Parallel Architectures and Programming Models, 2024W

Assignment 1: OpenMP Tasking, Roofline Model

PROGRAMMING PART :: SHORT DESCRIPTION

Implement efficient task-parallel versions of the Global Pair-wise Sequence Alignment algorithm using:

1. OpenMP **taskloop** and related constructs only
2. OpenMP explicit **tasks** and related constructs only

Remarks for both variants:

- Use **explicit data scoping** for variables that you use in OpenMP directives
- Use only **tasking** and related constructs and clauses (parallel is naturally ok)
  - No worksharing constructs such as "for" and "section/sections" but **`single`** and/or **`master`** are fine
- Try at least 3 different task granularity settings (amount of work per task)
  - E.g, a fine-grained task would process only 1 element, while the coarser grained task would process more elements sequentialy
- Achieve the required speedup on Alma (see "assignment description" slides)
- Write a report describing the details of your work

Hints: You can use constructs like task, and **taskloop**, and all clauses, such as those for data scoping, **reduction**, **in_reduction**, **depend**, if, **grainsize**, **num_tasks**, nowait, and similar

GLOBAL PAIR-WISE SEQUENCE ALIGNMENT (GPSA)

Problem statement

- Given a pair of sequences, calculate the best possible alignment (the one that gives us the highest score)
- The optimal path must stretch from beginning to end in both sequences (global alignment)
- i.e., maximize the number matches and minimize the number of mismatches and gaps

Example with (two short sequences)

a. `Sequence X: A T A`
b. `Sequence Y: A G T T A`

```
A-T-A    A--TA
AGTTA    AGTTA
```

Best similarity scores

These two sequences can have different alignments

- Substitution Matrix
  - Score/penalty for matches/mismatches
  - Penalty for gaps (we use the linear gap penalty)
- We will use a simple scoring scheme for this assignment

GPSA :: DYNAMIC PROGRAMMING

Step 1: Preparation (already implemented)

- **Allocate** a matrix with (let's call this matrix **S**)

  X.length()+1 rows and
  Y.length()+1 columns

- **Create a simple scoring scheme**
  - A way of scoring two characters in two sequences
  - For simplicity we use a simple scoring scheme in this assignment
    - `match=1, mismatch=-1, gap = -2`

* Note that typically a block substitution matrix is used, see https://www.ncbi.nlm.nih.gov/IEB/ToolBox/C_DOC/lxr/source/data/BLOSUM62

GPSA :: DYNAMIC PROGRAMMING

Step 2: Calculating Similarity Matrix (S) (to parallelize)

- **Initialize** the first row and column with initial values
  i.e., gap penalties (boundary conditions)
- We apply the scoring scheme
  - match=1, mismatch=-1, **gap = -2**

```
for (int i = 1; i < rows; i++)
   S[i][0] = i * gap_penalty;

for (int j = 0; j < cols; j++)
   S[0][j] = j * gap_penalty;
```

This is the result

		A	G	T	T	A
	0	-2	-4	-6	-8	-10
A	-2					
T	-4					
A	-6					

Similarity matrix 6x6
5 letter long sequences

This substitution matrix also corresponds to a simple scheme:
match=1, **mismatch**=-1, **gap** = -2

GPSA :: DYNAMIC PROGRAMMING

Step 2: Calculating Similarity Matrix (S) (to parallelize)

- **Scan** the matrix in row-major order, and apply the following:
- **X** and **Y** are the sequences

$$S_{(i,j)} = max \begin{cases} S_{(i-1,j-1)} + \begin{cases} match\_score \; if \; X_{i-1} = Y_{j-1} \\ mismatch\_score \; if \; X_{i-1} \neq Y_{j-1} \end{cases} \\ S_{(i-1,j)} + gap\_penalty \\ S_{(i,j-1)} + gap\_penalty \end{cases}$$

- We use the following kernel to compute it:

```
float match = S[i-1][j-1]
              + (X[i-1]==Y[j-1] ? match_score : mismatch_score);
float del = S[i-1][j] + gap_penalty;
float insert = S[i][j-1] + gap_penalty;
S[i][j] = std::max( {match, del, insert} );
```

this is our computation **kernel**

		A	G	T	T	A
	0	-2	-4	-6	-8	-10
A	-2	**1**				
T	-4					
A	-6					

Similarity matrix 6x4
5 and 3 letter long sequences

GPSA :: DYNAMIC PROGRAMMING

Step 2: Calculating Similarity Matrix (S) (to parallelize)

- **Scan** the matrix in row-major order, and apply the following:

```
float match = S[i-1][j-1]
              + (X[i-1]==Y[j-1] ? match_score : mismatch_score);
float del = S[i-1][j] + gap_penalty;
float insert = S[i][j-1] + gap_penalty;
S[i][j] = std::max( {match, del, insert} );
```

- [i==1 and j == 1]:

```
    match = S[0][0] + match;    // 1, S[0][0] is 0 and AA is a match (1)
    delete = -2 + gap_penalty; // -4, S[0][1] is -2 and gap_penalty is -2
    insert = 0 + match;         // -4, since gap_penalty is -2
    S[i][j] = max(1, -4, -4);   // 1
```

		A	G	T	T	A
	0	-2	-4	-6	-8	-10
A	-2	**1**				
T	-4					
A	-6					

Similarity matrix 6x4
5 and 3 letter long sequences

GPSA :: DYNAMIC PROGRAMMING

Step 2: Calculating Similarity Matrix (S) (to parallelize)

- **Scan** the matrix in row-major order, and apply the following:

```
float match = S[i-1][j-1]
              + (X[i-1]==Y[j-1] ? match_score : mismatch_score);
float del = S[i-1][j] + gap_penalty;
float insert = S[i][j-1] + gap_penalty;
S[i][j] = std::max( {match, del, insert} );
```

- [i==1 and j == 1]:

```
match = S[0][0] + match;    // 1, S[0][0] is 0 and AA is a match (1)
delete = -2 + gap_penalty;  // -4, S[0][1] is -2 and gap_penalty is -2
insert = 0 + match;         // -4, since gap_penalty is -2
S[i][j] = max(1, -4, -4);   // 1
```

- [i==1 and j == 2]:

```
match = -2 + mismatch_score // -3, S[0][1] is -2 and AG is a mismatch (-1)
delete = 1 + gap_penalty    // -1, since the left neighbour is 1
insert = -2 + gap_penalty   // -6, since the neighbour above is -4
S[i][j] = max(-3, -1, -6)   // -1
```

		A	G	T	T	A
	0	-2	-4	-6	-8	-10
A	-2	**1**	**-1**			
T	-4					
A	-6					

Similarity matrix 6x4
5 and 3 letter long sequences

You can see how algorithm works here:
https://bioboot.github.io/bimm143_W20/class-material/nw/

GPSA :: DYNAMIC PROGRAMMING

Step 2: Calculating Similarity Matrix (S) (to parallelize)

- Scan the matrix in row-major order, and apply the following:

```
float match = S[i-1][j-1]
              + (X[i-1]==Y[j-1] ? match_score : mismatch_score);
float del = S[i-1][j] + gap_penalty;
float insert = S[i][j-1] + gap_penalty;
S[i][j] = std::max( {match, del, insert} );
```

- **Iterate** until everything has been calculated
- Note that floats values only for the assignment-specific purposes

→ Next, we also need to traceback to align the sequences, and get some statistics (already implemented)

		A	G	T	T	A
	0	-2	-4	-6	-8	-10
A	-2	**1**	**-1**	**-3**	**-5**	**-7**
T	-4	**-1**	**0**	**0**	**-2**	**-4**
A	-6	**-3**	**-2**	**-1**	**-1**	**-1**

Similarity matrix 6x4
5 and 3 letter long sequences

GPSA :: DYNAMIC PROGRAMMING

Step 3: Traceback (already implemented)

- Process of deduction of the best alignment
- Begins with the last cell
- We iterate in reverse and look for maximal scores

		A	G	T	T	A
	0	-2	-4	-6	-8	-10
A	-2	**1**	**-1**	**-3**	**-5**	**-7**
T	-4	**-1**	**0**	**0**	**-2**	**-4**
A	-6	**-3**	**-2**	**-1**	**-1**	**-1**

GPSA :: DYNAMIC PROGRAMMING :: SUMMARY

Step 1: Preparation

- Allocation is already implemented

Step 2: Calculating Similarity Matrix (S)

- Only sequential version implemented
- You need to develop 2 different versions of this step with OpenMP tasking that achieve the required speedup on ALMA
- You can start from the sequential version

Step 3: Traceback to aligning the sequences

- Already Implemented

		A	G	T	T	A
	0	-2	-4	-6	-8	-10
A	-2	**1**	**-1**	**-3**	**-5**	**-7**
T	-4	**-1**	**0**	**0**	**-2**	**-4**
A	-6	**-3**	**-2**	**-1**	**-1**	**-1**

GPSA :: SERIAL IMPLEMENTATION

`SequenceInfo::gpsa_sequential(...);`

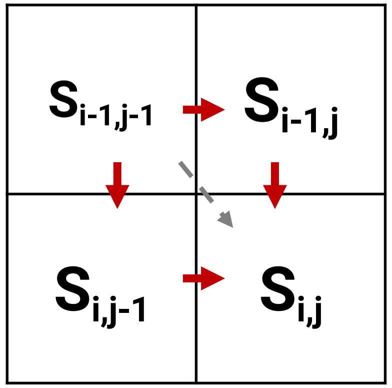
- Class member function
- Called automatically for you from the **main**
- Nested for-loops scans matrix S in row-major order and assigns a value to matrix entry `S[i][j]` per iteration
- Once the algorithm halts, it returns the number of "**`visited`**" cells for verification purposes
- Uses float values only for this purposes of this exercise
- This is you starting point, **gpsa_tasks**() and **gpsa_taskloop**() member function will have the same code at start
- Goal is to add OpenMP directives, and transform some loops so that the and the way program iterates over the S matrix so that the overall execution time is reduced

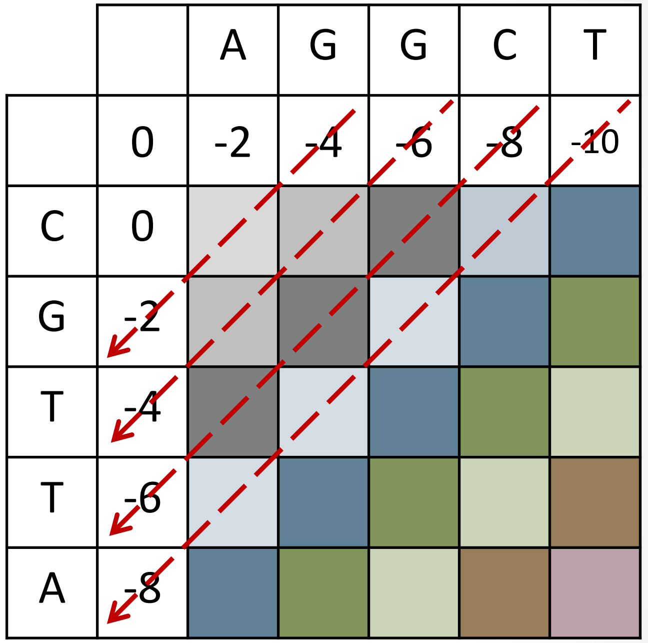
```
unsigned long SequenceInfo::gpsa_sequential(float** S) {
   unsigned long visited = 0;
   gap_penalty = SUB[0][cmap['*']]; // min score

   // Boundary
   for (unsigned int i = 1; i < rows; i++) {
      S[i][0] = i * gap_penalty;
      visited++;
   }

   for (unsigned int j = 0; j < cols; j++) {
      S[0][j] = j * gap_penalty;
      visited++;
   }

   for (unsigned int i = 1; i < rows; i++) {
     for (unsigned int j = 1; j < cols; j++) {
       float match = S[i - 1][j - 1]
                    +(X[i-1]==Y[j-1] ? match_score : mismatch_score);
       float del = S[i - 1][j] + gap_penalty;
       float insert = S[i][j - 1] + gap_penalty;
       S[i][j] = std::max({match, del, insert});

       visited++;
     }
   }

   return visited;
}
```

GPSA :: TASK-PARALLEL IMPLEMENTATION

Data dependences

- Maybe we can iterate the matrix differently?
- Example: **Wavefront (Antidiagonal)**





Iterating over antidiagonals

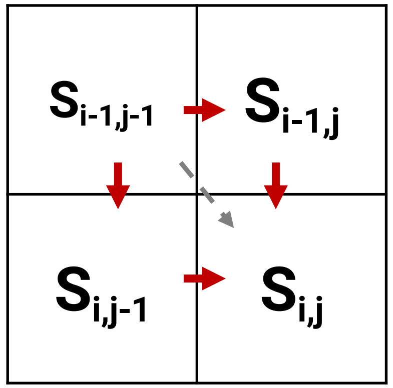
Tasks

- Still some tasks must wait for previous tasks
  - You must avoid data races!
- **Good task granularity?**
  - How much work should each task do?
  - 1 Element per task or more elements per task (blocks/group of elements)?
  - Should all tasks be of the same granularity?

GPSA :: TASK-PARALLEL IMPLEMENTATION

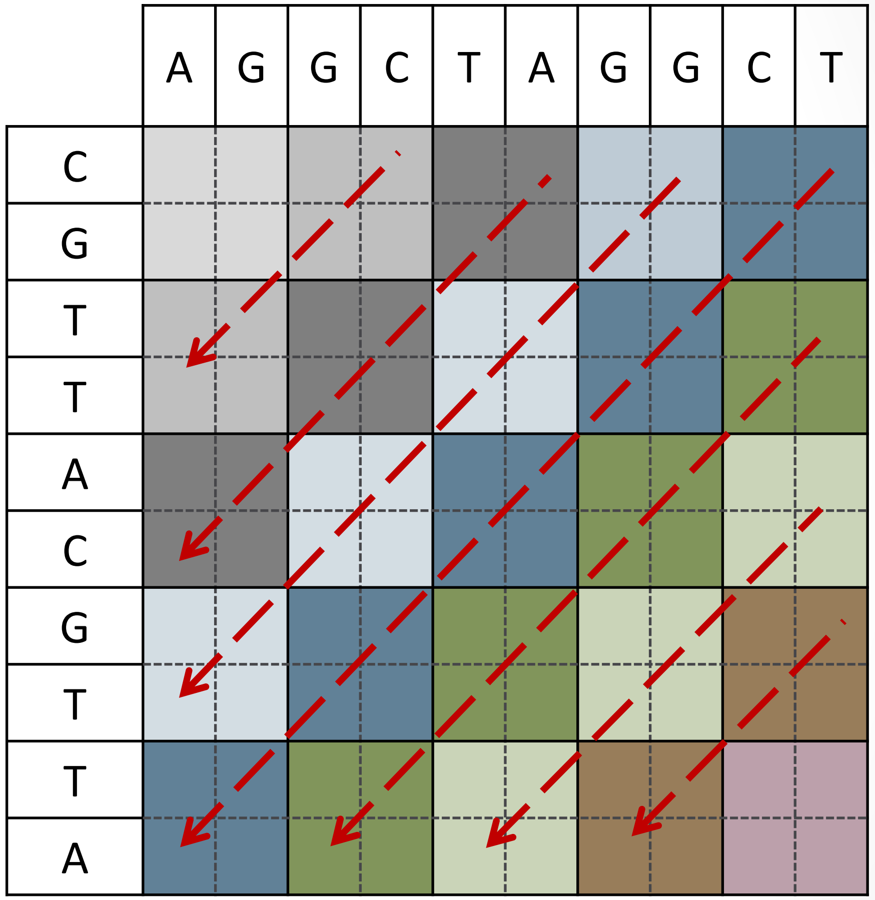
Data dependences

- Maybe we can iterate the matrix differently?
- Example: **Wavefront (Antidiagonal)**



Tasks

- Still some tasks must wait for previous tasks
  - You must avoid data races!
- **Good task granularity?**
  - How much work should each task do?
  - 1 Element per task or more elements per task (blocks/group of elements)?
  - Should all tasks be of the same granularity?

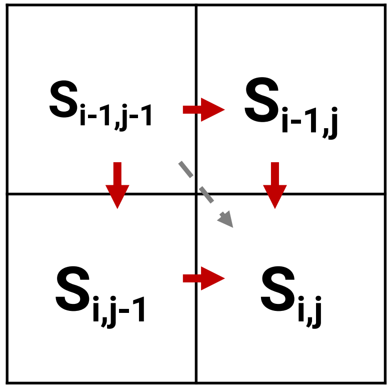


Iterating over antidiagonals

GPSA :: TASK-PARALLEL IMPLEMENTATION

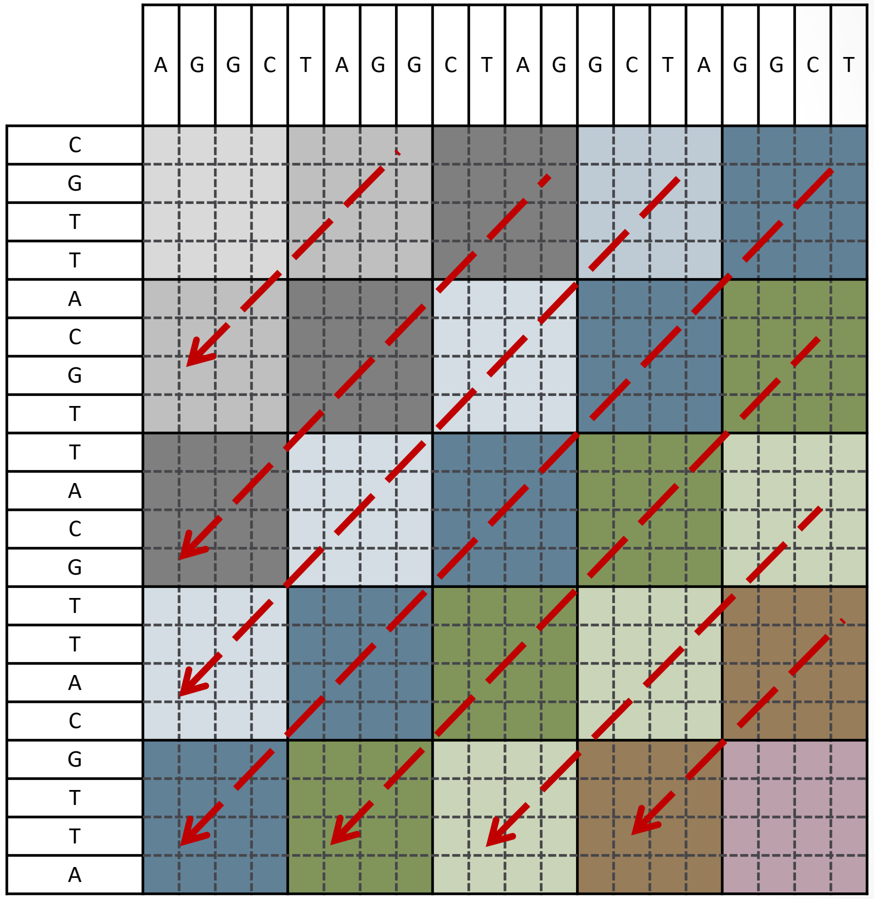
Data dependences

- Maybe we can iterate the matrix differently?
- Example: **Wavefront (Antidiagonal)**



Tasks

- Still some tasks must wait for previous tasks
  - You must avoid data races!
- **Good task granularity?**
  - How much work should each task do?
  - 1 Element per task or more elements per task (blocks/group of elements)?
  - Should all tasks be of the same granularity?



A G G C T A G G C T A G G C T A G G C T
C G T T A C G T T A C G T T A C G T T A

Iterating over antidiagonals

GPSA :: TASK-PARALLEL IMPLEMENTATION :: HINTS

Transforming the for-loops

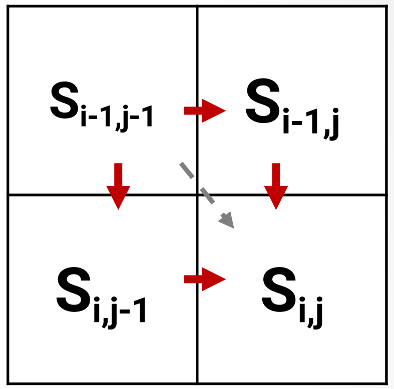
- You want iterate over anti-diagonals using elements or blocks

Dealing with granularity

- For performance you probably want to introduce blocking
  - Task granularity? Cache utilization? Other purposes?

Hints:

- **Use the wavefront (antidiagonal)**
- You can leave the kernel as it is and only adjust how you iterate the matrix
- The outermost for-loop could go over the anti-diagonals and
  the next (nested) loop level can then set up the indices for each anti-diagonal
  - If you want blocks, you probably need a 4 for-loops
    - The outermost loop would have fewer anti-diagonals in this case
    - The next (nested) loop can then set up the indices for each anti-diagonal
      but also needs to account for block sizes and set up bounds for the inner-most two loops that call the kernel
    - You can leave ***i*** and ***j*** in the inner-most loops to simplify things
- For explicit tasks you may want to look at the depend clause
- Always check for data races!


$$\begin{tikzcd} S_{i-1,j-1} \arrow[r] \arrow[d] \arrow[dr, dashed] & S_{i-1,j} \arrow[d] \\ S_{i,j-1} \arrow[r] & S_{i,j} \end{tikzcd}$$

ASSIGNMENT DESCRIPTION (1/3)

Implement efficient task-parallel implementations of the given algorithm using (1) OpenMP explict tasks and (2) taskloops.

- Your functions need to be added to "`implementation.hpp`" and have the following signatures:
  - `unsigned long SequenceInfo::gpsa_taskloop(float** S, int grain_size=1);`
  - `unsigned long SequenceInfo::gpsa_tasks(float** S, int grain_size=1);`
  - There are already templates for the two functions above, you can focus on writing the code inside these two functions
  - Granularity modifier is an argument that you can specify on terminal, and you can use to parametrize granularity if you need it – this is not meant to be passed as the **`grain_size`** in OpenMP (although you can also do that)
  - Remarks:
    - Functions for additional versions may also be provided, e.g. gpsa_tasks_v2, gpsa_taskloop_v2, etc. For that, you may need to adjust testing as well. Alternatively, you can just make another source file. However, these are not necessary.
    - Typically, there is **no need to change other files** and the rest of the code is purposefully implemented as it is for this specific use case and for this particular input data, and it should not be changed.
    - Data structures are passed as pointers to allow full compatibility with OpenMP Features
- The value of the variable **"`visited`"** must be incremented at each iteration of the nested for-loop. This value has to be equal to the value produced by the sequential version.
- Use explicit data-scoping for variables declared outside of the task generating constructs

ASSIGNMENT DESCRIPTION (2/3)

Implementation.hpp

- This is the only file that you need to edit
- Analyze loops and see if they can be parallelized
  - There are three, and the last one can be tricky
- Analyze all variable before and after OpenMP **parallel regions**, and check if anything needs synchronized accesses
- Check if loop 3 can be somehow transformed with respect to the slides 13-16
- Adjust granularity with OpenMP clauses and with loop transformations
- Which variables can be made **private** or **firstprivate**, and which need to be **shared**?
- Make sure that you generate tasks from a single thread
- Taskloop and explicit task will differ both in complexity and possibly in performance
  You can ignore most of the other code
- Check last 10 slides of OpenMP Tasking lecture for additional ideas

Are these safe?

```
unsigned long SequenceInfo::gpsa_taskloop(float** S)
{
    unsigned long visited = 0;
    gap_penalty = SUB[0][cmap['*']]; // min score

    for (unsigned int i = 1; i < rows; i++) {
        S[i][0] = i * gap_penalty;
1       visited++;
    }

    for (unsigned int j = 0; j < cols; j++) {
2       S[0][j] = j * gap_penalty;
        visited++;
    }

    for (unsigned int i = 1; i < rows; i++) {
        for (unsigned int j = 1; j < cols; j++) {
            float match = S[i-1][j-1]
            + SUB[cmap.at(X[i-1])][cmap.at(Y[j-1])];
3           float del = S[i - 1][j] + gap_penalty;
            float insert = S[i][j - 1] + gap_penalty;
            S[i][j] = std::max({match, del, insert});

            visited++;
        }
    }

    return visited;
}
```

ASSIGNMENT DESCRIPTION (3/3)

Measure performance with at least 3 different task granularities per implementation version

- For example, *a fine-grained* task would process only 1 element, while the coarser grained task would process more elements sequentialy – are there values that work better?
- The tree values should show at which grain size does the performance starts to drop

Test and measure on Alma

- Test early to avoid peak usage times
- Performance numbers (speedup) for your report is only relevant for the assignment if measured on Alma
- You need to be connect to university network to access it (or use VPN), and then ssh <your_username>@alma.par.univie.ac.at (https://moodle.univie.ac.at/mod/forum/discuss.php?d=3695157)

Your best effort solution should aim to achieve a speedup of 12+ on Alma for both versions

- Better versions go above this speedup with the given dataset(s)
- It not required that both version achieve the 12+ speedup, but the slower one should be around 9

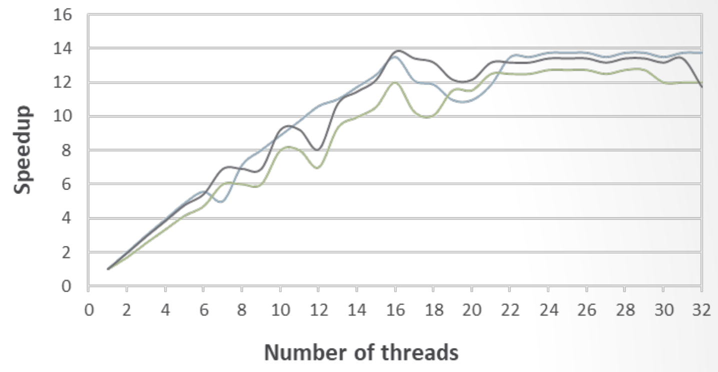
The total number of threads in use should work via OMP_NUM_THREADS environment variables

- There is a runall.sh script that gives you an idea how the code is executed when checking your submission

REPORT

Write a report addressing the following:

- A brief description of your approach (**tasks** vs **taskloop**)
- Granularity and the total number of tasks
  - How many tasks are generated in each loop
    - No need for a concrete number, just a desciption
  - Does this number change or is it statically determined?
  - How many items per task is processed?
    - If this value changes dynamically, describe how
- Dependences
  - How are task dependences handled so that the produced result is correct?
  - Do (or could) the depend clauses be beneficial for the explicit tasks version?
- How data scoping was used and what did you need to declare and why?
- Speedup graph(s) and measurements with respect to the sequential version (~12s on Alma - default dataset)
  - Different variants (and different task granularities) using **1 – 32** threads
- Bottlenecks, observations, and how can your code be improved
  - Do V1 and V2 exhibit different performance? If that is the case, try to explain why!
  - What helped you code run faster and why?



An example how a speedup graph should look like*

*Your graph can look differently, but it needs to have Speedup on the vertical axis and all labels on the graph

SUBMISSION

Put your files in a zip file and and upload to Moodle

~aMatrNr/(your files)

- Source files + report-a1.pdf

Upload all to Assignment 1 on Moodle before the deadline.

Notes:

- Both code and documentation are required!
  i.e., your code without the report will not be graded positive!

GETTING STARTED

Serial version in Moodle

`main.cpp`	(main source file)
`helpers.cpp`	(helper classes and functions, e.g., loading a sequence from file, traceback, etc..)
`implementation.hpp`	(your work goes in the two tasking related functions in this file)
`X.txt, Y.txt`	(input data - sequences)
`X2.txt, Y2.txt`	(input data – sequences with equal lengths)

To compile the code on Alma you need to:

```
/opt/global/gcc-11.2.0/bin/g++ -O2 -std=c++20 -fopenmp -o gpsa  main.cpp
```

This version is necessary on Alma, if the compiler is older, it may compile, but some of the OpenMP constructs and clauses may be ignored. For local development, you need to check your GCC version and modify/remove this switch.

To run with all available threads:

```
srun --nodes=1 ./gpsa
```

- To run with 16 threads instead of default 32: **`OMP_NUM_THREADS=16 srun --nodes=1 ./gpsa`**

Alma system: http://www.par.univie.ac.at/teach/doc/alma.html

GETTING STARTED

Usage ./gspa

--x	provide an input file for the first sequence
--y	provide an input file for the second sequence
--exec-mode	run different versions of the program: 0 – all versions, 1 – sequential version only 2- taskloop version only, 3 – explicit tasks version only
--grain-size	an optional parameter passed to functions, which you optionally use to easier testing of task granularity
--save-to	an optional parameter to specify output filename for the sequential alignment

There is also a “runall.sh” script that you can run to executes configurations on 1 to 32 threads. You can use it after you compile your code, as you need to pass the executable as the first argument, e.g., “`sh runall.sh ./gpsa`”

GETTING STARTED

Sequences

X.txt, Y.txt, size: [51480x53640]

- Random, big sequences.

X2.txt, Y2.txt, size: [32768x32768]

- It may be easier to start with, since the Similarity matrix sizes divide well with 16, 32, 64, 128, 256, and both sequences have the same size, which means that it transform the for-loops if needed.

simple1.txt, simple2.txt, size: [3x5]

- Small sequences that you can use for debugging (matching the slides)
- You can change this file as you please.

```
A-T-A
AGTTA
```

longer-simple.txt, longer-simple.txt, size: [20x20]

- Small sequences that you can use for debugging (matching the slides)
- You can change this file as you please.

```
GATTACAG--A--TTACAGATTAC
AGTTA-AGTTAAGTTA-AG-TTA-
```

To switch to a different sequence, you can pass the command line names, for example:

```
./gpsa --x X2.txt --y Y2.txt
```