Introduction to Parallel Computing

Jesper Larsson Träff
Technical University of Vienna
Parallel Computing

Shared-memory programming, OpenMP and/or pthreads

Three „project" exercises

Implementation, test, benchmark

Hand-in: brief explanation, including correctness argument (informal), testing summary, benchmark

Presentation: ½ hour per group

Due date: hand-in mid-January, presentations end-of-January, exact dates TBD

Parallel Computing

Exercise 1: pthreads or OpenMP

Implement the 3 parallel prefix sums algorithms from the lecture:
- Recursive parallel prefix with auxiliary array y
- In-place iterative algorithm
- O(nlog n) work algorithm (Hillis-Steele)

All algorithms shall work on arrays of some basetype given at compile time (int, double, …) with the „+" operator

Implement non-intrusive „performance counters" for documenting that the work is indeed O(n) and O(n log n)

The implementations shall be correct for all array sizes n

Test and benchmark the implementations, for OpenMP compare to „reduction" clause

Parallel Computing

Hints:

- `#define ATYPE int`

- Performance counters shall count the number of + operations and the number of array accesses (if there are more than + operations), but shall affect execution time as little as possible. No global variables! No critical sections/locks! Idea: use additional array, perform summation after prefix sums computation

- For OpenMP summation can be implemented with a summation variable and a reduction-clause; benchmark this, and compare to the full prefix-sums implementations. Bonus: can the prefix-sums algorithms be simplified (less operations) to compute only the total sum?

Parallel Computing

Exercise 2: pthreads or OpenMP

Estimate the effects of false sharing by implementing the simple matrix-vector computation from the lecture. The implementation shall work for an nxm matrix A and m-vector x, and compute y = A*x

The implementation consists of two nested loops. Experiment with different loop tilings/blockings, either explicity or by OpenMP schedule clauses, to achieve various cache sharing behaviors. Try to establish best and worst case. Show results as functions of n and m. Experiment with placement of threads in the 48-core system for the best and worst-case loops, and document effects of placement.

Bonus: discuss algorithms/implementations that would be immune to false sharing

Exercise 3: OpenMP

Implement the work-optimal merge algorithm for merging two sorted arrays of size n and m in $O((m+n)/p+\log n+\log m)$ steps. The implementation shall work for all n and m, but may assume that elements in the two array are all different

Describe briefly the special cases for the binary search for locating subarrays, and how this leads to all sub-merge problems having size $O(n/p+m/p)$.

Argue for correctness by testing

Benchmark and compare to standard merge implementation from lecture (or better one, if known)

Hints:

Test cases could be as follows. All elements in first array smaller than elements of second array; perfect interleaving, random-block interleaving; all elements of second array smaller than first array

Easy correctness test: first array has even elements, second array odd elements, verify (in parallel) that resulting array has elements 0,1,2,… (mutatis mutandis when n≠m), don't forget to clear result array

Bonus: how can the algorithm be extended to allow element repetitions? Which properties can be guaranteed?

Bonus: can the algorithm be used for implementing a parallel mergesort? What is missing?

Testing, correctness

Programs shall do something sensible for all inputs, never crash. If there are conditions on input, terminate (e.g. „n has to be power of 2“, …) when not fulfilled

Construct small set of test cases, including the extreme cases, argue that this covers the program execution, construct such that verification is easy (and can be implemented in parallel)

Measuring time, benchmarking

Parallel performance/time varies… (system availability, „noise")!!!

Aim: accurate, robust, reproducible measurements (and fast)

- Benchmark on many input instances and sizes – not only powers of two or other special values
- Repeat
- Report average (eliminate outliers), or better: best seen, minimum time

Recall: Tpar is time for last thread/core to finish!! For OpenMP, time in master thread, more care required for pthreads

Use wall-clock time, not CPU time

OpenMP: `omp_get_wtime()`
pthreads: on your own, `clock_gettime()`, or `gettimeofday()`

- Plot time as function of problem size, fixed number of threads
- Plot time or speedup as function of number of threads/cores, fixed problem size (but for different sizes)

Use gnuplot (or something more modern)

Pthread implementations: try not to measure `pthread_create` time. Bonus: what is the cost of thread creation?

Hand-in

Short report, 1-10 pages (depending) plus performance plots.
Be ready to discuss this at presentation, also program code

Be concise, clear, brief:
- What you have done
- What you have not done („the program assumes p is even"…)
- Be honest – things that don't work
- What you intend to show with the experiments