High Performance Computing Proseminar 2024
Assignment 3

Stefan Wagner & Sebastian Bergner
Team: Planning Desaster

October 26, 2024

The goal of this assignment is to improve the 2D heat stencil application.

Exercise 1

This exercises consists in improving the 2D heat stencil application, fixing any existing bugs and/or improving its performance.
Tasks

- If you need to fix errors, run through the usual debugging workflow detailed in the lecture on debugging:

  i enable compiler warnings

  ii check with sanitizers

  iii run with a debugging tool of your choice (a working installation of MUST for debugging MPI applications on LCC3 is provided in `/scratch/c703429/software/must-1.9.1`)

  We started using a functional version, thus omitting this first task.

- If you are looking to improve performance, do a detailed performance analysis. Use either tools discussed in the lecture (perf, gprof, gperftools, etc.) or any tools you deem fit for generating a performance profile. Provide a report that discusses the most expensive source code locations ("hot spots") along with explaining why they are expensive and how to possibly improve on that. Compare blocking and non-blocking if possible, as well as 1D and 2D if you get bored otherwise.

  We did a bunch of testing, first we analyzed the performance of builds compiled with `-O0` using gprof and perf, but we discussed and found it would make sense to measure the performance (or rather behavior) of the program when compiled with `-O3`.

Gprof output sequential -O3

The results of the sequential analysis are not helping much, besides finding out that most of the time is spent in the main function (which most probably is the computation itself).

```
Each sample counts as 0.01 seconds.
cumulative   self              self     total
time   seconds   seconds    calls  Ts/call  Ts/call  name
97.64     27.73    27.73                             main
0.00      27.73     0.00        6     0.00     0.00  print_temperature
0.00      27.73     0.00        2     0.00     0.00  create_matrix
0.00      27.73     0.00        1     0.00     0.00  data_to_csv
```

Gprof output parallel blocking -O3

For the blocking program we can observe that there are some more functions. `calc_index_supermatrix` is being called quite often, but has low impact ($< 1\%$).

```
Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  Ts/call  Ts/call  name
100.24      7.16     7.16                             main
0.00        7.16     0.00   737280     0.00     0.00  calc_index_supermatrix
0.00        7.16     0.00        8     0.00     0.00  create_matrix
0.00        7.16     0.00        8     0.00     0.00  create_vector
0.00        7.16     0.00        6     0.00     0.00  print_temperature
0.00        7.16     0.00        5     0.00     0.00  gather_data_from_submatrices
0.00        7.16     0.00        1     0.00     0.00  calc_rank_factors
0.00        7.16     0.00        1     0.00     0.00  data_to_csv
```

Gprof output parallel non-blocking -O3

For the first time, we can observe a drastic difference. The non-blocking program has been structured differently, moving the computation inside a separate function. Thus making the computation more distinguishable. But identifying "hot-spots" besides the computation remains hard (if not impossible).

```
Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
99.50       6.77     6.77   115200     0.06     0.06  propagate_heat
0.44        6.80     0.03                             main
0.15        6.81     0.01        6     1.67     1.67  print_temperature
0.15        6.82     0.01        1    10.02    10.02  data_to_csv
0.00        6.82     0.00   737280     0.00     0.00  calc_index_supermatrix
0.00        6.82     0.00        8     0.00     0.00  create_matrix
0.00        6.82     0.00        8     0.00     0.00  create_vector
0.00        6.82     0.00        5     0.00     0.00  gather_data_from_submatrices
0.00        6.82     0.00        1     0.00     0.00  calc_rank_factors
```

So looking at the above outputs, we cannot conclude much more than that the computation most probably takes up the majority ($\geq 99\%$) of the execution time.

Perf output -O0

Program	L1d load misses	L1d loads	LLC load misses	LLC loads	Time (s)	User time (s)	Sys time (s)
2D_seq_debug 384	1,420,684,620	271,673,069,550	10,609	27,885,185	100.128659	99.884303	0.002983
2D_par_blocking_debug 384	263,918,449	196,663,072,308	130,081	25,743,849	97.633189	97.267849	0.051793
2D_par_non_blocking_debug 384	276,756,820	197,688,716,869	117,548	28,575,851	96.544740	96.110856	0.067598

Table 1: Perf measurement -O0

Perf output -O3

Program	L1d load misses	L1d loads	LLC load misses	LLC loads	Time (s)	User time (s)	Sys time (s)
2D_seq_debug 384	1,418,812,102	28,372,084,936	3,399	27,291,708	22.882091899	22.806702000	0.002974000
2D_par_blocking_debug 384	261,226,191	8,665,614,974	84,869	16,154,820	7.556716632	7.374099000	0.038544000
2D_par_non_blocking_debug 384	275,399,148	8,702,055,510	85,964	18,602,539	6.374551468	6.190521000	0.043667000

Table 2: Perf measurement -O3

TODO description HIGH L1 load misses

- Improve your stencil application using the instructions above. When you do, document the improvement you managed to achieve. If you don't, argue why you failed or think you hit the limit.

We tried multiple approaches to improve our benchmark performance. At first we tried changing the distribution pattern of the grid to one where ranks obtain multiple "full" rows. We did this, despite having an "optimal" implementation (of the access pattern) because we couldn't measure wall times resembling this.

Using this new blocking (or **slicing** that's how we called it) approach we would avoid many cache misses, otherwise introduced by copying into a send and receive buffer each iteration (which should occur quite often in the left and right buffers).

```
for (int i = 0; i < cols_per_rank; i++){
  send_up_temps[i] = A[calc_index(0, i, cols_per_rank)];
}
for (int i = 0; i < cols_per_rank; i++){
  send_down_temps[i] = A[calc_index(rows_per_rank-1, i, cols_per_rank)];
}
for (int i = 0; i < rows_per_rank; i++){
  send_left_temps[i] = A[calc_index(i, 0, cols_per_rank)];
}
for (int i = 0; i < rows_per_rank; i++){
  send_right_temps[i] = A[calc_index(i, cols_per_rank-1, cols_per_rank)];
}
```

⇓

```
for (int i = 0; i < cols_per_rank; i++){
  send_up_temps[i] = A[calc_index(0, i, cols_per_rank)];
}
for (int i = 0; i < cols_per_rank; i++){
  send_down_temps[i] = A[calc_index(rows_per_rank-1, i, cols_per_rank)];
}
```

This small change yielded almost no improvement, with all measurements landing in the margin of error.

Next we tried using a modified access (**modaccess**), where we wouldn't calculate the position of the array each time.

```
int calc_index(int i, int j, int N) {
  return ((i) * (N) + (j));
}
...
for (int i = 0; i < cols_per_rank; i++){
  send_up_temps[i] = A[calc_index(0, i, cols_per_rank)];
}
for (int i = 0; i < cols_per_rank; i++){
  send_down_temps[i] = A[calc_index(rows_per_rank-1, i, cols_per_rank)];
}
...
for (int i = 0; i < rows_per_rank; i++) {
  for (int j = 0; j < cols_per_rank; j++) {
    if(my_rank == rank_with_source && (i == (source_i % rows_per_rank)) && (j == (source_j % ↩
    cols_per_rank))){
      B[calc_index(i, j, cols_per_rank)] = A[calc_index(i, j, cols_per_rank)];
      continue;
    }

    // get temperature at current position
    value_t current_temp = A[calc_index(i, j, cols_per_rank)];

    // // get temperatures of adjacent cells
    value_t left_temp = (j != 0) ? A[calc_index(i, j-1, cols_per_rank)] : current_temp;
    value_t right_temp = (j != cols_per_rank-1) ? A[calc_index(i, j+1, cols_per_rank)] :  current_temp;
    value_t up_temp = (i != 0) ? A[calc_index(i-1, j, cols_per_rank)] : (up != MPI_PROC_NULL ? ↩
    recv_up_temps[j] : current_temp);
    value_t down_temp = (i != rows_per_rank-1) ? A[calc_index(i+1, j, cols_per_rank)] : (down != ↩
    MPI_PROC_NULL ? recv_down_temps[j] : current_temp);
```

```
27
28      B[calc_index(i, j, cols_per_rank)] = current_temp + 1/8.f * (left_temp + right_temp + down_temp + ↩
        up_temp + (-4 * current_temp));
29    }
30  }
31
```

⇓

```
1  // get temperature at current position
2  int current_pos = calc_index(i, j, cols_per_rank);
3  value_t current_temp = A[current_pos];
4
5  // // get temperatures of adjacent cells
6  value_t left_temp = (j != 0) ? A[current_pos-1] : current_temp;
7  value_t right_temp = (j != cols_per_rank-1) ? A[current_pos+1] :  current_temp;
8
9  value_t up_temp = (i != 0) ? A[current_pos-N] : (up != MPI_PROC_NULL ? recv_up_temps[j] : current_temp);
10 value_t down_temp = (i != rows_per_rank-1) ? A[current_pos+N] : (down != MPI_PROC_NULL ? recv_down_temps↩
       [j] : current_temp);
11
12 B[current_pos] = current_temp + 1/8.f * (left_temp + right_temp + down_temp + up_temp + (-4 * ↩
       current_temp));
```

This strategy of avoiding these small computations seemed to work a bit better with some improvement (up to 2x) in the small problem size, but diminishing improvements at medium ps or even worse (average) performance at the largest problem size. For the sequential program this minor code change helped to improve (speed as well as efficiency) about 7%.

Lastly, we tried avoiding the send buffers all together and rely on the matrices and the blocking send/recv calls (would've worked in the blocking implementation as well, just some reordering required).

```
1  for (int i = 0; i < cols_per_rank; i++){
2    send_up_temps[i] = A[calc_index(0, i, cols_per_rank)];
3  }
4  for (int i = 0; i < cols_per_rank; i++){
5    send_down_temps[i] = A[calc_index(rows_per_rank-1, i, cols_per_rank)];
6  }
7
8  // determine neighbors
9  int up    = (my_rank-ranks_per_row >= 0) ? my_rank-ranks_per_row : MPI_PROC_NULL;
10 int down  = my_rank+ranks_per_row < num_ranks ? my_rank+ranks_per_row : MPI_PROC_NULL;
11
12 // send and receive up and down ghost cells
13 if (submatrix_row_id % 2) {
14   MPI_Send(send_up_temps, cols_per_rank, MPI_FLOAT, up, MPI_TAG, MPI_COMM_WORLD);
15   MPI_Send(send_down_temps, cols_per_rank, MPI_FLOAT, down, MPI_TAG, MPI_COMM_WORLD);
16   MPI_Recv(recv_down_temps, cols_per_rank, MPI_FLOAT, down, MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
17   MPI_Recv(recv_up_temps, cols_per_rank, MPI_FLOAT, up, MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
18 }
19 else {
20   MPI_Recv(recv_down_temps, cols_per_rank, MPI_FLOAT, down, MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
21   MPI_Recv(recv_up_temps, cols_per_rank, MPI_FLOAT, up, MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
22   MPI_Send(send_up_temps, cols_per_rank, MPI_FLOAT, up, MPI_TAG, MPI_COMM_WORLD);
23   MPI_Send(send_down_temps, cols_per_rank, MPI_FLOAT, down, MPI_TAG, MPI_COMM_WORLD);
24 }
```

⇓

```
1  int up    = (my_rank-ranks_per_row >= 0) ? my_rank-ranks_per_row : MPI_PROC_NULL;
2  int down  = my_rank+ranks_per_row < num_ranks ? my_rank+ranks_per_row : MPI_PROC_NULL;
3
4  // send and receive up and down ghost cells
```

```
if (submatrix_row_id % 2) {
  MPI_Send(&A[0], cols_per_rank, MPI_FLOAT, up, MPI_TAG, MPI_COMM_WORLD);
  MPI_Send(&A[rows_per_rank-1], cols_per_rank, MPI_FLOAT, down, MPI_TAG, MPI_COMM_WORLD);
  MPI_Recv(recv_down_temps, cols_per_rank, MPI_FLOAT, down, MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(recv_up_temps, cols_per_rank, MPI_FLOAT, up, MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else {
  MPI_Recv(recv_down_temps, cols_per_rank, MPI_FLOAT, down, MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(recv_up_temps, cols_per_rank, MPI_FLOAT, up, MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(&A[0], cols_per_rank, MPI_FLOAT, up, MPI_TAG, MPI_COMM_WORLD);
  MPI_Send(&A[rows_per_rank-1], cols_per_rank, MPI_FLOAT, down, MPI_TAG, MPI_COMM_WORLD);
}
```

This again quite small change didn't improve the results drastically, as probably the optimizer of the gcc compiler were already doing the heavy lifting, thus softening the impact of our changes.

We tried to improve the sequential version as well using a fast convolution (correctly speaking cross-correlation). The algorithmic pattern from the 2D grid problem is 1:1 the cross-correlation applied to any 2D signal. Using this insight we wanted to implement a "simple" algorithm where we first transform our kernel, and then our grid, into frequency space, which would then only require applying an elementwise multiplication to be equivalent to a convolution. If FFT would be used for the transformation we could achieve a nice speedup, as the normal 2D convolution/cross-correlation is computed in $\mathcal{O}(n^2)$ and the FFT based one should be in the order of $\mathcal{O}(n \log n)$. A problem which neither of us were aware of, was that the FFT based convolution is a circular convolution thus not resembling the correct grid after computing. We attempted to rearrange the result such that the grid would be correct, but didn't get far. Another problem was that we did not know how the multiplication has to be done and what shape the kernel (9×9) and the matrix should be. Long answer short, the matrices should both be the same and be padded to the next power of 2 (e.g. $768 \rightarrow 1024$). We also found some interesting paper about this or similar implementations e.g. TurboStencil.

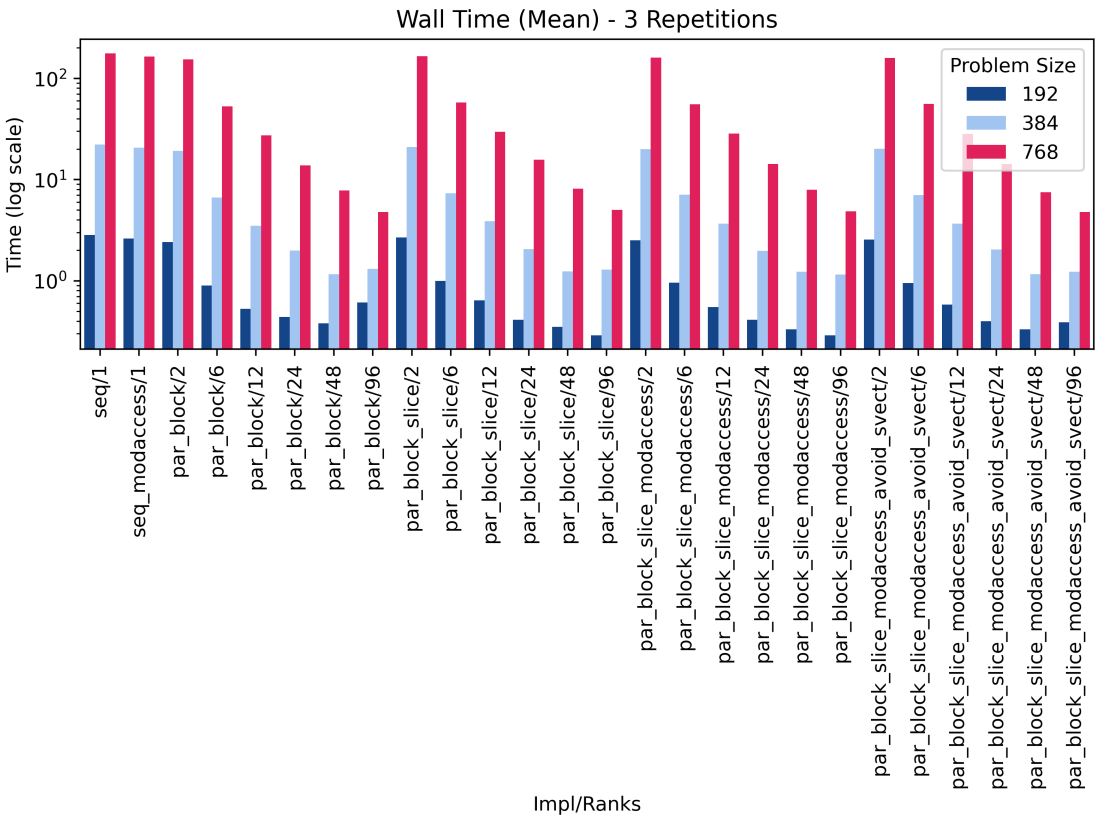

Figure 1: Wall time of all the different implementations.

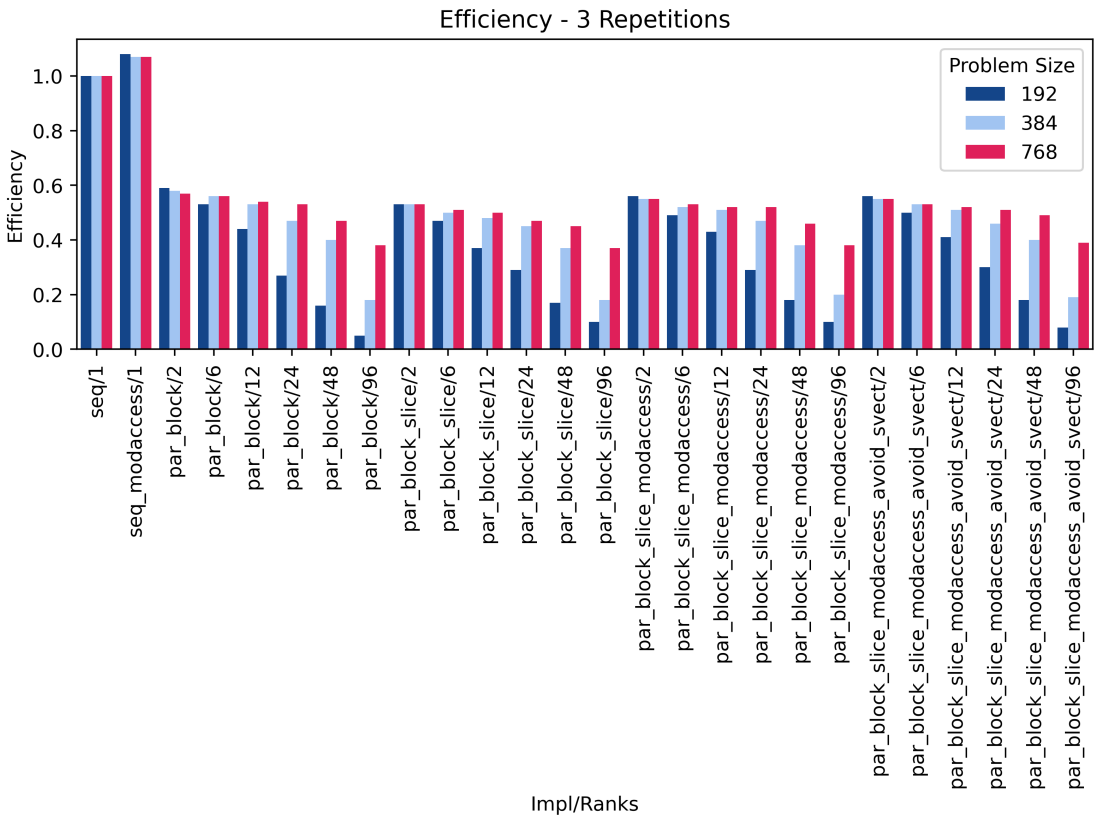

Figure 2: Efficiency of all the different implementations.

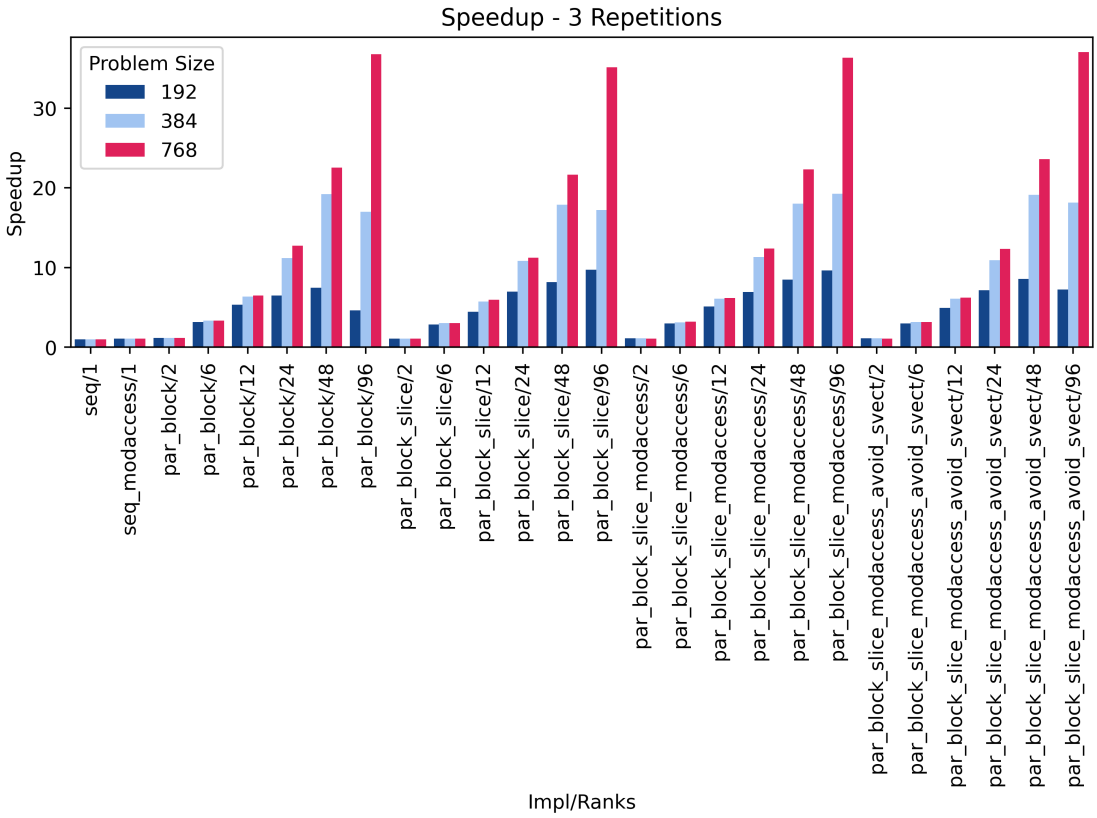

Figure 3: Speedup of all the different implementations.

Results of 2D Heat Stencil Execution												
Impl/Ranks	**Problem Size**											
	192				**384**				**768**			
	μ [s]	σ [s]	S(p)	E(p)	μ [s]	σ [s]	S(p)	E(p)	μ [s]	σ [s]	S(p)	E(p)
seq/1	2.84	0.08	1.00	1.00	22.23	0.17	1.00	1.00	176.17	0.13	1.00	1.00
seq_modaccess/1	2.62	0.05	1.08	1.08	20.69	0.18	1.07	1.07	164.38	0.19	1.07	1.07
par_block/2	2.41	0.01	1.18	0.59	19.26	0.08	1.15	0.58	153.31	0.27	1.15	0.57
par_block/6	0.90	0.01	3.17	0.53	6.64	0.00	3.35	0.56	52.88	0.25	3.33	0.56
par_block/12	0.53	0.01	5.32	0.44	3.49	0.01	6.36	0.53	27.25	0.56	6.47	0.54
par_block/24	0.44	0.01	6.50	0.27	1.99	0.08	11.19	0.47	13.80	0.12	12.76	0.53
par_block/48	0.38	0.00	7.46	0.16	1.16	0.02	19.20	0.40	7.82	0.33	22.54	0.47
par_block/96	0.61	0.23	4.64	0.05	1.31	0.02	17.00	0.18	4.79	0.19	36.79	0.38
par_block_slice/2	2.67	0.05	1.06	0.53	20.89	0.21	1.06	0.53	166.09	0.57	1.06	0.53
par_block_slice/6	1.00	0.01	2.84	0.47	7.34	0.00	3.03	0.50	57.83	0.25	3.05	0.51
par_block_slice/12	0.64	0.06	4.44	0.37	3.87	0.08	5.75	0.48	29.58	0.50	5.96	0.50
par_block_slice/24	0.41	0.01	6.98	0.29	2.05	0.02	10.85	0.45	15.68	0.66	11.24	0.47
par_block_slice/48	0.35	0.01	8.18	0.17	1.24	0.12	17.88	0.37	8.13	0.44	21.67	0.45
par_block_slice/96	0.29	0.00	9.73	0.10	1.29	0.11	17.24	0.18	5.02	0.38	35.12	0.37
par_block_slice_modaccess/2	2.52	0.02	1.13	0.56	20.05	0.13	1.11	0.55	159.75	0.62	1.10	0.55
par_block_slice_modaccess/6	0.96	0.00	2.97	0.49	7.07	0.07	3.14	0.52	55.30	0.02	3.19	0.53
par_block_slice_modaccess/12	0.55	0.02	5.12	0.43	3.66	0.01	6.07	0.51	28.46	0.45	6.19	0.52
par_block_slice_modaccess/24	0.41	0.03	6.92	0.29	1.97	0.01	11.30	0.47	14.20	0.03	12.40	0.52
par_block_slice_modaccess/48	0.33	0.01	8.50	0.18	1.23	0.18	18.02	0.38	7.90	0.41	22.31	0.46
par_block_slice_modaccess/96	0.29	0.01	9.63	0.10	1.15	0.03	19.26	0.20	4.85	0.46	36.31	0.38
par_block_slice_modaccess_avoid_svect/2	2.55	0.04	1.11	0.56	20.09	0.11	1.11	0.55	159.50	0.42	1.10	0.55
par_block_slice_modaccess_avoid_svect/6	0.95	0.00	2.98	0.50	7.02	0.01	3.17	0.53	55.72	0.39	3.16	0.53
par_block_slice_modaccess_avoid_svect/12	0.58	0.04	4.92	0.41	3.66	0.01	6.08	0.51	28.22	0.11	6.24	0.52
par_block_slice_modaccess_avoid_svect/24	0.40	0.01	7.17	0.30	2.03	0.11	10.93	0.46	14.31	0.23	12.32	0.51
par_block_slice_modaccess_avoid_svect/48	0.33	0.01	8.56	0.18	1.16	0.03	19.12	0.40	7.47	0.05	23.58	0.49
par_block_slice_modaccess_avoid_svect/96	0.39	0.16	7.26	0.08	1.23	0.13	18.13	0.19	4.76	0.22	37.03	0.39

- If you want to share your performance results of the improved version, insert the final walltime and speedup of the 2D stencil for 96 cores for N=768x768 and T=768x100 into the comparison spreadsheet: spreadsheet.