Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences
Fachbereich Informatik
Lehrgebiet Programmierung komplexer Systeme
Prof. Dr. Rudolf Berrendorf

Parallel Systems
WS 19/20
given out : 30.09.2019
to deliver : 20.10.2019

Assignment 1 : Parallel Programming

Remarks for all assignments:

- Check regulations for assignments! You find these regulations on the web page for this course. Please read them carefully!
- Unless otherwise stated, all OpenMP programs must run on **`wr43`** (batch queue **`wr43`**) started inside a batch job. You may use **`wr0`** for interactive small non-MPI program development tests.
- All MPI programs must run on queue **`hpc3`** using a batch script that is submitted on **`wr0`** to that queue. You may use **`wr14`** for interactive **small** MPI tests (small means small parallelism and small amount of data). From **`wr0`** you can ssh to **`wr14`**.
- Assignments must be delivered up to the given deadline in the directory **`/home/parsys/youraccountname`** on the computer **`wr0.wr.inf.h-brs.de`** where **`youraccountname`** is your individual account name as explained in the rules for assignments. A cron job runs at 0:01 on the day following the deadline gathers all assignments. I.e., the deadline is always hard.

Assignment 1 (10 points) :
For a given vector $(x_1,...,x_n) \in K^n$ for a field K and a binary associative operator $\otimes : K \times K \to K$ the result of a prefix operation is defined as $(x_1, x_1 \otimes x_2, x_1 \otimes x_2 \otimes x_3, ..., x_1 \otimes ... \otimes x_n) \in K^n$. Example: for a vector $(1,2,3) \in \mathbb{N}^3$ and $+$ as operator a prefix operation computes the result vector $(1,3,6) \in \mathbb{N}^3$.

On the web page for this course you find a folder **`prefix`** which contains a sequential implementation for such a prefix operation. The README in this folder explains some details of files and the implementation. To run this program just call **`make run`** in the command line on **`wr0`**. After the program stops you see the time the fastest out of three (same) prefix operations took for a large vector.

Your task is to investigate what changes improve this run time shown on the display. You have three possibilities to do that:

1) compiler: we have three different compilers on our systems (see below).
2) compiler options (see below)
3) source code changes: you are allowed to modify whatever you want in the source code, but only inside **`prefix.c`**. Don't touch the rest and leave the code sequential.

We have three different C compilers on our systems:

- GNU C compiler **`gcc`**. Documentation is available through **`man gcc`**. Use **`module load gcc`** once a session.
- Intel compiler **`icc`**. Documentation is available through **`man icc`**. Use **`module load intel-compiler`** once a session.
- PGI compiler **`pgcc`**. Documentation is available through **`man pgcc`**. Use **`module load pgi`** once a session.

Check in the documentation what options may influence the run time of your program. Try some of them and document the influence of a compiler option on the runtime of the program. Just add the compiler option you want to check to the appropriate CFLAGS line in the Makefile (see README) and re-compile everything and run the program.

Submit a protocol that shows what you did (relative to the unmodified base case) and how this influenced the runtime of the program.

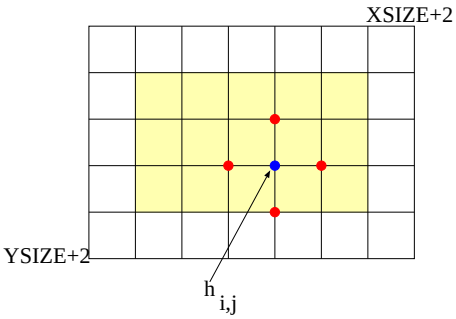
You can run all of your tests interactively on **wr0**.

Assignment 2 (10 points) :
On the web page for this course you find for this assignment a sequential program (folder **heat**) that simulates (in a simple form) the heat distribution in a twodimensional area. The area that is simulated is discretized in an grid of size $XSIZE \times YSIZE$. We assume that the temperature on each of the inner points of that area is influenced by the temperature of the four neighboring points.

If $h_{i,j}$ is the temperature at the discretization point i, j, we compute a new temperature value $h_{i,j}^{new}$ at i, j according to the following formula:

$$h_{i,j}^{new} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4} \tag{1}$$

To simplify the program, the grid in the program has two additional points in each dimension i.e. the array is dimensioned as $(XSIZE+2) \times (YSIZE+2)$).



The structure of the program is as follows:

```
Initialization
Iterate
   display()   // only after niter iterations
   update()
   heat_source()
```

The functions in the program work as follows:

- display(): display the current heat distribution (called every *niter* iterations)
- update(): updates the grid according to the above formula
- heat_source(): in our example simulation we have at 4 points in the simulation area permanent heat sources (i.e. $h_{i,j} = const$ independent of neighboring values).

The program can be compiled with **make** and the compiled program can be run interactively with **make run** and using the batch system with **sbatch job_heat.sh**.

Parallelize the program for OpenMP. Take care that the graphic display function to set a pixel (inside **display()**) can not be executed in parallel. Think about what might be a useful solution to that problem. Alternatives?

Run the sequential version of the program with the bath job. Run your parallel program and vary the number of OpenMP threads from 1-64 in powers of 2. Determine speedup numbers that you submit in your solution. To get meaningful results:

- Use the batch system and request always the node **wr43** (queue **wr43**) with all (96) cores (see job script).
- Make each measurement 3 times and take the best of the 3 times. Do you see any variance in the timings? If yes, have you any explanations for that?

Assignment 3 (10 points) :
Develop an MPI program that exchanges an array $a[n]$ of n numbers in a ring of n MPI processes. At startup of the program, process 0 sends a vector of size n that is initialized with $a[0] = 1$ and $a[i] = 0$ for all $i \neq 0$ to MPI process 1. Process 1 receives that vector, assigns $a[1] = 2$, and sends the modified vector to process 2. Process 2 receives that vector, sets $a[2] = 3$ and sends the modified vector to process 3 and so on. After MPI process $n-2$ sends the vector to $n-1$, this process $n-1$ modifies its value $a[n-1] = n$ and sends the final vector back to the original sender, MPI process 0. On process 0, check that values in the received vector are correct, i.e. $a[i] = i+1$ for all $i = 0, ..., n-1$. The program should use always all available processors assigned to a program run.

You may use the folder `mpi` as a starting point with Makefile and job script. See the README in the folder. Test your program with different values of n. Therefore you have to copy/change in the job script in `mpirun -np ...` the number of assigned processors for each program run. The number of tasks specified in the resource specification of the job script must always be at least as large as the number of started MPI processes in mpirun (which corresponds to n).