1
        }
}

male
female
```

Algorithm Efficiency

Forever!

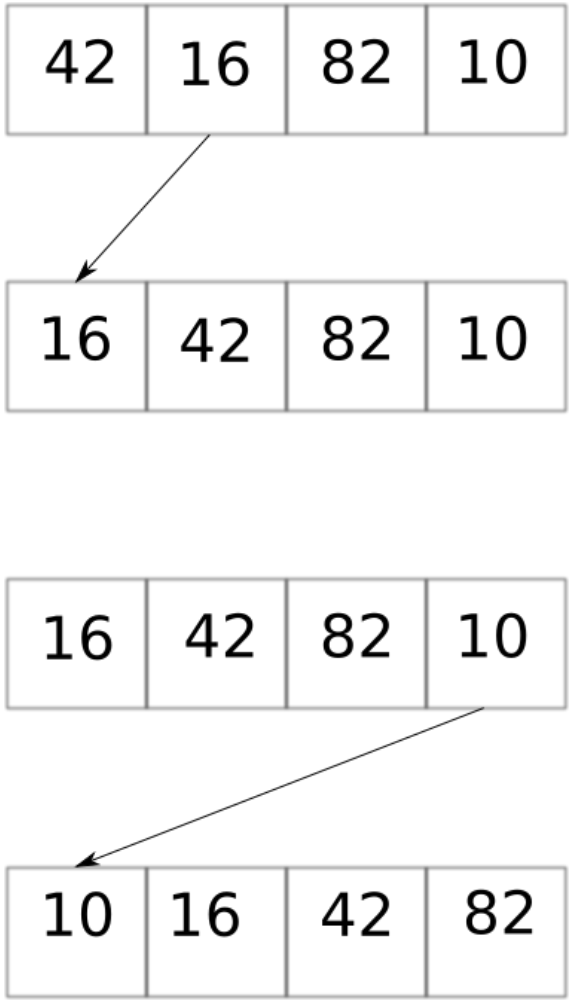
Algorithm Efficiency

```
cat VS09MORT.DUSMCPUB | cut -b 69 |
awk '{ if ( $1=="F" ) s = s + 1 }
        END { print s, NR-s }'
```

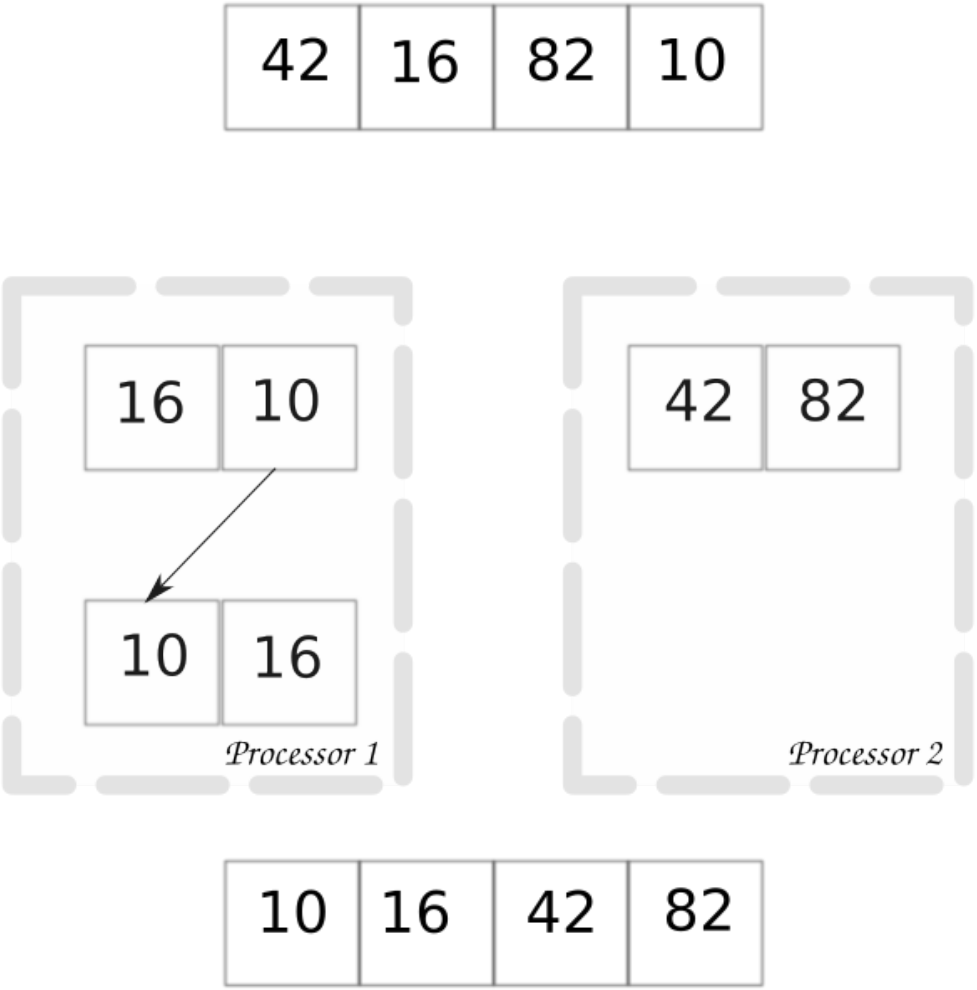
Algorithm Efficiency

5.679 seconds!

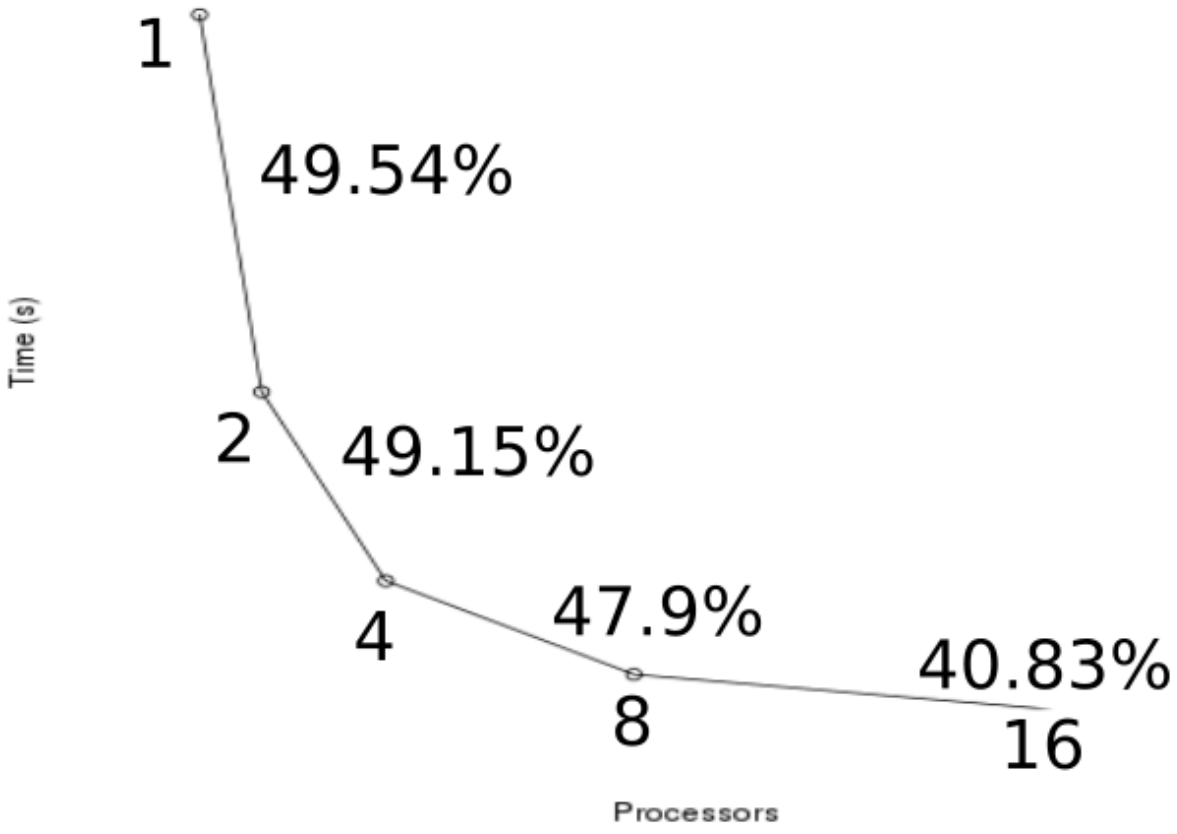
Serial Sort



Parallel Sort



Observed Speedup



Observed Speedup

12.64x

Amdahl's law

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

#define COLUMNS    10000
#define ROWS       10000
#define MAX_TEMP_ERROR 0.01

double Temperature[ROWS+2][COLUMNS+2];
double Temperature_last[ROWS+2][COLUMNS+2];

void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {

    int i, j;
    int max_iterations;
    int iteration=1;
    double dt=100;
    struct timeval start_time, stop_time, elapsed_time;

    initialize();

    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                            Temperature_last[i][j+1] + Temperature_last[i][j-1]);
            }
        }

        dt = 0.0; // reset largest temperature change

        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
                  dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
                  Temperature_last[i][j] = Temperature[i][j];
            }
        }

        if((iteration % 100) == 0) {
              track_progress(iteration);
        }

            iteration++;
    }
}


void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }

    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
```

Amdahl's law

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

#define COLUMNS    10000
#define ROWS       10000
#define MAX_TEMP_ERROR 0.01

double Temperature[ROWS+2][COLUMNS+2];
double Temperature_last[ROWS+2][COLUMNS+2];

void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {

    int i, j;
    int max_iterations;
    int iteration=1;
    double dt=100;
    struct timeval start_time, stop_time, elapsed_time;

    initialize();

    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                            Temperature_last[i][j+1] + Temperature_last[i][j-1]);
            }
        }

        dt = 0.0; // reset largest temperature change

        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
                dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
                Temperature_last[i][j] = Temperature[i][j];
            }
        }

        if((iteration % 100) == 0) {
                track_progress(iteration);
        }

            iteration++;
    }
}


void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }

    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
```

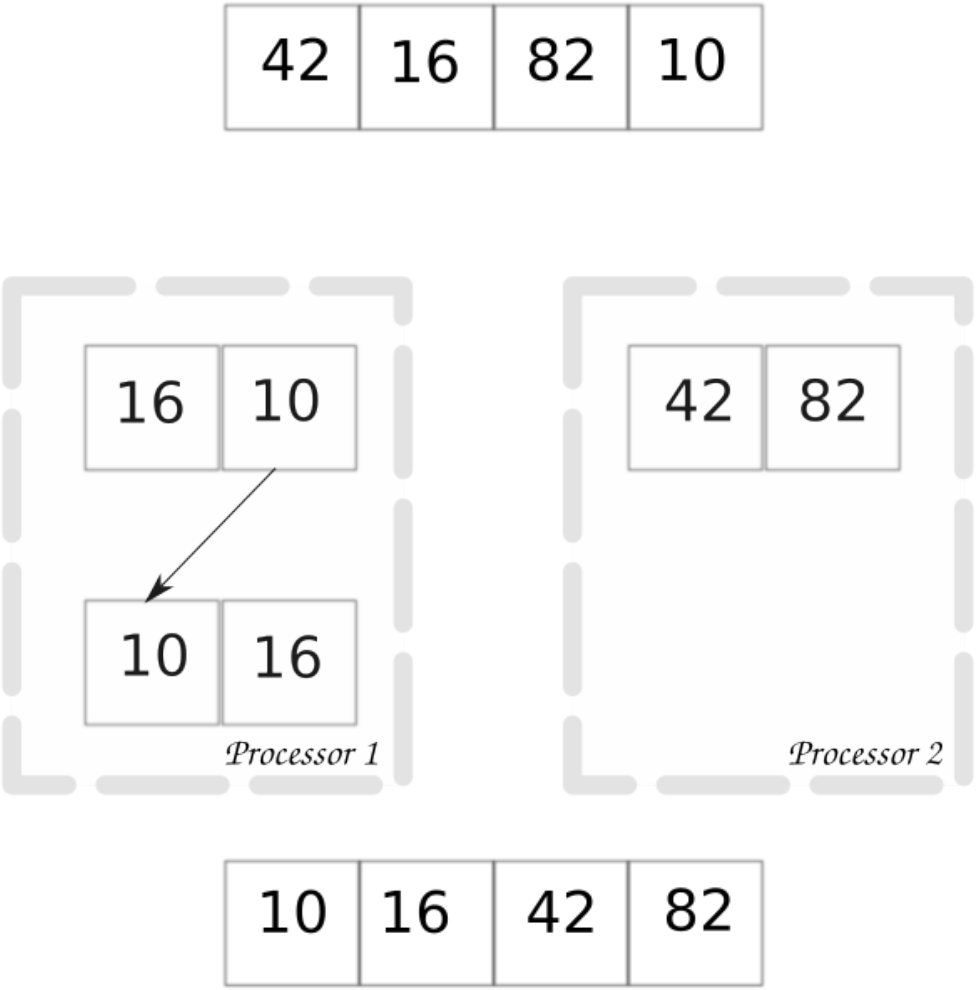
Amdahl's law

72 total lines of code
24 parallelizable lines of code

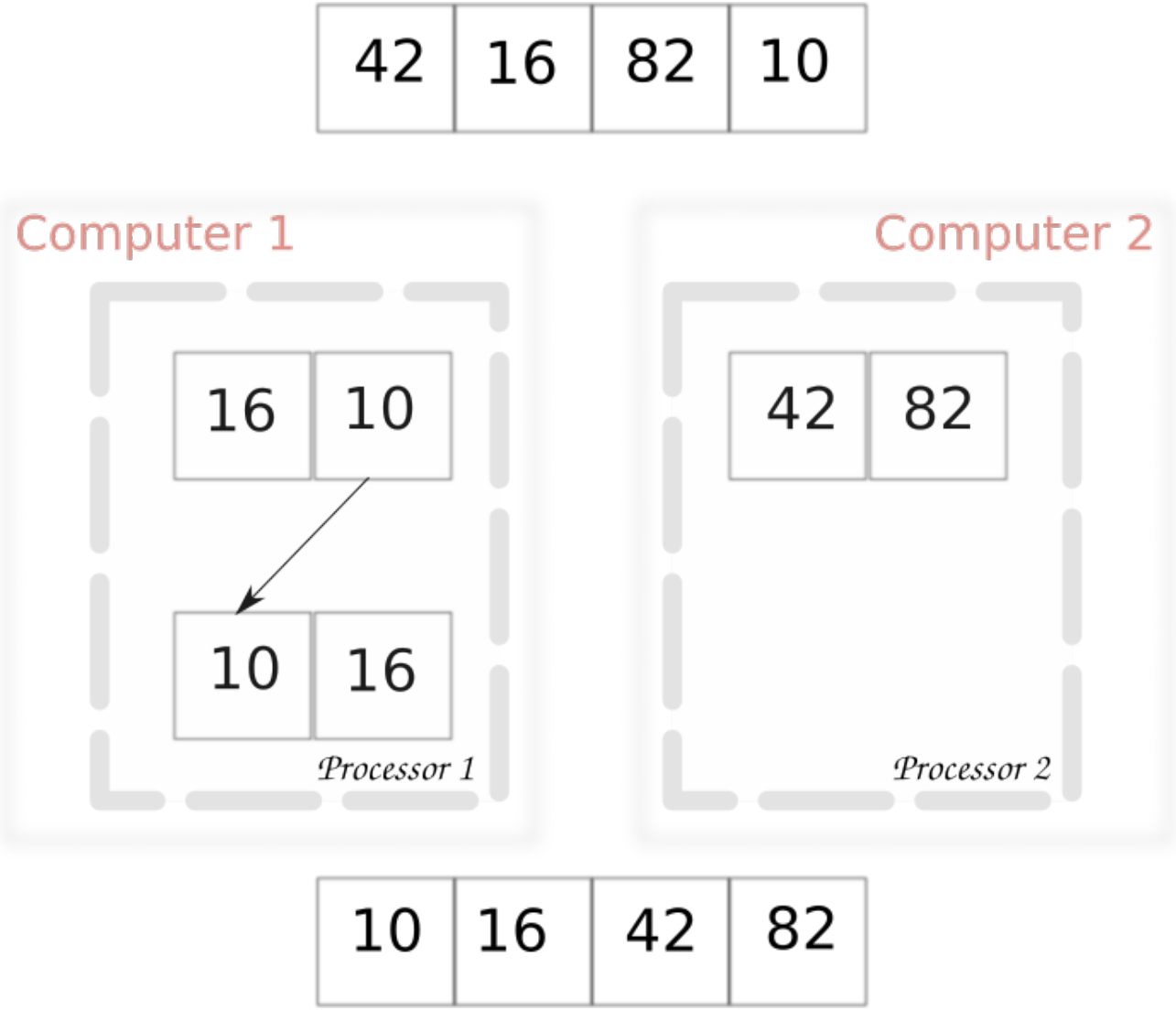
Parallel Overhead

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

#define COLUMNS    1000
#define ROWS       1000
#define MAX_TEMP_ERROR 0.01

double Temperature[ROWS+2][COLUMNS+2];
double Temperature_last[ROWS+2][COLUMNS+2];

void initialize();
void track_progress(int iter);


int main(int argc, char *argv[]) {

    int i, j;
    int max_iterations;
    int iteration=1;
    double dt=100;
    struct timeval start_time, stop_time, elapsed_time;

    initialize();
    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

        #pragma omp parallel for private (i,j) schedule(dynamic)
        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                                                    Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                                    Temperature_last[i][j+1] + Temperature_last[i][j-1]);
            }
        }

        dt = 0.0; // reset largest temperature change

        #pragma omp parallel for private (i,j) reduction(max:dt) schedule(dynamic)
        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
                                                    Temperature[i][j]), dt);
                Temperature_last[i][j] = Temperature[i][j];
            }
        }

        if((iteration % 100) == 0) {
                track_progress(iteration);
        }

            iteration++;
    }

}

void initialize(){

    #pragma omp parallel for private(i,j)
    for(i = 0; i <= ROWS+1; i++){
            Temperature_last[i][j] = 0.0;
    }

    #pragma omp parallel for private(i)
    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    #pragma omp parallel for private(j)
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
```

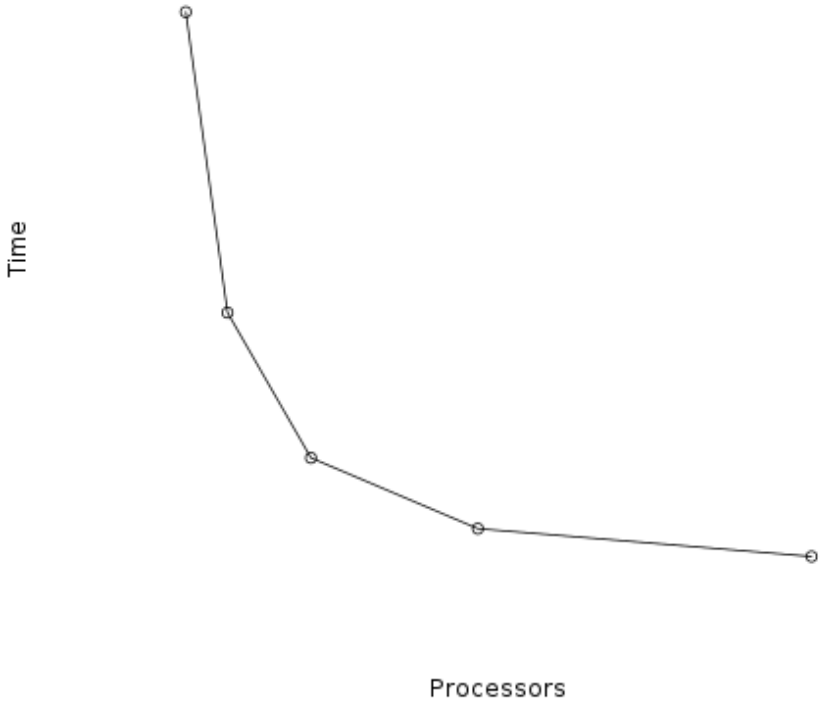
Parallel Sort



Distributed Sort



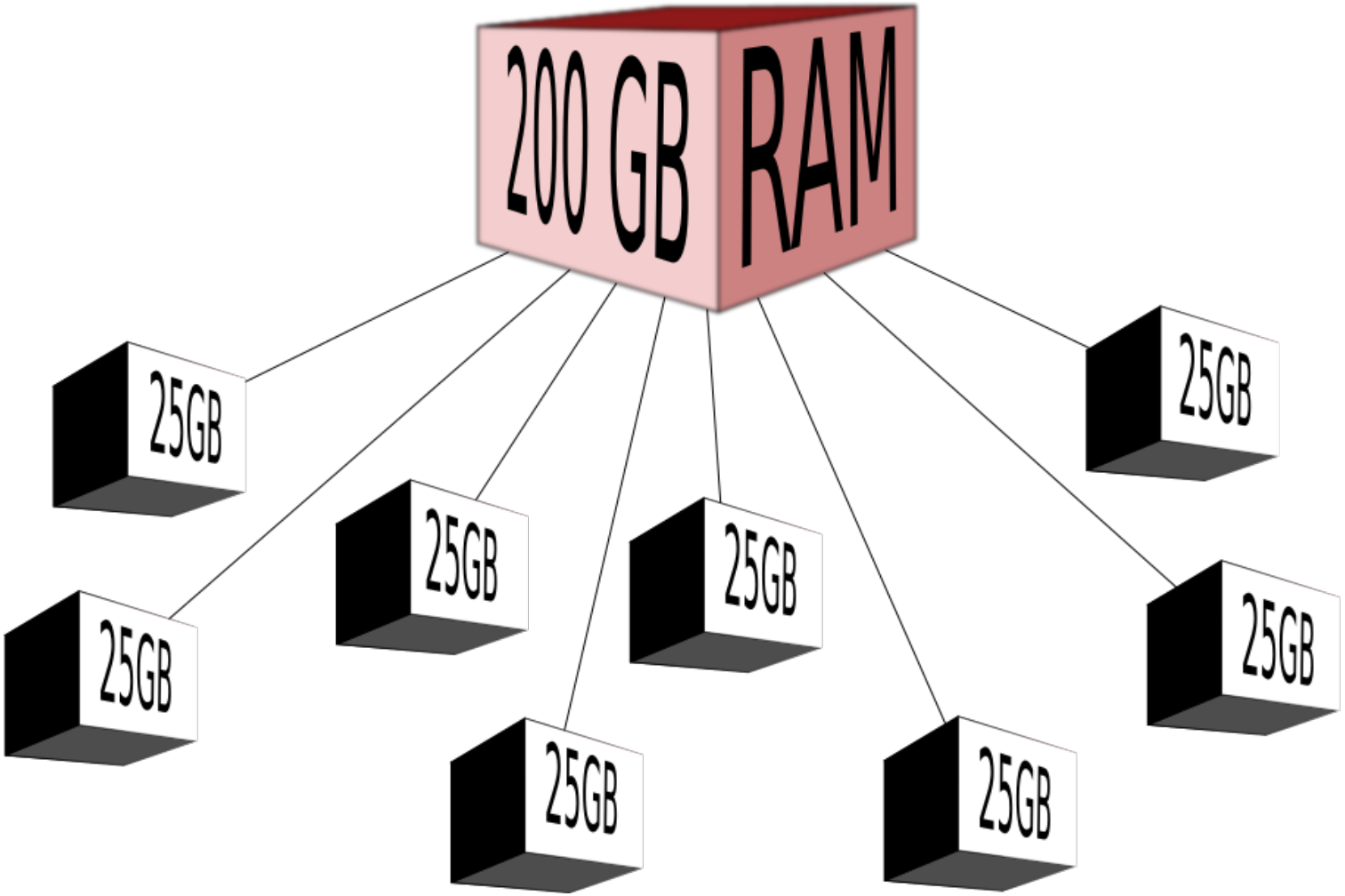
Observed Speedup



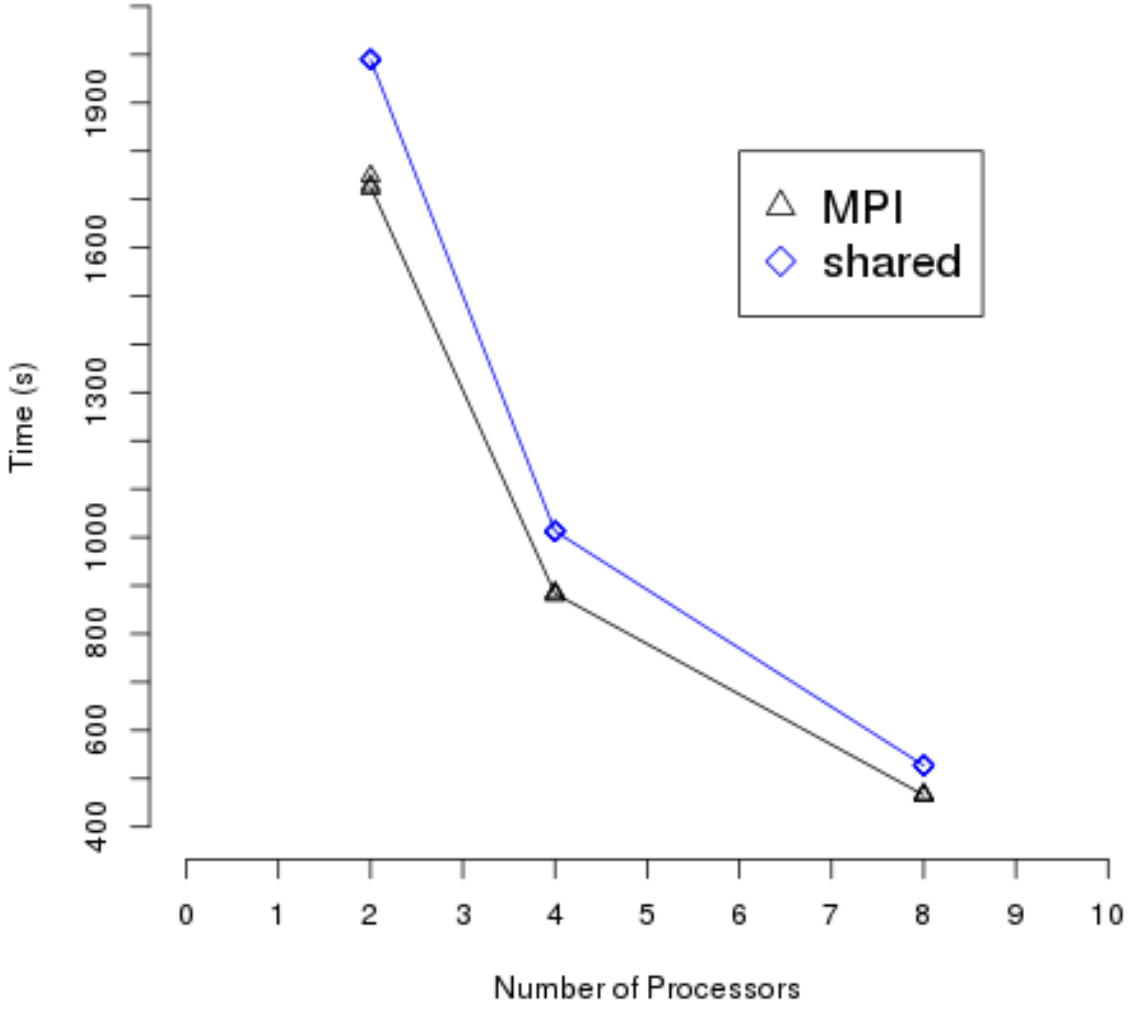
Observed Speedup

13.19x

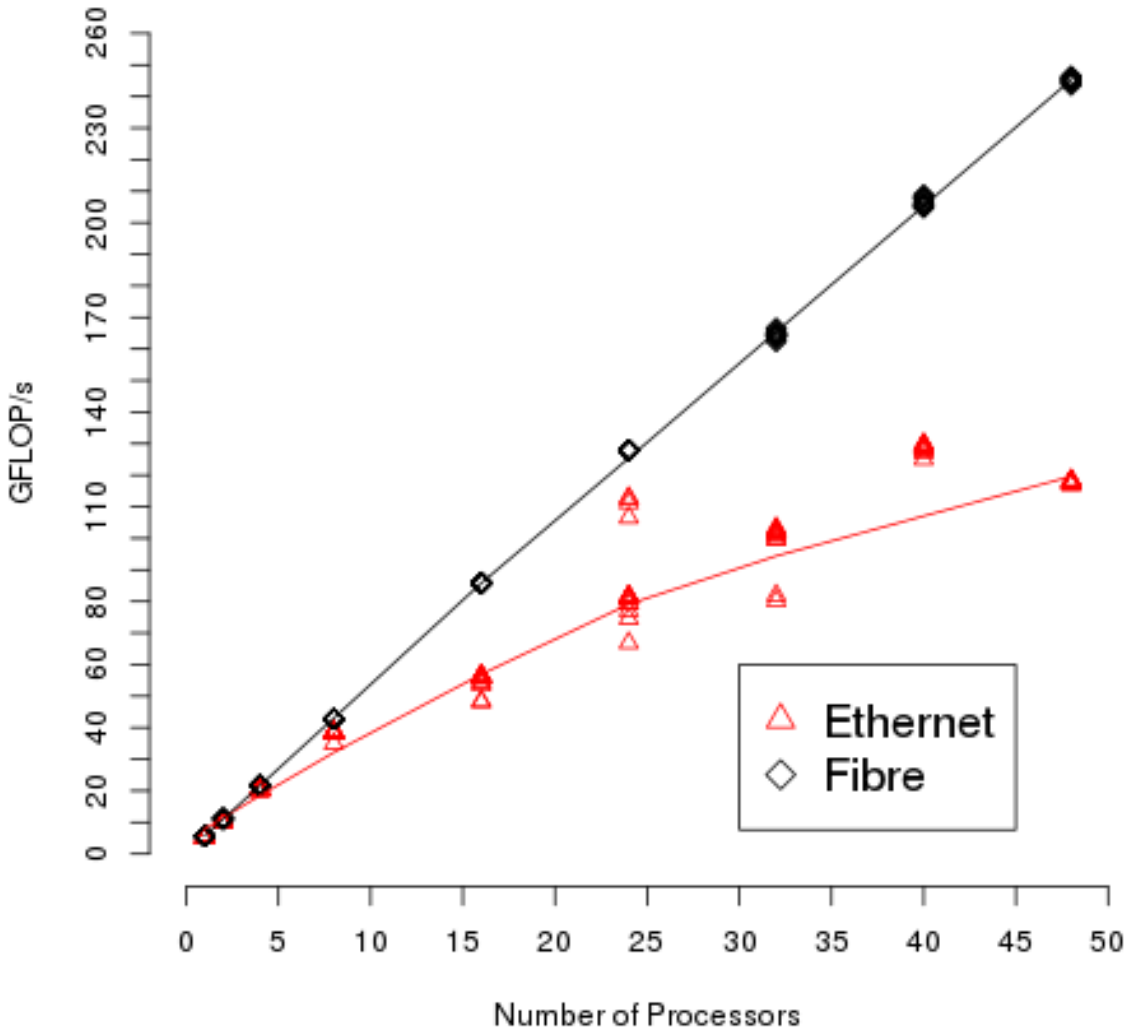
Efficient Memory Use



MPI is faster



MPI infrastructure requirements



Extreme Scientific and Engineering Discovery Environment

www.xsede.org

Online

wvuhpc.github.io/bigData