Parallel Computing
184.710
Group 5 Projects

Peter Neubauer
0725263

Bruno Pfeiffer
0717311

January 28, 2013

Contents

1 Introduction

The entire project consists of the subprojects OMP project one, two and three, Cilk project one and MPI project one, two and three. All project files (code, Makefiles, scripts, R files) and results can be found here:

`http://github.com/animiral/ParComp5`
`https://dl.dropbox.com/u/23993646/parcomp_group5.zip`

For all tests, we employed a deterministic number generator in various configurations that provided numbers based on the following conditions, given an *upper* and *lower* bound:

1. Powers of 2 $\in [lower, upper]$
2. Prime numbers $\in [lower, upper]$, at most fifty for a single query
3. Single digits, appended by trailing zeros $\in [lower, upper]$, meaning numbers of the form: `[1-9]0*`

All tests were performed on Jupiter and/or Saturn. During testing, we noticed that performance was poor in several circumstances due to other processes disturbing the execution.

2 OpenMP: Prefix Sums

For this project, we decided to implement an OpenMP program as opposed to a pthread implementation. We chose this particular option due to the simpler handling of OpenMP. All three parallel algorithms have been programmed in separate programs, each according to the guidelines found in the lecture script. The data type was generified using a C defintion ($ATYPE$). The programs accepts the arguments (n, t) to set the size of the problem and number of threads to be used. All programs (save the sequential reference implementation) use *Performance Counters* to track work:

Operation Counter Counts only the '+' operations used in the prefix sums.

Access Counter Counts every single array access (read AND write).

We found that all algorithms showed similar performance in general. The data shows that Hillis-Steele performs marginally worse than other algorithms. All implementations showed that the work is $O(n)$ for sequential reference solution (`totalsum.c`), but could not confirm that work is $O(n * \log n)$ for the parallel implementations (`recursive.c`, `iterative.c`, and `hillis-steele.c`) with regards to operations and measured time: Figures 4, 5, 6, 7 and 8 make it seem like the work is $O(n)$, which is contrary to the expected $O(\frac{n}{p} + \log n)$. You can see the specific implementation code snippets in figures 1, 2 and 3. It was noticeable that the time required for any of the algorithms amounts to ~ 3 times the time required to compute the total sum using the OpenMP reduction clause.

Testing was performed using scripts testing various array sizes provided by the described number generator. The array content wass calculated using *modulo* operations. A reference solution was integrated for debugging purposed to validate the correctness of the results. While testing with extremely large inputs ($> ca.123'000'000$) we encountered segmentation faults that could not be remedied.

Figure 1: Recursive Prefix Sums in OMP

```
static void scan(ATYPE a[], int n, int* plus_ops, int*
   acc_ops)
{

  #pragma omp parallel for reduction(+: p_ops) reduction(+:
     a_ops)
  for (i = 0; i < n/2; i++)
  {
    b[i] = a[2*i] + a[2*i+1];
    p_ops++;
    a_ops+=3;
  }

  scan(b, n/2, plus_ops, acc_ops);

  a[1] = b[0];

  #pragma omp parallel for reduction(+: p_ops) reduction(+:
     a_ops)
  for (i = 1; i < n/2; i++) {
    a[2*i] = b[i-1] + a[2*i];
    a[2*i+1] = b[i];
    p_ops++;
    a_ops+=5;
  }

  if (n % 2)
  {
    a[n-1] = b[n/2-1] + a[n-1];
  }
}
```

Figure 2: Iterative Prefix Sums in OMP

```
static void scan(ATYPE a[], int n, int* plus_ops, int*
   acc_ops)
{

  // phase 0
  for (r = 1; r < n; r = r2)
  {
    r2 = r*2;

    #pragma omp parallel for reduction(+: p_ops) reduction(+:
        a_ops)
    for (s = r2-1; s < n; s += r2)
    {
      a[s] = a[s-r] + a[s];
    }
  }

  // phase 1
  for (r = r/2; r > 1; r = r2)
  {
    r2 = r/2;

    #pragma omp parallel for reduction(+: p_ops) reduction(+:
        a_ops)
    for (s = r-1; s < n-r2; s += r)
    {
      a[s+r2] = a[s] + a[s+r2];;
    }
  }
}
```

Figure 3: Hillis-Steele Prefix Sums in OMP

```
static void scan(ATYPE** in, int n, int* plus_ops, int*
   acc_ops)
{
  for (k = 1; k < n; k<<=1)
  {

    #pragma omp parallel for reduction(+: a_ops)
    for (i = 0; i < k; i++)
    {
      b[i] = a[i];
    }

    #pragma omp parallel for reduction(+: p_ops) reduction(+:
        a_ops)
    for (i = k; i < n; i++)
    {
      b[i] = a[i] + a[i-k];
    }

    // swappity
    temp = a;
    a = b;
    b = temp;
  }

  *in = a;
}
```

Figure 4: OMP: Comparison of Algorithms at n=200'000. Red = Recursive, Blue = Iterative, Green = Hillis-Steele, Yellow = Total Sum

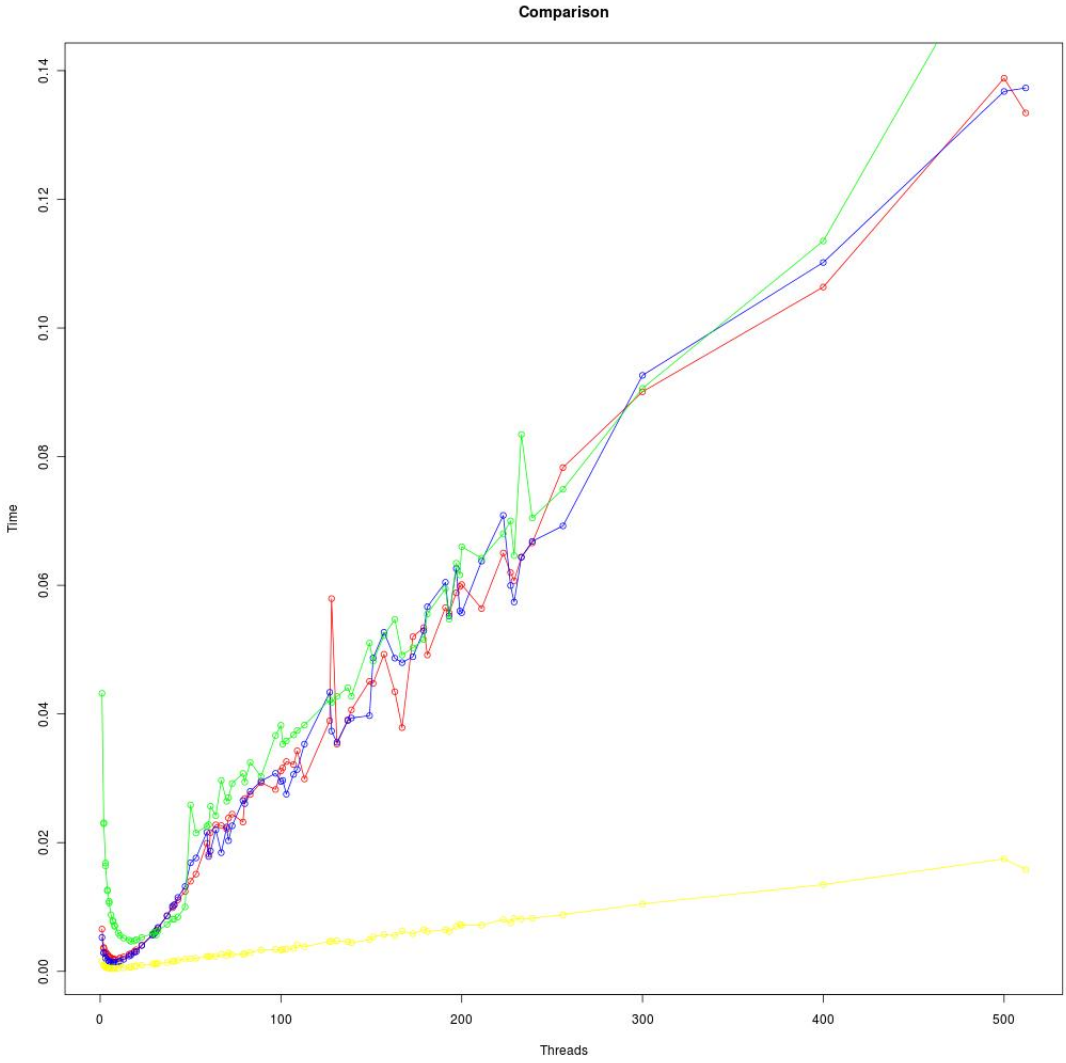


Figure 5: OMP: Comparison of Algorithms at n=10'000. Red = Recursive, Blue = Iterative, Green = Hillis-Steele, Yellow = Total Sum

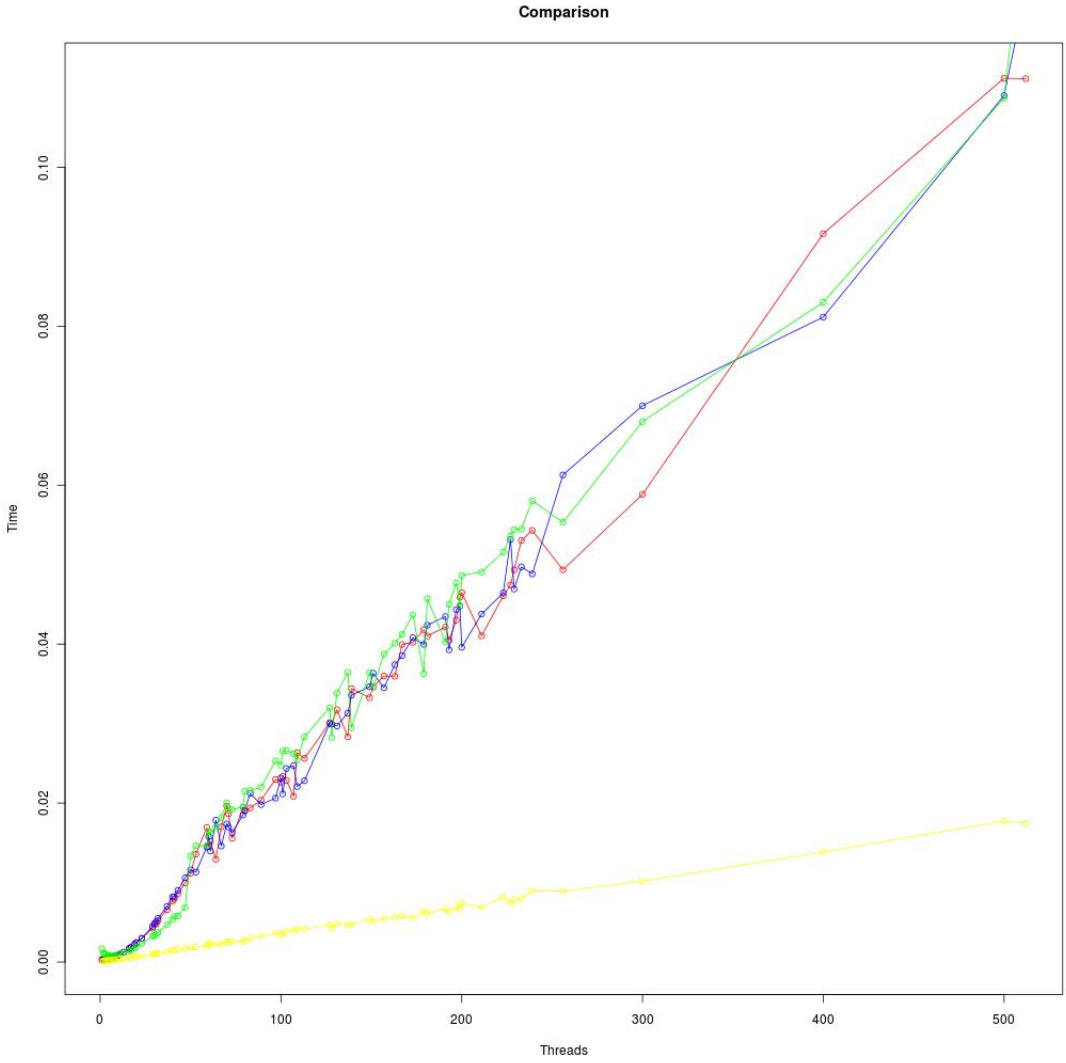


Figure 6: OMP: Comparison of Algorithms at n=256. Red = Recursive, Blue = Iterative, Green = Hillis-Steele, Yellow = Total Sum

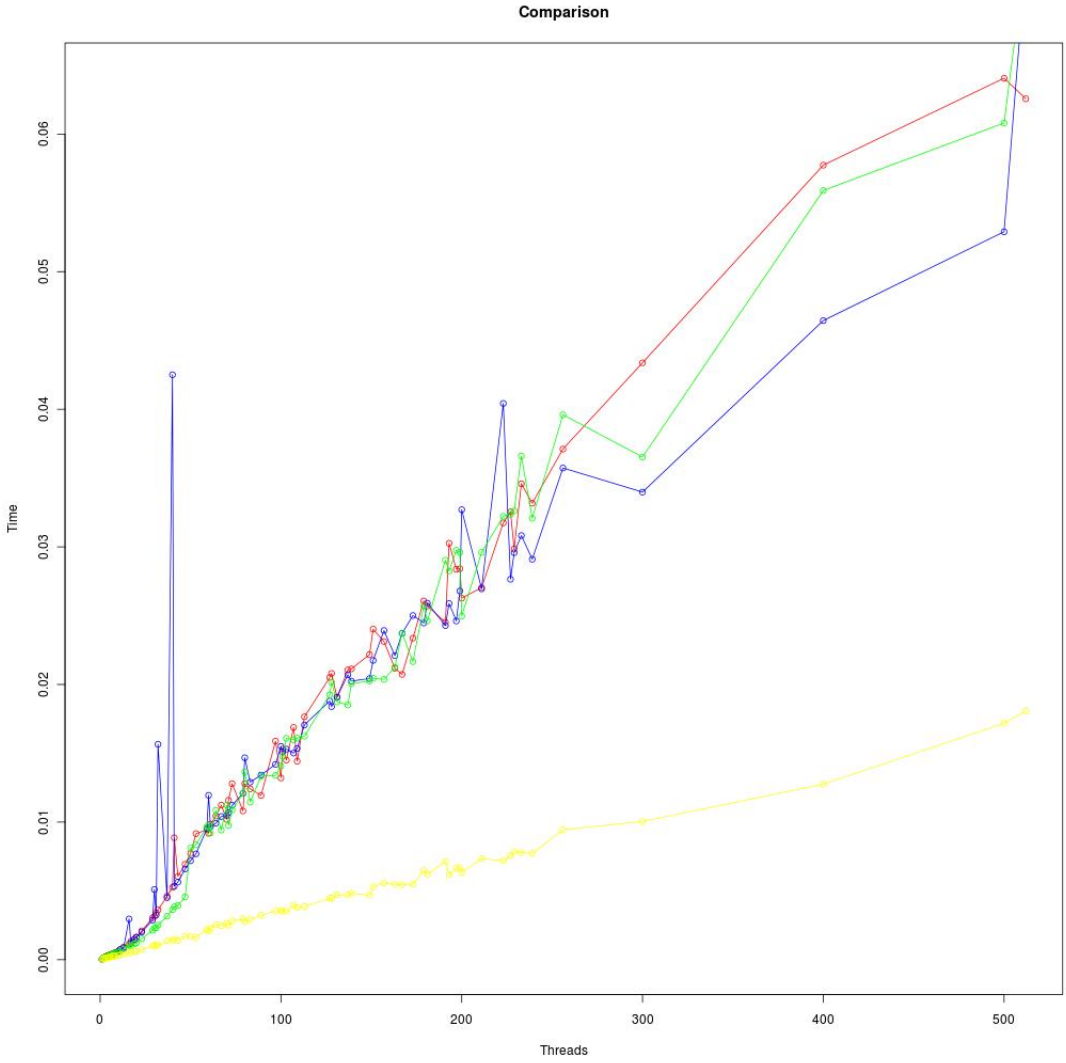


Figure 7: OMP: Iterative Algorithm. Fixed Number of Threads (2000)

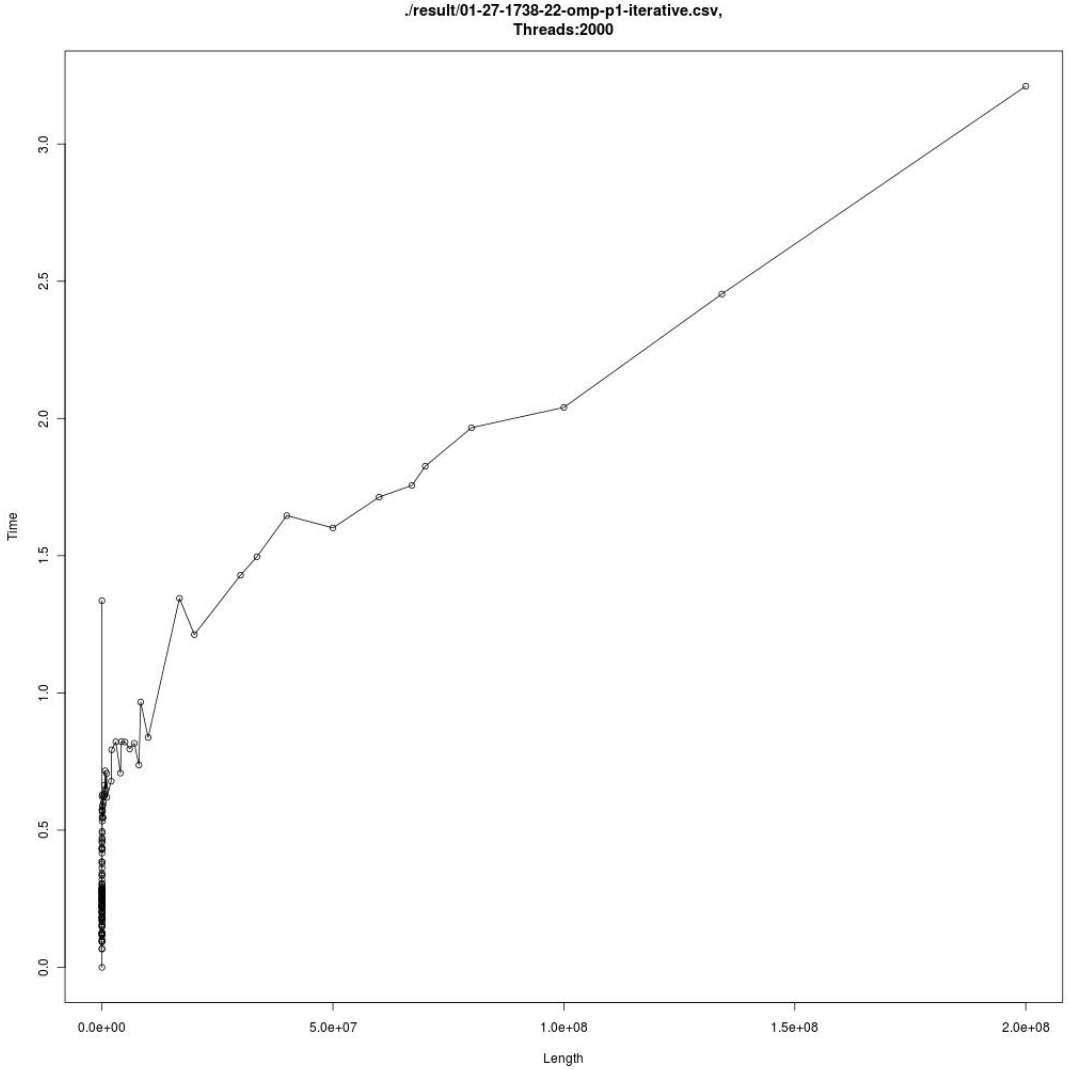
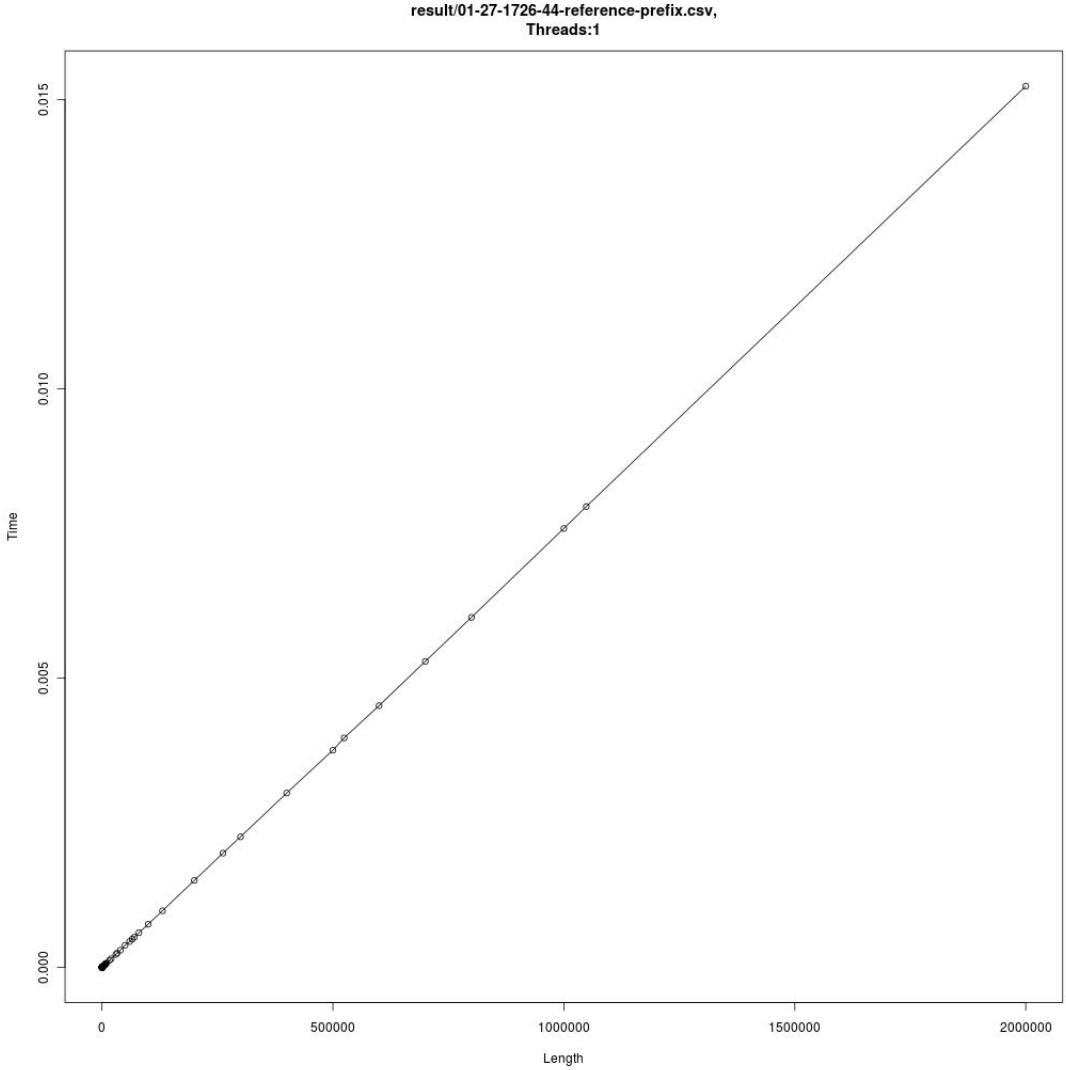


Figure 8: OMP: Sequential Algorithm. Fixed Number of Threads (1)



3 OpenMP: Matrix / Vector Multiplication

This implementation was also done in OpenMP rather than using pthreads. When calling the program, a tuple (n, m, t) is required to set the dimensions of the matrix and the vector, as well as the amount of threads that should be used to calculate the problem. Invalid input (negative threads, sizes) concludes the program with a message. Once given adequate input, the program generates a matrix & vector based on the transmitted size dimensions, once more using *modulo* operations.The implementation of the algorithm is located in figure 9.

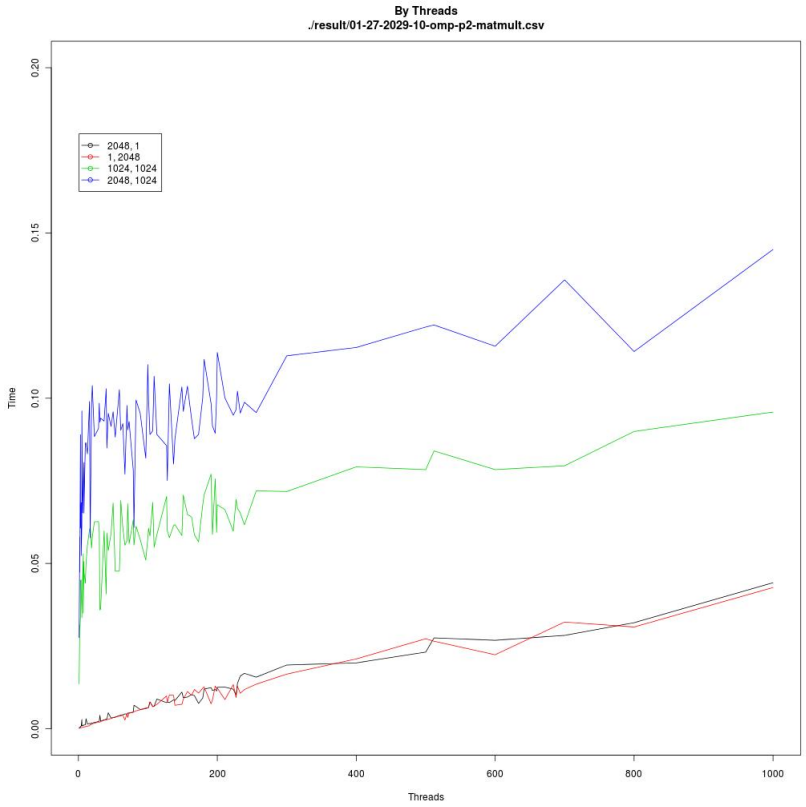
By experimenting with different Matrix input combinations, we wanted to determine how the matrix dimensioning affects performance, specifically in combination with false sharing. However, our data does not show significant deviations due to matrix dimensioning: Thus, we are not able to conclude any facts about false sharing based on matrix dimensions.

Our observations show different matrix dimensioning approaches and their calculation time for varying thread counts (please see figure 10). The data shows that the calculation time increases when using more threads; this increase may be caused by false sharing after all, but the evidence is not conclusive.

Figure 9: Matrix / Vector Multiplication in OMP

```
static void matmult(ATYPE mat[], ATYPE vec[], ATYPE* res, int
    m, int n)
{
  int i, j;

  #pragma omp parallel for schedule(static,1)
  for (i = 0; i < m; i++)
  {
    res[i] = 0;
    for (j = 0; j < n; j++)
    {
      // MINDEX returns value of A_(i, j)
      res[i] += mat[MINDEX(i,j)] * vec[j];
    }
  }
}
```

Figure 10: OMP: Prefix Sums Performance



4 Cilk: Task-parallel Prefix-Sums

We elected to implement the prefix-sum project for the Cilk category due to the experience gained with the OMP implementations. Again, the program requires an input of (n, c, t) to determine the *total size*, *chunk size* and the number of threads (of course exiting on invalid input). The code in figure 11 shows the core of the implemented algorithm.

During the implemmentation, we encountered several challenges with regards to segmentation faults, arising due to calls to C's `alloca`. The calls were replaced by Cilk's own implementation of this functionality (thread-safe): `Cilk_alloca`.

The task parallel ($c > 1$) performance was (suprisingly) observed to be vastly inferior to data parallel runs. These findings are largely independent of input size n and thread count t and can be found in figures 12, 13, 14. The graphs all commonly show that data parallel runs benefit from parallelilzation and provide constant run-times. The task parallel measurements are *consistently erratic*, although they do sometimes show a trend toward decreasing run-times with growing thread count.

Figure 11: Task-parallel Prefix-Sums in Cilk

```
cilk static void scan(ATYPE* in, int length, int chunk)
{
  int start;
  int count;
  int half;
  ATYPE* out;

  if (length <= 1) return;

  half = length/2;
  out = Cilk_alloca(sizeof(ATYPE) * half);

  for (start = 0; start < half; start += chunk)
  {
    count = chunk;
    if (start + chunk > half) count = half-start;

    spawn add_up(in, out, start, count);
  }
  sync;

  spawn scan(out, half, chunk);
  sync;

  in[1] = out[0];

  for (start = 1; start < half; start += chunk)
  {
    count = chunk;
    if (start + chunk > half) count = half-start;

    spawn reduce_down(in, out, start, count);
  }
  sync;

  if (length % 2)
  {
    in[length-1] = out[half-1] + in[length-1];
  }
}
```

Figure 12: Cilk: Prefix Sum Performance, n=500

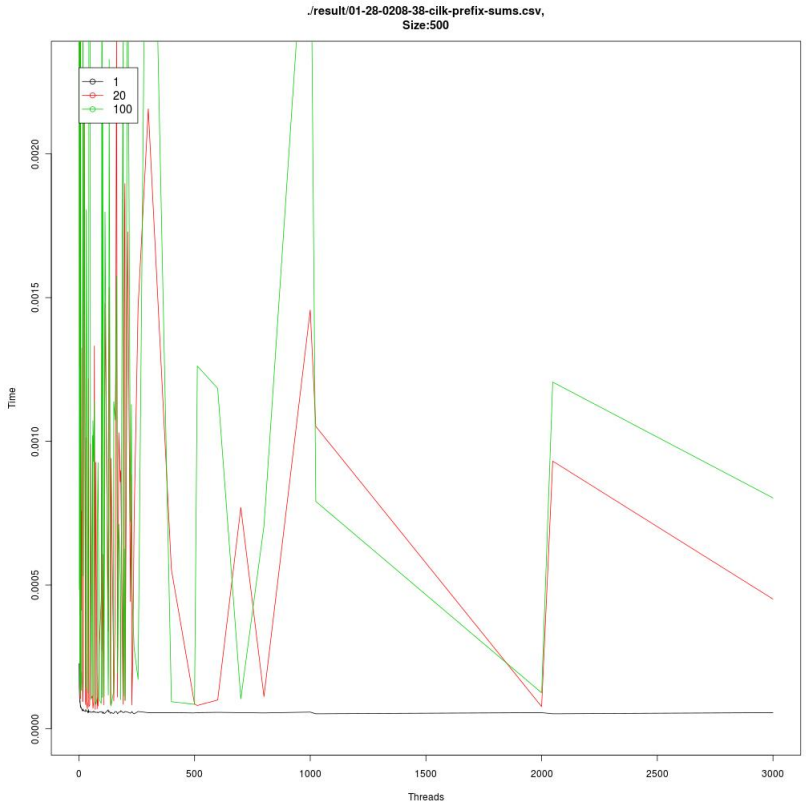


Figure 13: Cilk: Prefix Sum Performance, n=10'000

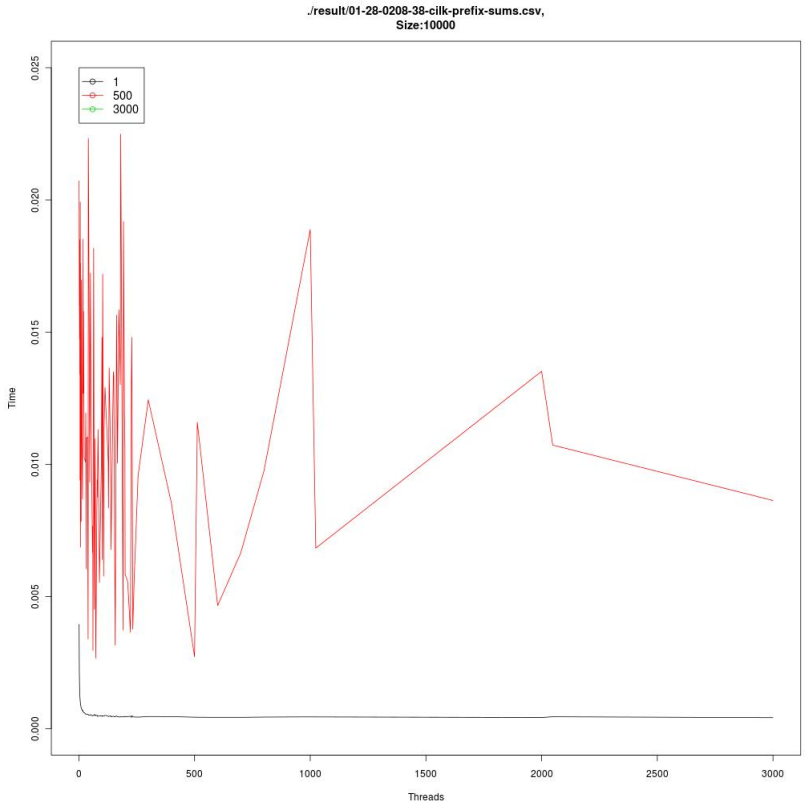
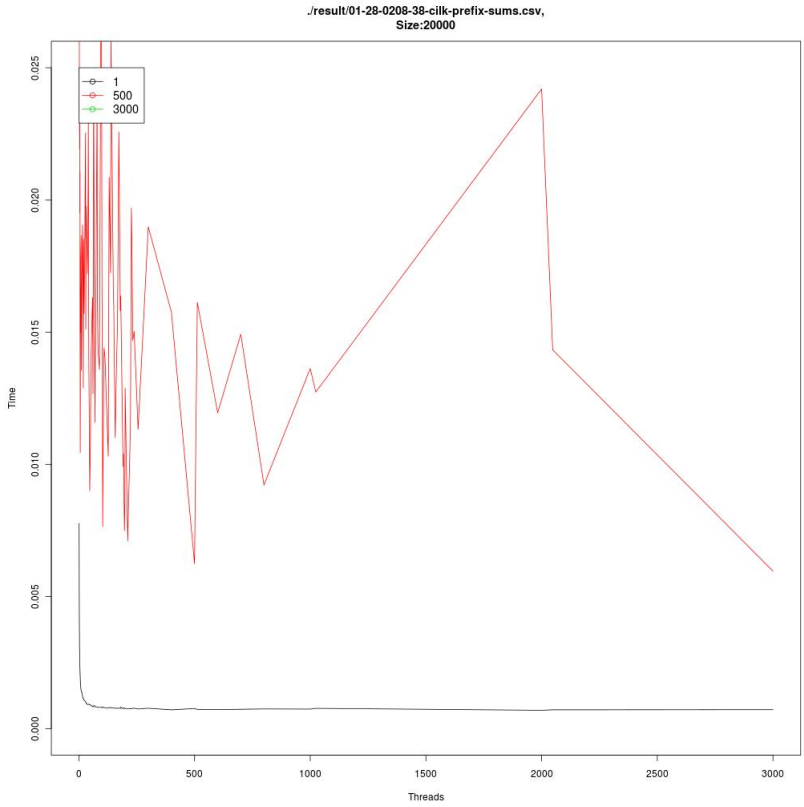


Figure 14: Cilk: Prefix Sum Performance, n=20'000



5 MPI: Stencil Operations

The stencil operation was first implemented and tested locally using *Open-MPI* 32-Bit, then re-compiled using *NECmpi* 64-Bit on Jupiter. Instead of using `MPI_Send` or `MPI_Recv`, we opted to use the non-blocking `MPI_ISend` and `MPI_IRecv`. The reasoning behind this approach is that the program can compute the local center of the submatrix while waiting for processes to exchange data concerning the external ring, leading to more efficient use of the CPU and idle times. This implementation supercedes a previous implementation using blocking `Send / Recv` that should show vastly improved performance to its predecessor.

Based on the algorithm in the implementation, we derived that the running time can be calculated as follows (please see figure 15 for code extract):

$$C_{par} \cdot (\max(T_{StencilMid}, T_{Comm}) + T_{StencilRing}) \tag{1}$$

$$C_{par} = O\left(\max(\frac{r \cdot c}{p}, 1)\right) \tag{2}$$

$$T_{StencilMid} = O\left(\frac{m}{c} \cdot \frac{n}{r}\right) \tag{3}$$

$$T_{Comm} = O\left(\frac{m}{r} + \frac{n}{c}\right) \tag{4}$$

$$T_{StencilRing} = O\left(\frac{m}{r} + \frac{n}{c}\right) \tag{5}$$

C_{par} is a factor that increases the required time, depending on whether enough cores are available to distribute the $(r \cdot c)$ work blocks evenly and process them all in parallel. If not enough cores are available, the time increases. $T_{StencilMid}$ is the time needed to calculate the stencil values for the center cells of a submatrix (where no communication with other processes is required) and T_{Comm} represents the time needed to communicate the data with neighboring processes. $T_{StencilRing}$ is the time required to run remaining stencil calculations after new data has been received.

To achieve maximum parallellization under this model, we should choose r and c in a way so that $rc = p$, leading to the lowest $C_{par} = 1$. We can now consider the following scenarios:

1. (Block format) $r = c = \sqrt{p}$
2. (Column Format) $r = 1, c = p$

3. (Row Format) $r = p, c = 1$

We can conclude that $T_{StencilMid}$ is not affected by the choice of r and c, since $p = rc$, leading to $O(\frac{mn}{p})$. We therefore aim to minimize T_{Comm} to the point of $T_{Comm} \leq T_{StencilMid}$. Determing which distribution (block vs. column) is optimal could not be determined analytically; we therefore turned to our empirical measurements for clarification.

The implementation was tested with various matrix sizes and thread counts. Presented in figures 16, 17, 18 and 19 are the extreme cases. The graphs show the runtime with regard to distribution of rows & columns at individual nodes, with block format in the middle (x-axis), column format on the left and row format on the right. The results did not show a clear general case advantage of a specific format over the others. However, for specific matrix sizes and thread counts, there seem to be distinct differences: For example, $(n = 16, p = 16)$ shows an advantage for block format rather than column/row formatting, while $(n = 512, p = 16)$ shows the exact opposite. Tests with larger thread counts seem to give the row format an advantage in general.

Figure 15: Stencil Operation in MPI

```
/*
  Terminology:
  m: Number of rows in matrix,          e.g. 15
  n: Number of columns in matrix,       e.g. 8
  r: Number of block rows,              e.g. 5
  c: Number of block columns,           e.g. 4
  h: Height (rows) of a single block,   e.g. 3
  w: Width (columns) of a single block, e.g. 2
  i: Block Row index                    (0-4)
  j: Block Column index                 (0-3)
  x: Row index in matrix                (0-14)
  y: Column index in matrix             (0-7)
 */
// start indices of this node's submatrix
x = j * w;
y = i * h;
// allocate my submatrix (twice)
ATYPE* source = xmalloc((w+2) * (h+2) * sizeof(ATYPE));
ATYPE* dest = xmalloc(w * h * sizeof(ATYPE));
genmatrix(source, x, y, w, h, n);

int ret;
MPI_Request request[8];
MPI_Datatype column_type;
MPI_Type_vector(h, 1, w+2, ATYPE_MPI, &column_type);
MPI_Type_commit(&column_type);

int l_peer = rank-1; if (j <= 0)   l_peer = MPI_PROC_NULL;
int r_peer = rank+1; if (j >= c-1) r_peer = MPI_PROC_NULL;
int u_peer = rank-c; if (i <= 0)   u_peer = MPI_PROC_NULL;
int d_peer = rank+c; if (i >= r-1) d_peer = MPI_PROC_NULL;

ret = MPI_Isend(&source[(w+2)+1],       1, column_type,
   l_peer, LEFT,  MPI_COMM_WORLD, &request[0]);
ret = MPI_Isend(&source[(w+2)+w],       1, column_type,
   r_peer, RIGHT, MPI_COMM_WORLD, &request[1]);
ret = MPI_Isend(&source[(w+2)+1],       w, ATYPE_MPI,
   u_peer, UP,    MPI_COMM_WORLD, &request[2]);
ret = MPI_Isend(&source[h*(w+2)+1],     w, ATYPE_MPI,
   d_peer, DOWN,  MPI_COMM_WORLD, &request[3]);

ret = MPI_Irecv(&source[(w+2)],         1, column_type,
   l_peer, RIGHT, MPI_COMM_WORLD, &request[4]);
ret = MPI_Irecv(&source[(w+2)+w+1],     1, column_type,
   r_peer, LEFT,  MPI_COMM_WORLD, &request[5]);
ret = MPI_Irecv(&source[1],             w, ATYPE_MPI,
   u_peer, DOWN,  MPI_COMM_WORLD, &request[6]);
ret = MPI_Irecv(&source[(h+1)*(w+2)+1], w, ATYPE_MPI,
   d_peer, UP,    MPI_COMM_WORLD, &request[7]);

stencil_middle(source, dest, w, h);
ret = MPI_Waitall(8, request, MPI_STATUS_IGNORE);
stencil_ring(source, dest, w, h);
```

Figure 16: Various (r, c) Process Distributions at $n = 16$, $p = 16$

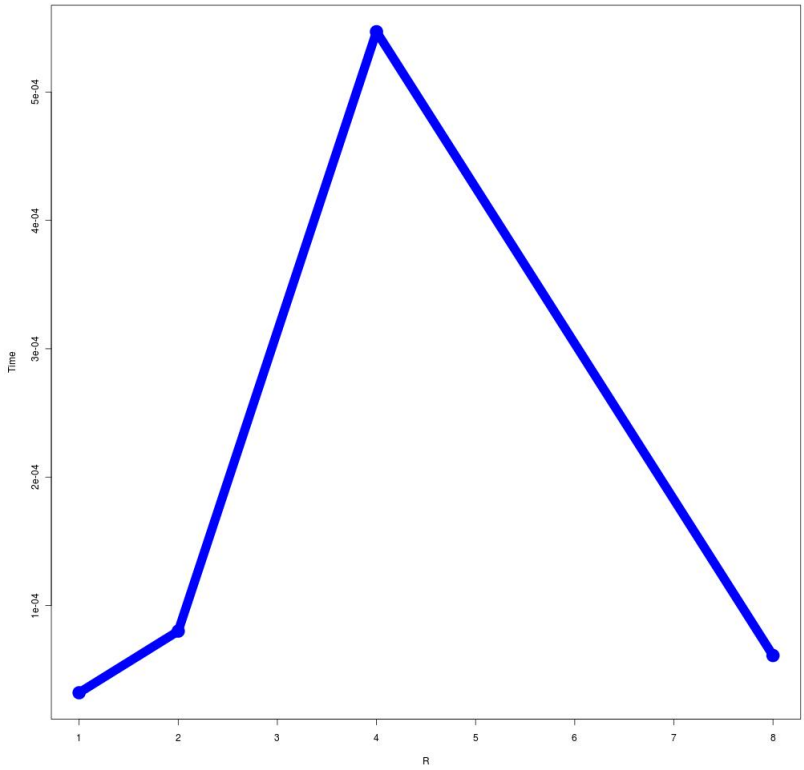


Figure 17: Various (r, c) Distributions at $n = 16$, $p = 128$

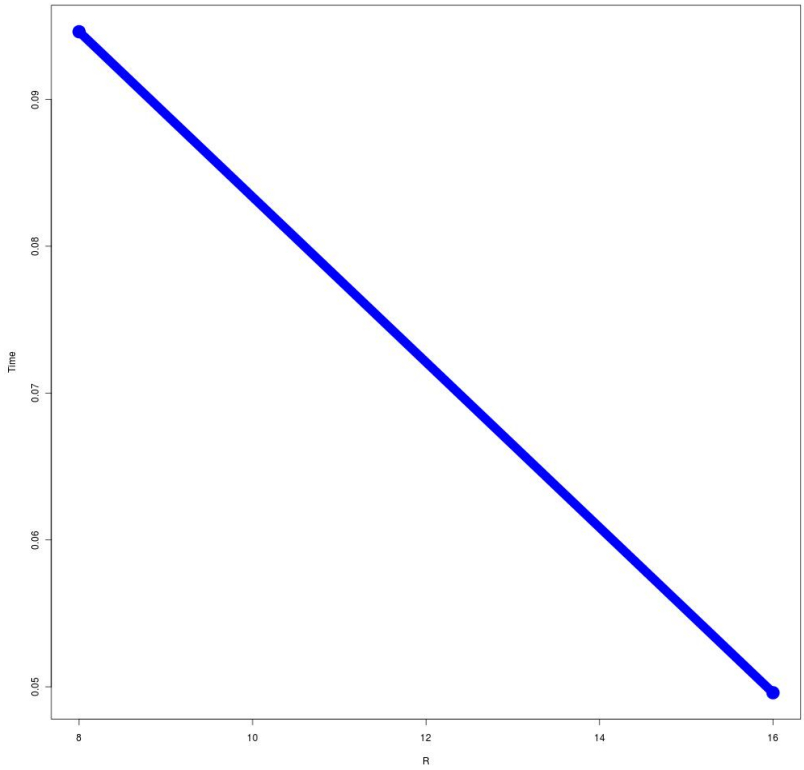


Figure 18: Various (r, c) Distributions at $n = 512$, $p = 16$

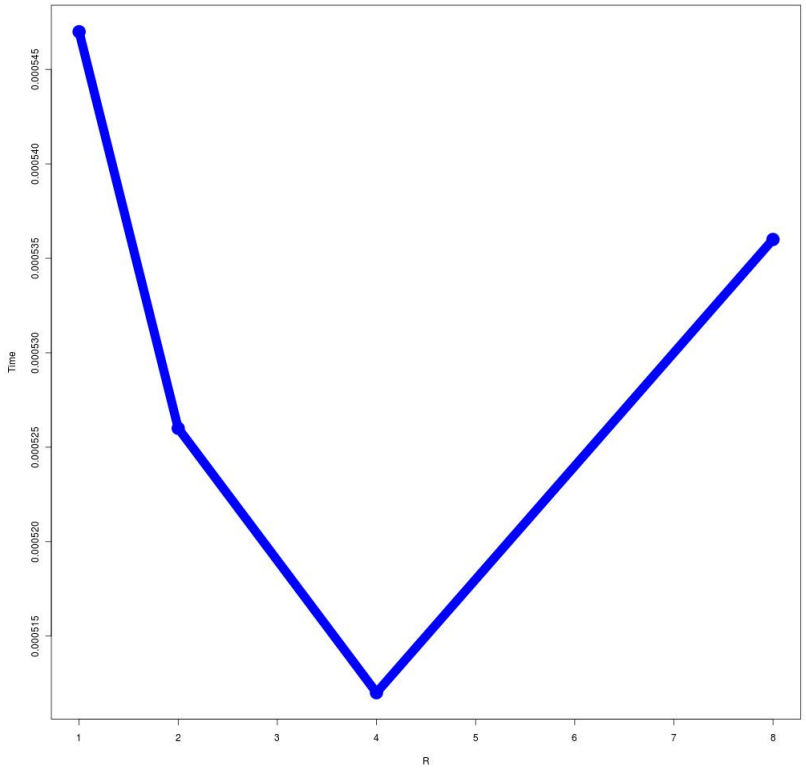
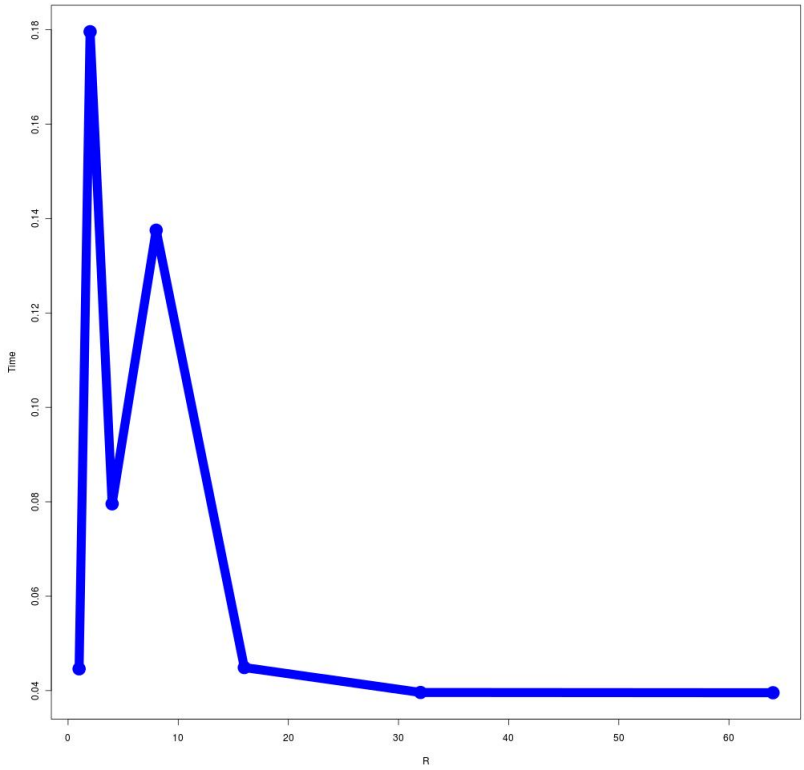


Figure 19: Various (r, c) Distributions at $n = 512$, $p = 128$



6 MPI: Prefix Sums

The algorithm was implemented using the `MPI_Sendrecv` to realize the *Hillis-Steele* algorithm with communication.The implemented algorithm differs from MPI's standard algorithm in that MPI's version allows for further consideration of nodes / ranks with regard to influencing factors such as locality, which the presented algorithm does not take into account.

We have established correctness of the algorithm via integrated reference function and also other project results (see OMP).

The running time of the algorithm $R(n, p)$ is calculated as below, given the following assumptions:

1. The network throughput is constant, i.e. there is no network congestion. Two communicating nodes can always transmit data at constant speed.

2. T_{comm} shows linear growth relative to the amount of data to be transmitted.

$$T_{par} = T_{Iteration} * Iterations \tag{6}$$
$$Iterations = O(\log n) \tag{7}$$
$$T_{Iteration} = T_{comm} + T_{ops} \tag{8}$$
$$T_{comm} = O\left(\frac{n}{p}\right) \tag{9}$$
$$T_{ops} = O\left(\frac{n}{p}\right) \tag{10}$$

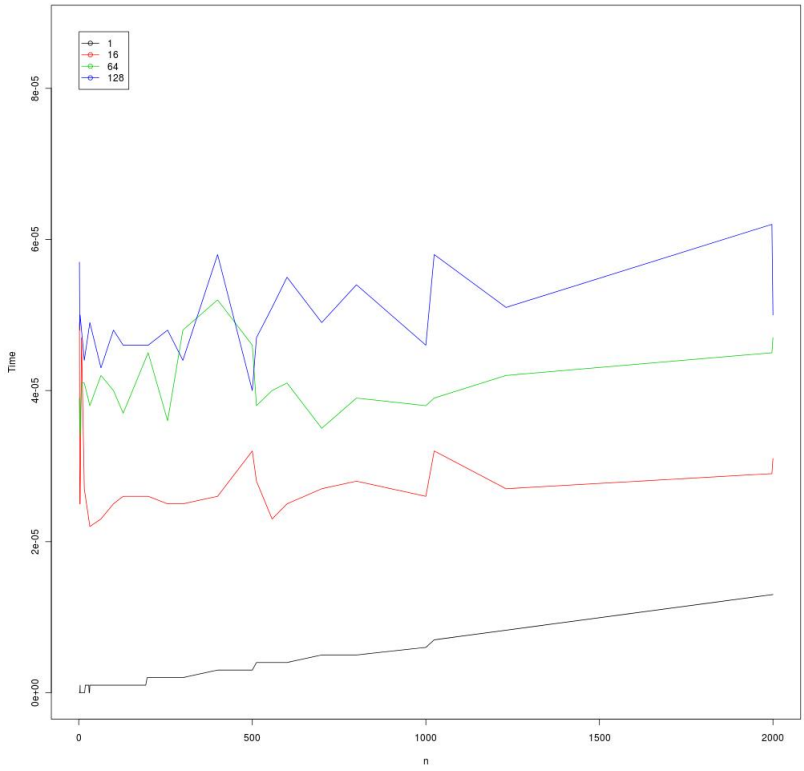
Observations in figure 21 show the run time in n, comparing different process counts alongside the sequential reference performance. In our experiments, the parallel algorithms could not outperform the sequential algorithm. We are assuming that this is due to the (regrettably) small input sizes used for benchmarking; at the time of testing, Jupiter could not complete a run with the desired number of elements in a reasonable time frame.

With the given range of measurements, it is not safe to make a statement concerning the accuracy of the above described model.

Figure 20: Prefix sum in MPI

```
static void dist_sum(int rank, int p, ATYPE b, ATYPE* prefix)
{
  ATYPE buf;
  *prefix = 0;

  for (int k = 1; k < p; k<<=1)
  {
    int rpeer = rank - k;
    int speer = rank + k;

    if (speer >= p)
    {
      speer = MPI_PROC_NULL;
    }

    if (rpeer < 0)
    {
      buf = 0;
      rpeer = MPI_PROC_NULL;
    }

    int ret = MPI_Sendrecv(&b, 1, ATYPE_MPI, speer, k, &buf,
       1, ATYPE_MPI, rpeer, k, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);

    *prefix += buf;
    b += buf;
  }
}
```

Figure 21: MPI-Prefix Sums: Development of various process counts



7 MPI: Matrix / Vector Multiplication

Both the `MPI_Allgather` and `MPI_Reduce_scatter` algorithms were used to calculate their performance relation. Testing was done using already existing infrastructure from previous projects. The implementation makes no restrictions on the dimensions; specifically, the generated matrix does not need to be square for the algorithm to work. However, the program does require that both matrix dimensions are divisible by p (if not, error message is given).

The correctness verification was achieved using an integrated reference function in the programs, which computed a solution based on the same input using a sequential algorithm.

Our measurements point out the fact that the `Allgather` implementation requires less time to perform the calculations, especially with growing input size. The difference seems to be about a factor of two in the experiment of size $n = m = 512$. Graphs are found in figures 22, 23, 24 and 25.

Figure 22: Comparison of Allgather vs. Scatter, $n = 64$

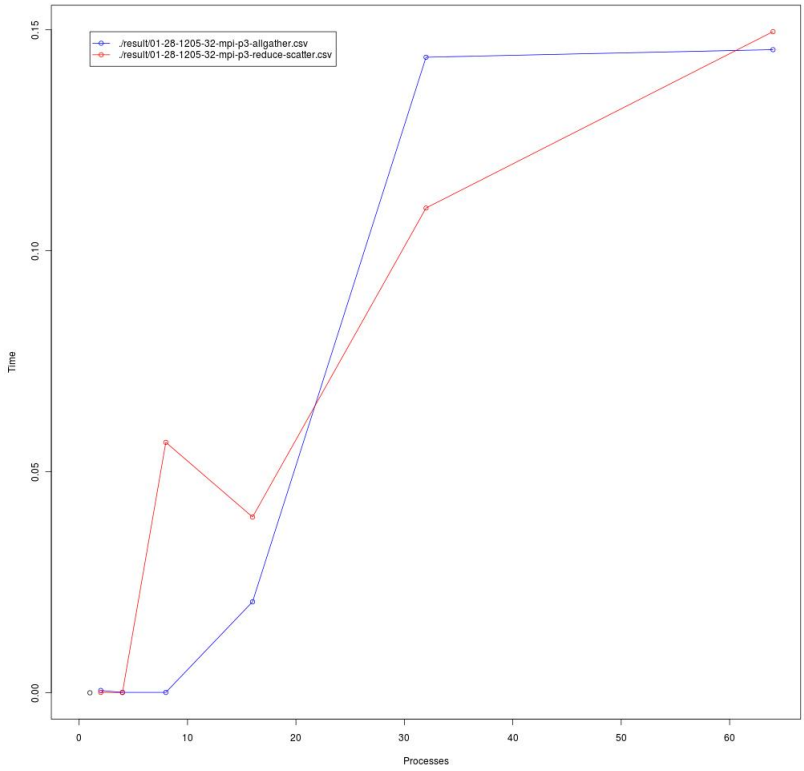


Figure 23: Comparison of Allgather vs. Scatter, $n = 128$

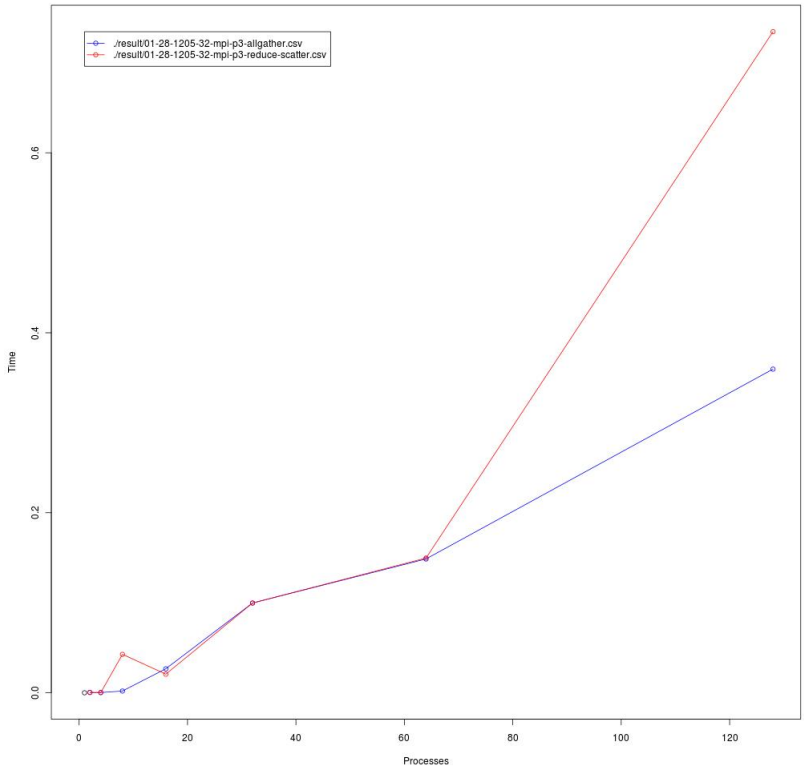


Figure 24: Comparison of Allgather vs. Scatter, $n = 256$

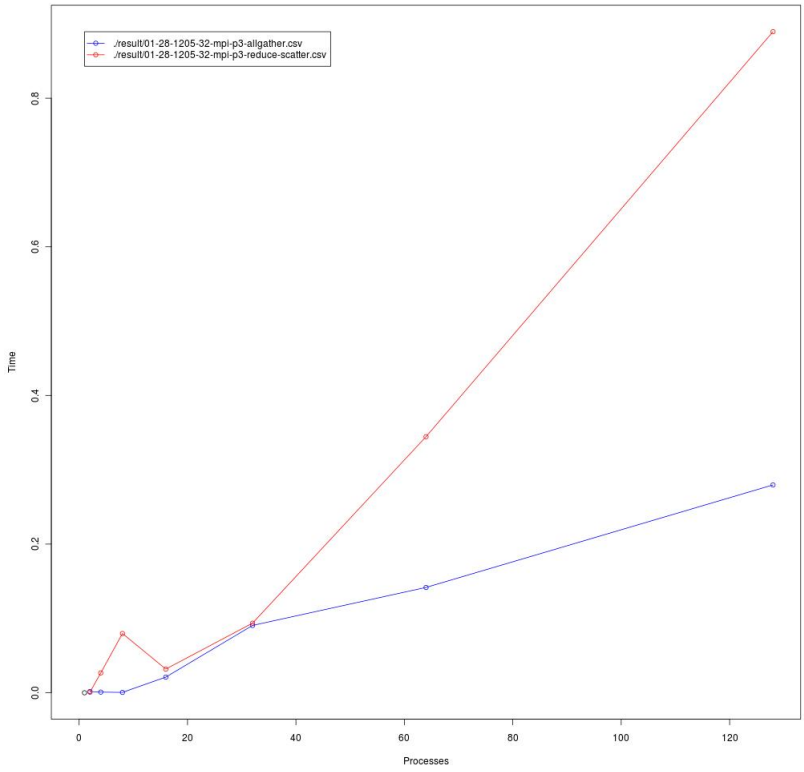


Figure 25: Comparison of Allgather vs. Scatter, $n = 512$