UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

Computer Architecture

OpenMP Assignment
Vector Sorting

Serafin.Benito@uclm.es

UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

Contenido

- Program `OrdenaVector.c`
- Sorting methods: general description and application example
- Task 1: measure sequential algorithms times
- Task 2: OpenMP parallelization
- Optional Task
- Task 3: improved parallel version
- Assignment upload
- Evaluation

Program `OrdenaVector.c`

- Sorts a vector using different sequential algorithms: `ord_secA`, `ord_secB`, `ord_secC` y `ord_secD`
- Blocks of the main program:
  1. Vector population with random data (in the range [0,M[)
  2. Print unsorted vector
  3. For each algorithm: sort vector, check order is correct and print running time

     More details in the source code itself
  4. If the vector not large, print it sorted by the last method

Sorting algorithms: general description and application example

Algorithm `ord_secA`

- Take consecutive pairs of elements and sort them
- Build a sequence of 4 ordered elements out of every 2 consecutive ordered pairs
- Build a sequence of 8 ordered elements out of every 4 consecutive ordered pairs
- And repeat the process until all vector sorted

Application example
`ord_secA`

Initial vector:	44	54	13	42	95	19	32
incr = 2 →	44	54,	13	42,	19	95,	32
incr = 4 →	13	42	44	54,	19	32	95
incr = 8 →	**13**	**19**	**32**	**42**	**44**	**54**	**95**

Algorithm `ord_secB`

- The sequence of elements
  `vector[0], …, vector[i-1]`
  `is` sorted (`i` initialized to 1)
- At each iteration, `vector[i]` is placed in the right position inside the sorted part of the vector (left part in green in the example)
- When `i` reaches the last value and `vector[i]` is placed, the vector is sorted

Application example
`ord_secB`

Initial vector:	44	54	13	42	95	19	32
i = 1 →	44	54	13	42	95	19	32
i = 2 →	13	44	54	42	95	19	32
i = 3 →	13	42	44	54	95	19	32
i = 4 →	13	42	44	54	95	19	32
i = 5 →	13	19	42	44	54	95	32
i = 6 →	**13**	**19**	**32**	**42**	**44**	**54**	**95**

Algorithm `ord_secC`

- Each element is compared with the next and swapped if bigger than it
- This way, elements are promoted through the vector until a bigger one is found
- In particular, the larger will be moved to the upper position in the first iteration
- At each iteration, the biggest available is placed at the right location in the upper part of the vector

Application example
`ord_secC`

Initial vector:	44	54	13	42	95	19	32
list_length = 7 →	44	13	42	54	19	32	95
list_length = 6 →	13	42	44	19	32	54	95
list_length = 5 →	13	42	19	32	44	54	95
list_length = 4 →	13	19	32	42	44	54	95
list_length = 3 →	13	19	32	42	44	54	95
list_length = 2 →	**13**	**19**	**32**	**42**	**44**	**54**	**95**

Algorithm `ord_secD`

- Similar to the preceeding, but differentiated even and odd phases:
  - During even phase, each element with even index is compared with the next one, and swapped if bigger than it
  - During odd phase, each element with odd index is compared with the next one, and swapped if bigger than it
- As many phases as elements in the vector are required to complete the sorting

Application example
ord_secD

Initial vector:	44	54	13	42	95	19	32
phase = 0 →	44	54,	13	42,	19	95,	32
phase = 1 →	44,	13	54,	19	42,	32	95
phase = 2 →	13	44,	19	54,	32	42,	95
phase = 3 →	13,	19	44,	32	54,	42	95
phase = 4 →	13	19,	32	44,	42	54,	95
phase = 5 →	13,	19	32,	42	44,	54	95
phase = 6 →	**13**	**19**,	**32**	**42**,	**44**	**54**,	**95**

Tasks

UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

Task 1: measure sequential algorithms times

- Compile program `OrdenaVector.c` as described in the source code
- Measure the execution times of each algorithm for different vector sizes:
  - Try at least 5 sizes: from 10 up to a number when the slowest algorithm takes at least 1 minute
  - For each measure, repeat it 5 times, discard the fastest and the slowest, and take the average of the remaining three
- Summarize in a table the times you got for each vector size and sorting algorithm
- Write down your comments, pointing out how the times increase for each algorithm with respect to vector size(proportionally?, faster?, equally for all algorithms?...)

Task 2: OMP parallelization

- File `OrdenaVectorOMP.c` has been obtained from the parallelization of algorithm `ord_secD` from `OrdenaVector.c`:
  - name `ord_secD` has been changed to **`ord_parD`** and, in the main program, `ord_secD` has been replaced by `ord_parD`, as well as word “secuencial” for “paralelo” in `printf`
- You are requested to keep the example to perform the corresponding modifications to `OrdenaVectorOMP.c` (including `main()`), as described:

Task 2: OMP parallelization

- Create parallel sections there where it makes sense
  - Set private and shared variables and, where necessary, protect the updating of the shared variable
  - If you parallelize a sorting algorithm:
    - **Change its name, replacing `sec` by `par`**
    - Replace the sequential method call by the parallel one and fix the text "secuencial" into "paralelo" in the corresponding `printf`
- For each `for` in the program, decide if it can be paralellized or not
  - Take into account that **it's not the same thing a parallel `for` than a `for` inside a parallel section**:
    - A **parallel `for`**, in addition to be inside a parallel section, has to be preceeded of a directive that turns it into an **instruction for work distribution**

Task 2: OMP parallelization

- Just before each `for` loop not parallelizable, add a **comment** in the source code telling:
  - If there is a *loop-carried dependency* which avoids such parallelization
    - Understanding the sorting method will help to find the explanation
    - you have two examples of such explanation before the second `for` of function `ord_secC` and before the first `for` of function `ord_parD`
  - If there's any other reason why it can't be parallelized
- Take into account that sorting algorithms **should still work after parallelization**

Optional Task

- Extra point
- Add an algorithm **`ord_parDm`** that improves `ord_parD` in the following way:
  - Avoid that at each iteration of the first `for`, that is, at each phase, a team of threads is created and destroyed
  - The resulting `ord_parDm` should be faster than `ord_parD`
- Replace call to `ord_parD` by `ord_parDm` in the main program

Task 3: improve parallelization

- Compile `OrdenaVectorOMP.c` with the same options used to compile `OrdenaVector.c`
- For **the same vector sizes** than in task 1, compare the running times of each parallel algorithm with:
  - The corresponding for the sequential one
  - The fastest sequential algorithm for that vector size
    - Use the same procedure for the measuring than in task 1
    - Compute the speed gain for the comparison
    - Summarize the results in a table

Assignment upload

- Groups of 1 or 2 students
- One student per group will upload the assigment to Campus Virtual in 3 weeks time:
  - File `OrdenaVectorOMP.c` from task 2 (including optional in case it was done)
    - Should include at the header of the file 2 comments:
      - Names of authors
      - Processor type and number of cores used for the measurements
  - Pdf file with the results of tasks 1 and 3

Remeber cheating is a bad idea

We cross-check all uploads!!

Evaluation

- Task 1: 2 points. We value:
  - Clearness and completeness of the timings table
  - Explanation about the obtanied results
- Task 2: 6 points. We value:
  - The creation of the right parallel sections
  - The identification of the private and shared variables
  - The identification of the parallelizable and comments of the non-parallelizable `for`s

Evaluation

- Optional task: 1 point. We value:
  - Error free implementation
- Task 3 (and result of task 2) 2 points. We value:
  - The program is correct
  - Clearness and completeness of the timings table

Errors which imply severe misunderstanding of main OpenMP concepts, may lead to a penalty