Second individual report

-, SUMUKHA SHRIDHAR, S221654

1 DATABAR EXERCISE 4

1.1 OpenMP and exercise

OpenMP is a shared memory model for parallel computing. OpenMP primarily works on the concept of fork/join model, where a master thread initialise a number of threads, and in the end all these threads are given back to the runtime when the parallel work is completed. In this model we can communicate with the processes in different nodes. So, OpenMP enbles on-node parallelism. In this model for loops are the main work horses. For the program to compile, we need to compile with the -fopenmp flag. Often we run into race conditions while using OpenMP, which needs to be taken into careful consideration. Possible solutions are locks and atomic statements or mutexes. OpenMP also facilitates different types of data sharing options like shared, private, firstprivate, lastprivate.

In order to parallelise calc.c, the main task is to parallelise the for loops. The stencil computations can be easily parallelised, which already gives us a significant speedup. We see that speedup slightly flatten as we approach 16 threads; this is due to the the overhead of thread spawning.

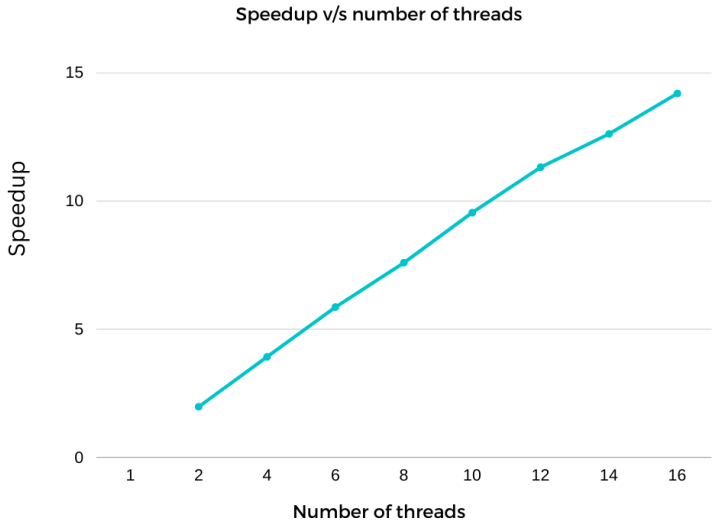

Fig. 1. Speedup v/s Numner of threads, OpenMP implementation

While the current program is statically scheduled, experiments with guided and dynamic scheduling resulted in a considerably higher runtime. omp_get_wtime was used to measure the execution time of the main parts of the program.

Author's address: -, Sumukha Shridhar, s221654, s221654@dtu.dk.

2 DATABAR EXERCISE 5

2.1 Submitting MPI jobs

The run.sh script is used to specify all the parameters like memory requirement, number of processors needed, maximum memory limit, job name, log files, executable, output directory etc. mpi.log contains the output of the code. So, the execution does not print anything on the terminal.

2.2 MPI, Distributed calculations

MPI is a distributed model for parallel computing. In this model we can communicate with the processes situated at different nodes. Each process in the program can do the computation as well as, send the data to the other processes by passing them, thus 'Message Passing'. Simple communications are operations are, for example MPI_Send and MPI_Recv. These are point-point protocols, i.e information sharing between two processes. In collective communications it is possible for a process to share its information to a number of processes. Message passing is essential for the stencil computations, like the ones in calc.c. Each process has a copy of the data and it communicates the boundary data with the the other processes, if needed. In the end there is a collective call MPI_Gather, where the root process collects the private copies of data from all the process in the communicator.

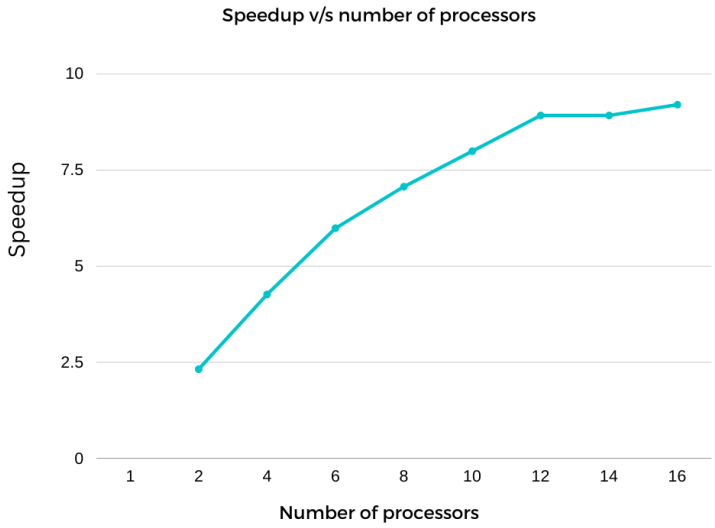

Fig. 2. Speedup v/s Numner of processors, MPI implementation

In order to parallelise calc.c, the main task is the domain decomposition. Each process has an array *for* from and *to*. Inside the while loop the boundary values from each process is communicated with the immediate neighbours. When the while loop exits, a we gather the *from* from each process into a global array. Initially we see a better speedup than the OpenMP, but as we increase the number of processes, there is more communication which is more expensive than the computation. So, the curve flattens as we increase the number of processors. Time was averaged over 3 observations, in order to have statistically sound results.

3 DATABAR EXERCISE 6

DeviceInfo.c gives the specifications of the GPU, memory, compute units, name of the vendor etc.
vadd_c.c in Exercise05 compiles and it gives a speedup of 2.52 when run on GPU. The steps followed are:

- we initialise the data on the CPU
- allocate memory on GPU
- copy data from CPU to GPU
- call the kernel where the computations are done
- copy the data back to CPU
- free the memory on GPU