Introduction to Parallel Computing
Programming Projects

Jesper Larsson Träff, Angelos Papatriantafyllou
Vienna University of Technology
Parallel Computing
{traff,papatriantafyllou}@par.tuwien.ac.at

Finish, hand-in:

MONDAY, 2nd February 2015
Hand-in via TUWEL:
Report, program-code & access to the programs

Examination: Slots February 9-13th, 2015

Group-wise examination, ½ hour, based on project, sign-up via TiSS

Parallel Computing Project Exercises

Concrete, small programming projects on material from the lectures:

- understand and implement basic algorithms with potential for linear speed-up

- Using the three parallel programming frameworks: pthreads/OpenMP, Cilk, MPI with C

Understand, implement, test, benchmark, engineer, conclude!

Document by short report plus code. Oral presentation (defense, examination) at end of semester

Goal

- Understand a basic problem in parallel computing
- Gain practical experience with some well-established, current parallel programming frameworks
- Demonstrate speedup and scalability – and/or understanding where obstacles and limitations are
- Document in a concise form (including code), support conclusions by experiment

Execution

Can start now… (get account via TUWEL, deadline 17.11.14)

Use systems at TU Wien (saturn, jupiter), can develop at own PC (OpenMP in gcc, Cilk and MPI, e.g., mpich, can be downloaded and installed)

Finish by end of semester (2.2), hand-in via TUWEL

Groups of two; group (ideally) gets same grade (unless blatantly clear that there is a huge asymmetry)

Groups must be registered in TISS

Hand-in

Short report with performance plots (8-15 pages), code not in report but available. PREFERABLY ENGLISH!

1. Problem statement, hypothesis: what will you try to show
2. Explanation of algorithm (can use code-snippet), if needed argue for correctness
3. Implementation in frameworks (can use code snippets for explanation)
4. Correctness/testing (brief)
5. Experimental set up: benchmarking strategy (number of repetitions, statistics), machines, problem sizes, parameters
6. Experimental results: performance, SPEEDUPs
7. Summary of results, comparison of the machines and the frameworks (quantitative an qualitative)

Short report with performance plots (8-15 pages), code not necessarily in report but must be available. PREFERABLY ENGLISH!

Be concise, clear, brief:

- What you have done
- How you tested (main test cases, problems encountered)
- What you have not done (assumption like: „the program assumes p is even", „n must be a power of two", …)
- Be honest – things that don't work
- What you intend to show with the experiments
- What came out
- Cite the literature you have used (excluding lecture slides), acknowledge other sources (www, tech documents, friends, …)

Hand-in all solutions (TUWEL/email) at the latest Monday 2.2.2014!

Hand-in

- Report, including plots
- Code

As zipped tar-file in TUWEL

Access to program code (leave on saturn/jupiter): we will take samples (compile&run)

Hand-in all solutions (TUWEL/email) at the latest Monday 2.2.2014!

Grading

Grade will be based on presentation/discussion, and hand-in.

Criteria:

- Completeness: all parts of exercise done
- Correctness, by argument (e.g. merging, prefix-sums), and test
- Well chosen test cases, in principle exhaustive, show that you have thought about what needs to be tested
- Program actually working, given stated restrictions
- Code (readable, enlightening)
- Good plots/tables showing the properties (speed-up, work) of the implementations
- Achieved performance improvement – don't be too depressed if speed-up is modest and less than p

½ hour discussion per group February 2014

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 (25-100 times)
- Report average (perhaps median), and best seen, minimum completion time (do they differ?)

Recall: Tpar is time for last thread/core to finish! For OpenMP, time in master thread, Cilk time in „master" task, more care required for pthreads. For MPI time on all processes, use MPI_Reduce(MPI_MAX) to get slowest process

Use wall-clock time, not CPU time

- OpenMP: `omp_get_wtime()`
- Cilk: `cilk_get_wall_time()`
- pthreads: on your own, `clock_gettime()` or `gettimeofday()`
- MPI: `MPI_Wtime();`

- Plot time as function of problem size, fixed number of threads/cores

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

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

Testing, correctness

Programs shall do something sensible for all inputs, never crash.

If there are conditions on input, terminate gracefully when not fulfilled (e.g. „Sorry: n has to be power of 2", …)

„External testing":
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); also do verification by comparing to sequentially computed result (needed anyway for speedup measurements)

Use "performance counters" to verify that e.g. number of operations (of a certain type) are as expected

Input/output

All implementations take input either from a file, or (better) from an input generator (implement generators for different, relevant test cases).

For OpenMP/Cilk: input stored in array/matrix in shared memory. Structure/representation of the input (row-wise, column-wise, …) is part of the specification of the implementation, and can be chosen freely.

For MPI: input distributed in roughly evenly sized substructures over the MPI processes. Output shall follow the same distribution.

Generation and distribution of input NOT part of the problem, the time for this shall not count in speed-up calculation

Tools (Softare Engineering anno 1995)

- gcc, mpicc, …
- emacs, vi
- gdb, dbx, gprof, valgrind, …
- latex, pdflatex
- gnuplot

- But all standard linux debug tools, plot tools, performance tools, … are allowed, and can be used

To a limited extent:
we can install other tools on the machines if really needed – contact us per email

Rules

Each group presents an own implementation. Both group members will be responsible for all parts of the solution. It is joint work, and not the point to split the project in two parts

Goal is to understand the algorithms and problems, and to get some practical parallel implemention experience

Discussion in plenum (Q&A sessions) and with other groups allowed and encouraged – but should lead to own solution

Solutions (even in part) that are copied from other groups, last years material, or from the internet, … will get lowest possible grade (NOT PASS)

Rules

Each group selects one (1) out of the following five (5) projects

There will be Q&A sessions on Tuesdays:
- 25.11: Q&A
- 2.12: Q&A (possibly)
- 9.12: Q&A
- 16.12: Q&A
- And in January

Start early!

Getting account on the systems (saturn/jupiter)

Use TUWEL: need 4K ssh public key

Instructions on how to log on and use the systems on TUWEL (don't circulate)!

DEADLINE for getting account:
17. November (Monday), 12:00

184.710 Parallel Computing Einführung paralleles Rechnen (VU 4,0) 2014W

NAVIGATION +
184.710-2014W
Participants
Reports
Activities
Assignments
Forums
Resources
General
Parallel computers
Link to TISS Lecture

ADMINISTRATION
Course administration
Turn editing on
Edit settings
Users
Filters
Reports
Grades
Backup
Restore
Import
Reset
Question bank
Switch role to...
My profile settings

LATEST NEWS
Add a new topic...
(No news has been posted yet)

Nachrichtenforum

Parallel computers

For solving the exercises, you are expected to use the shared memory parallel computer `saturn.par.tuwien.ac.at` and the distributed memory parallel computer cluster `jupiter.par.tuwien.ac.at`. Please read the instructions carefully and upload your SSH public key as soon as possible, latest by the deadline specified in the assignment below.

How to use the parallel computers

Read this document carefully.

Upload your SSH public key

Upload your *4096 bit RSA* public key for SSH access to the systems `saturn.par.tuwien.ac.at` and `jupiter.par.tuwien.ac.at`. Keys with a lower number of bits than 4096 or of another type than RSA will not be accepted. The user accounts will be created after this assignment is due.

Topic 2

Topic 3

Topic 4

Topic 5

Topic 6

Topic 7

Topic 8

Topic 9

Topic 10

Project 1: Prefix-sums

Implement algorithms for the (inclusive) prefix-sums problem from the lecture, and compare achieved performance to "best known" sequential implementation

All implementations must work on arrays of some given type (not only a C base type) with an associative function f (like the "+", but on the given type) as the associative operation (i.e., commutativity must not be exploited), and must work for any array size and any number of threads/processes

The functionality can be viewed as a parallel library function. Decide on functionality (in-place, or input to output array; array of elements, or array of pointers to elements; type of function pointer to associative operation) – and explain your choice (motivated by convenience, or by performance)

Example:

Prefix(type *a, int n, (*f)(type *, type *))

OpenMP/pthreads:
1-2) Iterative and recursive, work-optimal solutions, 3) Hillis-Steele, 4) blocked, with Hillis-Steele. Input in array, output in (different or same) array. Verify with (NB: scalable) performance counters the claimed bounds on the number of "+" operations.

Cilk:
Devise a task-parallel, divide-and-conquer work-optimal solution (hint: use a sequential cut-off). Verify work-optimality with performance counters

MPI:
Each process has a block (array) of input, compute the prefix-sums for the whole, distributed array of per-process blocks. Process ranks determine the order of the blocks. Note that this problem is different from that solved by MPI_Scan (how?)

Benchmarking:

Obtaining speedup on simple arrays (integers, doubles) with + is difficult. Try to increase the computation per element pair by for instance considering multiplication of small, Boolean matrices (3x3, 4x4, …), use a simple n^3 algorithm for the multiplication ; and show the speedup as the matrix-size increases

Project 2: Merging

Implement work-optimal algorithms for merging two ordered sequences stored in arrays (of size m and n)

All implementations must work on array of some given C base type (can be given as typedef or macro) ordered by "<", and must work for any array sizes and any number of threads/processes. It may be assumed that all elements in the arrays are different (Bonus: stable merging)

OpenMP/pthreads: Either of the work-optimal algorithms, be careful with the load-balancing

Cilk:
Either basic, "data-parallel" algorithm, or recursive, divide-and-conquer approach; argue for the complexity of the latter. Bonus: compare the two appraches. Bonus: Implement mergesort

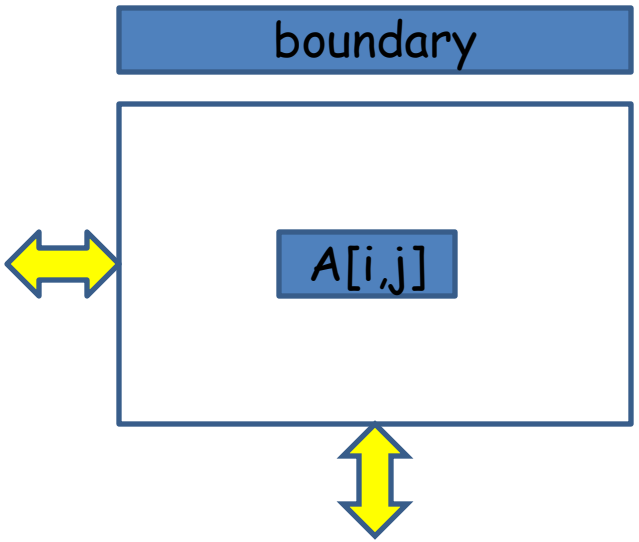
MPI:
Each has a block of the input arrays. The challenge is to implement the binary-like search; for this use one-sided communication (MPI_Win_fence or MPI_Win_lock). It is acceptable to assume that p divides n, and that m=n, to ease the (co)rank computations

Testing and benchmarking:
Extreme cases: all elements of m-array smaller than all elements of n-array; all elements of n-array smaller than all elements of m-array; perfect interleaving of m- and n-elements.
Some regular, controllable distributions; random, ordered arrays.

Verify by comparing output to sequentially merged sequences

Project 3: 2-dimensional 4-point stencil

Implement parallel 2-d stencil computations: given nxm matrix with boundary conditions given in 4 vectors, iterate the 2-d stencil update over some given number of iterations.



The input should be a matrix and vectors of doubles. It is acceptable to assume that p divides n and m

A[i,j] <- (A[i-1,j]+A[i+1,j]+A[i,j-1]+A[i,j+1])/4

The "best known" sequential implementation should not spend any extra steps in moving data between arrays (see lecture)

OpenMP/pthreads: Investigate and explain performance differences for updating row-wise, column-wise, or diagonally. Is it possible to save space, i.e., not to maintain a full, extra nxm matrix?

Cilk: Is there a natural, task-parallel formulation of the stencil update?

MPI: The input matrix is assumed to be distributed as n/rxm/c submatrices over the p processes, where p = rc. Use MPI vector datatype for the column wise exchange. Give a theoretical speedup estimate. How should r and c be chosen for best performance? (Hint: vector communication could be slower than consecutive communication). Give implementations using 1) MPI_Sendrecv (beware of deadlock), 2) non-blocking point-to-point, and 3) one-sided communication for the exchange. Bonus: is there an advantage by using a larger "halo" of "ghost cells"? What is the trade-off? Bonus: Try MPI 3.0 neighbor collectives.

Project 4: Bucket- and radixsort

Implement variants of (stable) bucket (counting) sort: sort a given array of n integers in some range [0,R[; normally R<<n, but this should be a parameter of the implementations. Must work for all n and p and R.

Bucket (counting) sorts by putting each element a[i] into its bucket B[a[i]], and then outputting the contents of the buckets, one after the other. The challenge is to compute the size of the buckets and index correctly when putting elements into buckets.

The MPI algorithm of the lecture uses MPI_Exscan and MPI_Allreduce for this; for the performance analysis it is crucial that both run in O(m+d) steps for m-element input vectors, for some small d (network property, e.g, log p). OpenMP and Cilk do not have this functionality; they have locks and atomics instead.

OpenMP/pthreads: Give a bucket sort algorithm using atomics or locks for counting and managing buckets. How badly does this perform (compared to "best" sequential bucket-sort implementation)? Try to devise a better parallel algorithm, using variants of the prefix-sums problem (possibly: merging), implement this as far as possible.

Cilk: Same considerations as above

MPI:
Implement the integer bucket sort algorithm from the MPI lecture. Assume an array of integers in a given range [0,R[distributed in roughly equal sized blocks over the MPI processes (it is acceptable to assume that p divides n)

The algorithm uses MPI_Allreduce and MPI_Exscan to compute the size of the buckets and to make it possible to determine for each array element its position in the sorted output. The (implementation) difficulty is to use this information to set up an MPI_Alltoallv operation to perform collectively the redistribution of the array elements into sorted order

Project 5: Fast, Discrete Fourier Transform (DFT, FFT)

The Discrete Fourier Transform (DFT) of an input vector x[0…n-1] is a vector y with

$y[j] = \sum_{0 \le k < n} \omega^{(jk)} x[k]$

where the complex number $\omega = e^{(i\, 2\pi/n)} = \cos(2\pi/n) + i \sin(2\pi/n)$ and $i = \sqrt{-1}$

ω^j is the j'th n'th root of unity (ω is called a primitive n'th root of unity)

Computing the Discrete Fourier Transform y of x takes $O(n^2)$ operations

The FFT algorithm (which easily follows from properties of the n'th roots of unity) computes the Discrete Fourier Transform y in O(n log n) operations, when n is a power of 2

```
FFT(x,n)
{
  if (n==1) return;
  for (j=0; j<n/2; j++) {
    z1[j] = x[j]+x[n/2+j];
    z2[j] = (ω^j)*(x[j]-x[n/2+j]);
  FFT(z1,n/2);
  FFT(z2,n/2);
  for (j=0; j<n; j++) {
    if (even(j)) y[j] = z1[j/2]; else y[j] = z2[j/2];
}
```

1. (Optional warm-up, but good – manipulation of roots of unity): Prove correctness of the FFT algorithm, e.g. that FFT computes the same vector as the specification says.
2. Derive the sequential complexity of the FFT algorithm (number of operations; use O to hide constants)
3. Show the memory access pattern of the algorithm for each recursive invocation
4. Convert the recursive version into an iterative version, implement both the trivial O(n^2) algorithm, the recursive and the iterative FFT
5. Estimate experimentally the n for which the FFT becomes faster than the trivial O(n^2) algorithm

For the implementations, use all the tricks you can think of. State which reference works/implementations you consulted (by try yourself!!); there is a lot!

OpenMP, Cilk: Give parallel implementations of the FFT algorithm, using the sequential formulation that is most suited. Assess speedup experimentally relative to your best sequential implementation . Bonus: compare also to the performance of the FFT in the MKL (Math Kernel Library, available on saturn). For OpenMP: experiment with different schedules and chunk sizes

MPI: Show how to adopt the algorithm to the case where both n and p are powers of 2, and $n \geq p$. State the complexity of the algorithm, and estimate experimentally speedup for (very) large n

Bonus: FFT when n is not a power of 2 (and p is not a power of 2)