Introduction to Parallel Computing
Programming Projects

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

Parallel computing projects

Concrete, small programming projects on material from the lectures

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

- Programming frameworks: pthreads/OpenMP, Cilk, MPI in C

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

Understand,implement, test!! Write short report. Short oral presentation

Hand-in

Short report, 1-3 pages (depending) per project plus performance plots (1-5 pages). Be ready to discuss this at presentation, also program code

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

Important: Specify where your code is available!!

Hand-in: Deadline 28.1.2013

Short report, 1-3 pages (depending) per project plus performance plots (1-5 pages). Be ready to discuss this at presentation, also program code

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

Hand-in all solutions (paper or email) at the latest Monday 28.1.2013!!

Grading

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

Criteria:

- 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
- 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 end of January: 31.1 and 4.2.2013

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 and best seen, minimum completion time

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

•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?

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. „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)

Tools

Basically, all standard debug tools, plot tools, performance tools, … are allowed, can be used

To a limited extent: we can install on the machines – contact us per email

This is not a tools or SE lecture…

Rules

Each group presents an own implementation

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

Discussion in plenum 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)

Shared-memory programming, OpenMP and/or pthreads

Project 1: prefix-sums
Project 2: matrix-vector multiply/false sharing
Project 3: Parallel merge

Hand-in: solve 2 out of 3 projects (choice free)

Project 1: pthreads or OpenMP

Implement the 3 parallel prefix sums algorithms from the lecture:

1. Recursive parallel prefix with auxiliary array y
2. In-place iterative algorithm
3. 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, and any number of threads, 1,…,max

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

Hints for project 1:

•`#define ATYPE int // compile time type, can change`

•„Performance counters" to 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, each thread record own number of operations, 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?

Project 2: pthreads or OpenMP

Estimate the effects of false sharing by implementing the simple matrix-vector multiplication with bad and good loop tiling. The implementation shall work for an nxm matrix A and n-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.

Project 3: pthreads or OpenMP
Implement a 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, any C base datatype, but may assume that elements in the two array are all different

Either:

- Algorithm 1: binary search from block starts of array a into array b. Merge (in parallel) all pairs no larger than (m+n)/p; handle larger pairs by binary search from array b to a

- Algorithm 2: binary search from block starts of a and block starts of b. 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)

Project 3: pthreads or OpenMP

Implement a 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, any C base datatype, but may assume that elements in the two array are all different

- Argue for correctness by testing

- Benchmark and compare to standard, sequential merge implementation from lecture (or better one, if known), report speed-up

Hints for project 3:
Test cases could be as follows. 1) All elements in first array smaller than elements of second array; 2) perfect interleaving, even elements in first array, odd elements in second array; 3) random-block interleaving; 4) all elements of second array smaller than first array

Easy correctness test: case 2), 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?

Shared-memory programming, Cilk

Project 1: task-parallel vs. data-parallel prefix-sums
Project 2: task-parallel vs. data-parallel merge (Bonus: mergsort)

Hand-in: solve 1 out of 2 projects (choice free)

Project 1: Cilk

Give a recursive, task-parallel solution to the prefix sums problem and implement in Cilk. The solution should use O(n) operations and thus have the potential for linear speed-up

Compare the performance to a solution (in Cilk) using either of the three data parallel algorithms from the lecture. Implement these by using the data parallel construct from the lecture

Project 2: Cilk

Implement the recursive, task parallel merge algorithm from the lecture.

Compare to an implementation of the binary search based algorithm from the lecture (also implemented in Cilk); use the data parallel construct to execute the desired number of binary searches in parallel

Comment of „ease of implementation“, and achieved performance

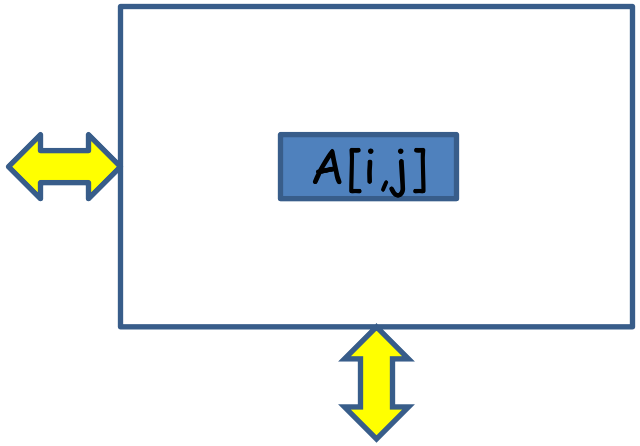
Distributed memory programming: MPI

Project 1: 2-dimensional stencil computation
Project 2: Distributed memory Scan (prefix-sums)
Project 3: Matrix-vector multiplication
Project 4: Integer(bucket) sorting

Hand-in: solve 3 out of 4 projects (choice free)

Project 1: MPI Jacobi-iteration/stencil computation

Let A be a 2-dimensional nxm matrix, that is to be distributed over the p MPI processes. Implement a distibuted version of the two-dimensional stencil-computation: each MPI process has a submatrix, and therefore needs to exchange (in each iteration?) only the elements on the border of the submatrix with its neighboring processes



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

•Give a „generic" implementation that allows distributions of *A* in rows, in coloums or in n/c x m/r blocks, where cr = p.

•Implement the border exchange with MPI_Sendrecv communication (other implementations allowed!); beware of deadlock

•Try to give an „analytical" estimate for the running time, depending on n and m, c and r, and the number of MPI processes. It can be assumed that communication time is dependent as O(k) on the size of the data being communicated.
•What is the matrix distribution giving the best performance under this model?

•Benchmark the implementation, test for correctness (compare to sequential solution, e.g.) . Does the measured performance correspond to the expectations derived from the model?

Benchmarking:

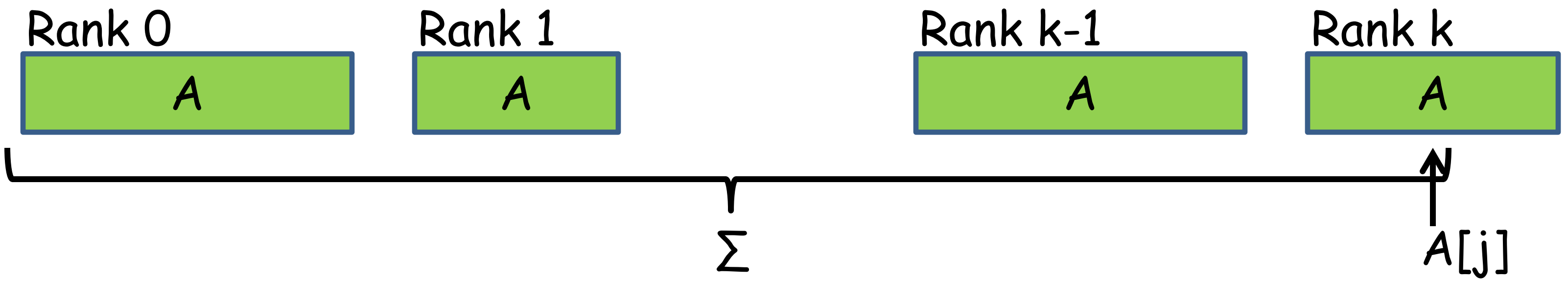
- Vary total size nxm
- Vary distribution, rc = p

Project 2: MPI Inclusive scan

Given an array A distributed blockwise over the p MPI processes. Implement an algorithm (see 2nd lecture!) for computing all inclusive prefix-sums of A: The function

```
arrayscan(int A[], int n, MPI_Comm comm);
```

shall compute for MPI rank i in A[j] the sum (∑(k=0; k<ni'): A[k] for each rank i'<i)+A[0]+…+A[k]



Operation is integer sum, „+“; but only associativity should be exploited for the parallelization. Note that the processes may contribute blocks of A of different sizes.

Implementation hint: compute prefix sums of blocks locally, use a scan algorithm (as in lecture; e.g. 3rd algorithm) to compute all prefix sums of local sums, add prefix locally:

1. Locally compute Scan(A,n), let B=a[n-1] for each process
2. Do distributed ExScan'(B) to compute for rank i the sum B0+B1+…+B(i-1)
3. Locally compute for rank i: A[j] = A[j]+B(i-1) for 0≤j<n

Step 2 is the crucial step and requires MPI communication. This can be done by MPI_Exscan – but the task of the project is to implement an own algorithm (hint: Hillis-Steele)

Tasks:

•Establish correctness by comparing to sequential scan

•What is the asymptotic running time of your algorithm as a function of n and p? Which assumptions do you need for the estimate?

•Compute speed-up relative to sequential Scan for different (large) n (=100,000, =1000,000, =10,000,000, …)

•How is this function different from MPI's scan?

Project 3: Matrix-vector multiplication

Implement the two different matrix-vector multiplication algorithms from the lecture (MPI_Allgather and MPI_Reduce_scatter). Benchmark (sound principles: repetitions, minimum of last process to finish) for a few select matrix orders (n=100, n=1000, …) and different number of MPI processes

•Verify result (how?), either:
1. make it possible to precompute result, or
2. compare to sequential solution

•Speed-up relative to single processor solution?
•Compare and discuss performance differences of the two algorithms (if any)

Assumptions: you may assume that p divides n. Bonus: what if not?

Theory bonus:
both algorithms run in $O(n^2/p+n)$ operations, and are scalable for up to p processes. Is it possible to combine the two algorithms to achieve scalability to larger numbers of processes?

Hint: rxc blockwise matrix distribution; consult the book by Grama, Gupta, Karypis, Kumar

Project 4: MPI integer bucket sort

Implement the integer bucket sort algorithm from the lecture. Assume an array of integers in a given range [0,R-1] distributed in roughly equal sized blocks over the MPI processes.

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

Tasks:

•Show correctness by testing

•Benchmark and show speed-up/decrease in run-time for different array sizes n (with m = n/p elements per MPI process) and different ranges R. Is there a „best" choice of R? Argue also theoretically, assuming that MPI_Exscan and MPI_Allreduce both have complexity O(m+log p)

Bonus: use the bucket sort algorithm to implement a full-fledged radix sort. How should the radix be chosen?