Introduction to Parallel Computing
Projects/exercises

Jesper Larsson Träff
Technical University of Vienna
Parallel Computing

Shared-memory programming, OpenMP and/or pthreads

Three „project" exercises

Implementat 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?

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?

Distributed memory programming: MPI

Four(!) „project" exercises

Implementat, test, benchmark

BUT: (1 or 2) and (3 or 4)

Hand-in: brief explanation, 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: Safe programming with point-to-point operations

Given an array A of n floats, iterate the following computation (as in lecture)

A[i] <- 1/3(A[i-1]+A[i]+A[i+1])

where the rhs denotes the values in the array of the previous iteration. Let the number of iterations be fixed (say 1000). Distribute A evenly across the MPI processes such that each process has a block of approx. n/p consecutive entries.

Implement and compare 3 different versions with different methods for exhanging values at block boundaries, and compute speed-up relative to a sequential iteration:

Version 1: Unsafe - does it deadlock?

Process i: // with modifications for first and last process
MPI_Send to rank i-1
MPI_Recv from rank i+1
MPI_Send to rank i+1
MPI_Recv from rank-1

(even worse:)

Process i: // with modifications for first and last process
MPI_Send to rank i-1
MPI_Send to rank i+1
MPI_Recv from rank i+1
MPI_Recv from rank-1

Version 2: Even-odd scheduling

Even ranked processes does MPI_Send followed by MPI_Recv, odd ranked processes does MPI_Recv followed by MPI_Send. Is it possible to achieve better speed-up this way?

Version 3: MPI_Sendrecv or non-blocking operations

Implement using either the combined MPI_Sendrecv, or explicitly MPI_Isend and MPI_Irecv

For all three versions: ensure correctness (by checking against sequentially computed result); compute and discuss speed-up relative to sequential solution

Bonus 1: implement the computation with one-sided communication

Bonus 2: Use a similar scheme to determine the point where the unsafe version 1 deadlocks; e.g. let A be an array of vectors (aka 2-dimensional matrix), and gradually increase the number of vector elements

Bonus 3: give a hybrid implementation: use OpenMP inside the shared-memory nodes, and communicate between nodes with MPI. Check the MPI standard for how to initialize MPI for threads!

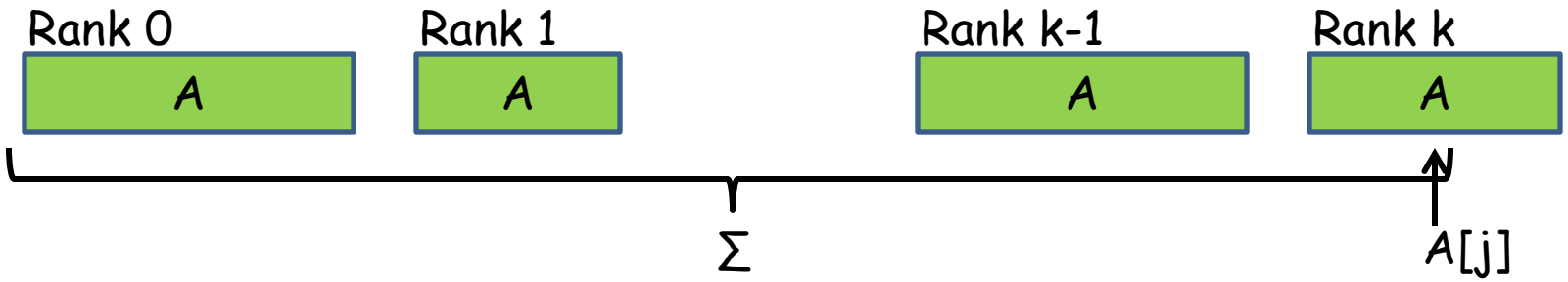
Parallel Computing

Exercise 2: 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

```
myMPIscan(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]



TU WIEN

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 $0 \leq j < n$

Step 2 is the crucial step and requires MPI communication.

Parallel Computing

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?

Parallel Computing

Exercise 3: Matrix-vector multiplication

Implement the two different matrix-vector multiplication algorithms from 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 processes

- Verify result (how? 1. make it possible to precompute, 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

Parallel Computing

Exercise 4: A high-quality benchmark

Implement benchmark for MPI collective operations as described in lecture: a number of repetitions over precomputed counts (not only powers-of-2!), record minimum of last process to finish; variants for from-cache and from-memory communication

Requirements:

- Range from 0 to x Mbytes (x predefined constant, x≥1)
- Basic datatype only, MPI_INT or MPI_DOUBLE; should be compile-time customizable (e.g. `#define BASETYPE MPI_INT`)
- Implement data-from cache and data-from memory (think!); customizable at compile-time, or run-time parameter
- MPI_COMM_WORLD and random communicator (customizable?)

Measure/report:
- MPI_Bcast (fixed root suffices)
- MPI_Reduce (fixed root suffices)
- MPI_Allreduce
- MPI_Alltoall

Plot for different number of MPI processes (all nodes, half the nodes, 1 process/node)

Check/discuss:
- MPI_Allreduce faster than/at least as fast as MPI_Reduce+MPI_Bcast
- How many repetitions seems to be needed to achieve a stable, reproducible result?

If possible: execute with different MPI libraries (NECMPI, mpich, OpenMPI)

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“, …) gracefully 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)

Hand-in

Short report, 1-3 pages (depending) per exercise plus performance plots (1-5 pages). 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

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