Parallel Computing Zusammenfassung

von richie

1 Introduction

Performance of Processors

- *nominal processor performance* is often measured in FLOPS (maximum number of Floating Point Operations per Second)
- performance measured as clock frequency and number of instructions completable per clock cycle (e.g. number of FLOP/cycle)
- number of instructions per cycle determined by architecture
- processor with smaller number of cores is called *multi-core*
  - nominal performance is then nominal performance of one core times the number of cores

Parallel vs Distributed Computing

- *parallel computing*: efficiently utilizing *parallel* resources to solve computational problems
  - deals with algorithms, their implementations and the structure of the computer architecture
- *distributed computing*: make *independent*, non-dedicated resources available to solve specific problem complexes
- *concurrent computing*: manage resources and processes that may or may not progress at the same time
- parallel computing problems often require a lot of interaction
- distributed resources may not be directly available, be distributed over a large area, may fail or change

Sample computational problems

- matrix multiplications
- sequence merging
- graph problems
- sorting problems
- reduction problems (summing or finding maxima...)
- etc.

The PRAM Model

- stands for *Parallel Random Access Machine*
- unrealistic model (but useful)
- processors are strictly synchronized (lockstep)
- memory may be accessed simultaneously, three variants
  - *EREW* (Exclusive Read Exclusive Write), memory cant be accessed simultaneously
  - *CREW* (Concurrent Read Exclusive Write), memory cant be written simultaneously
  - *CRCW* (Concurrent Read Concurrent Write)
    - in *Arbitrary CRCW PRAM* either simultaneous written value may be kept
    - in *Priority CRCW PRAM* processors have priority, the highest priority processor writes

Flynns Taxonomy

- differentiate machines based on instructions and data
- *SISD* (Single-Instruction, Single-Data), sequential computer
- *SIMD* (Single-Instruction, Multiple-Data), single instruction can operate on bigger amount of data (e.g. arrays), vector instructions are example of SIMD
- *MIMD* (Multiple-Instruction, Multiple-Data), general PRAM machine, each processor has own instructions and own data
- *MISD* (Multiple-Instruction, Single-Data), may be a pipline-like system where single data stream passses through different stages

Sequential and Parallel Time

- *Seq* for sequential algorithms, *Par* for parallel
- $T_{seq}(n)$ is sequential running time in amount of steps, $T_{par}(p, n)$ is parallel time with p processors

Speed-up

- Absolute speedup: $S_p(n) = \frac{T_{seq}(n)}{T_{par}(n)}$
  - measures gain of parallel algorithm over sequential

Linear speed-up is best possible

- parallel algorithm with p cores can be simulated on single core in $pT_{par}(p, n)$

- if speed-up would be non-linear, the simulation could run faster than the best known sequential algorithm (which would make the algorithm the best known)
- in practice *super-linear speed-up* may happen when work of sequential and parallel algorithms is not the same, e.g. randomized algorithms
  - also possible for search algorithms

Cost and Work

- cost is time of p processor cores occupied with Par
  - equals $pT_{par}(p, n)$
- work $W_par(p, n)$ is number of operations of algorithm
  - for sequential algorithms, work is same as time
  - for parallel algorithms work is total work of all processors
- algorithm is *Cost-optimal* if cost $pT_{par}(p, n)$ is $O(T_{seq}(n))$ (for a best known sequential algorithm)
- algorithm is *Work-optimal* if $W_par(p, n)$ is $O(T_{seq}(n))$ (for a best known *Seq*)
- cost optimal algorithms have linear speed-ups

Relative Speed-up and Scalability

- relative speed-up is ratio of parallel running time with one processor to running time with p processors
- $\text{SRel}_p(n) = \frac{T_{par}(1,n)}{T_{par}(p,n)}$
- fastest possible running time is $T\infty(n)$
- parallelism: $\frac{T(1,n)}{T\infty(n)} = \frac{T(1,n)}{T(\infty,n)}$
  - denotes the largest possible speed up

Overhead and Load Balance

- parallel algorithms often perform more work than best sequential ones
- excess work is called *overhead*
- can occur through communication, synchronization or extra preprocessing
- if overheads are in bounds of sequential work $O(T_{seq}(n))$ the algorithm can still be work-optimal
- communication intervals of algorithms are called *granularity*
  - rare communication: *coarse grained* algorithm
  - frequent communication: *fine grained* algorithm
- processors may have different amounts of work
- if $T_{par}(i, n)$ is the time a processor i takes, the *load imbalance* is the biggest difference between the times of two processors, over all processors

- *load balancing* is the problem of making sure the times of all processors are about the same
- *static load balancing* is when the amount of work is divided up front between processors
  - *oblivious static load balancing* is the subdivision of the problem only by input size and structure, regardles of actual input
  - *adaptive, problem-dependent load balancing* is load balancing where the input itself is taken into consideration
- *dynamic load balancing* is the exchange of work during execution of the algorithm
- problems where input can be statically distributed and no further interaction is needed are called *embarrassingly* or *trivially* or *pleasantly parallel*

Amdahls Law

- assuming algorithm can be subdivided into a sequential fraction s and a perfectly parallelizable fraction $r = (1 - s)$
  - maximum speed up by parallelization is $\frac{1}{s+\frac{1-s}{p}} = 1/s$, if $p \to \infty$
- problems with parallel algorithms:
  - input, output
  - sequential preprocessing
  - sequential data structures
  - operations that have to be performed sequentially
- good algorithms have the parallel part not be a constant fraction but decrease with n

Efficiency and Weak Scaling

- *scaled speed up* is when the faster $\frac{T(n)}{t(n)}$ converges, the faster the speed up converges
  - with $T(n)$ as the parallelizable term and $t(n) = T\infty(n)$ as the non parallelizable term
- efficiency is the comparison of *Par* against the best possible parallelization of *Seq*
- parallel efficiency: $E_p(n) = \frac{T_{seq}(n)}{pT_{par(p,n)}} = \frac{S_p(n)}{p}$
  - linear speed up if $E_p(n) = e$ for some e
- iso-efficiency: when efficiency is not constant (constant means it does not change with n) then we have to calculate n as a function of p
  - do this by calculating e as a function of n and p and then changing the formula for n
- *Weak Scalability* is when you want to achieve an efficiency $E_p(n) = e$ there is a function $f(p)$ so that n is in $\Omega(f(p))$

  - so the efficiency varies with the number of processors if the problem size remains constant
  - so a problem may be more efficiently solved with fewer processors
- other definition: if the average work per processor $T_{seq}(n)/p$ is constant at w the running time of the algorithm is constant at $T_{par}(p, n)$
- $f(p)$ is called the iso-efficiency function
  - says how n should grow as a function of p, so efficiency remains constant

Scalability Analysis

- *Strong Scaling* analysis, n is constant, if run time decreases proportionally to p (linear speed up) the algorithm is *strongly scalable*
- *Weak Scaling* analysis, work per processor is constant (through bigger n), if parallel running time is also constant algorithm is *weakly scalable*

[rest skipped]

2 Shared Memory

Caches

- three different types:
  - directly mapped: each block mapped to one distinct position in cache
  - fully associative: each block can go anywhere
  - (k-way) set associative: like directly mapped cache but space for multiple blocks per position
- chache misses
  - cold (compulsory) miss: no data in the cache line where you would look for it (i think)
  - capacity miss: all lines are full, one line has to be evicted (only fully associative)
  - conflict miss: already element in line (only directly and set associative)
- replacement strategies
  - LRU: least recently used
  - LFU: least frequently used
- on write either:
  - block already in cache: block is updated
  - block not in cache: block is allocated (called write-allocate) (may create conflict miss) or memory is written directly (write non-allocate)
- if block is updated:
  - write-through cache: block updated and value is written directly to memory too

  - write back: written to memory when line is evicted
- locality of access (for applications and algorithms)
  - temporal locality: memory address is reused frequently (will not be evicted)
  - spatial locality: addresses of the same block are also used

Matrix-Matrix Multiplication and Cache Performance

- $n \times l$ Matrix A multiplied with $l \times m$ Matrix B
- sequential algorithm in $O(nml)$ or $O(n^3)$ (for squares)
- three loops (for n, m, l) can be parallelized, order of parallelization matters
  - differences because matrices are accessable in row order
  - different index order causes more or fewer cache misses

Recursive Matrix-Matrix Multiplication Algorithm

- recursively split A and B in smaller submatrices until very small then iterative solution

Blocked Matrix-Matrix Multiplication

- split matrices into blocks at start
- then multiply with 6 nested loops
- block size can be chosen depending on cache size
  - called *cache-aware algorithm* (opposite is *cache-oblivious*)

Multi-core Caches

- cache system has several dimensions
- L1 (lowest level) to L3 (or more)
  - L1 is closest to processor, smallest (few KB) and fastest
  - L3 is *Last Level Cache* LLC, several MB big
  - L1 has data and instruction cache
- *Translation Lookaside Buffer* for memory management (pages)
- lowest level caches are private (only for one processor)
- higher level caches are shared
- *cache coherence problem* when one block is updated in one cache and also stored in another cache of another processor
  - *coherent* cache system if line of other cache will be updated at some point in time (can also just mean it is invalidated in the cache)
  - *non coherent* if it will never be updated
- problem solved through *cache coherence protocol*

  - may cause a lot of *cache coherence traffic*
- *false sharing* is when two caches have the same blocks of memory stored
  - update to one adress of block will cause the whole line of other to be replaced

The Memory System

- cache system part of *memory hierarchy*
  - from fast low levels (caches) to slower bigger higher levels
- *write buffer* for memory writes (FIFO)
- for multi-core CPUs not every processor has direct connection to main memory
  - instead connected to *memory controller*
- some processors closer to memory controller than others → address access times are not uniform

Super-linear Speed-up through the Memory System

- through memory hierarchy of large caches, working sets of each processor get smaller and eventually can fit into the fastest caches

Application Performance and the Memory Hierarchy

- *memory-bound*: when execution (and associated reading and writing from memory) of instructions is faster than reading or writing the instruction in memory
- *compute-bound*: other way around, intructions are read faster than they are executed

Memory Consistency

- when process 1 sets a certain flag in memory and process 2 checks if that flag is set it may happen that the flag is still in the write buffer and process 2 may read a wrong value
- frameworks like `pthreads` or `OpenMP` help

`pthreads`

- *thread* is smallest execution unit that can be scheduled

Thread Characteristics

- fork-join parallelism: thread can spawn new threads
- are symmetric peers, thread can wait for any other thread to complete
- have same program (SPMD) but can have different traces (MIMD)
- are scheduled by the OS
- no implicit synchronisation of threads, they progress independently from each other
- share the same memory
- constructs for coordination and synchronization are provided

`pthreads` in C

- use `#include <pthread.h>`

Race Conditions

- when two threads update variable at same time, outcome may be write of either thread
  - result is *non-deterministic*
- special race condition is called *data race*
  - two ore more threads access shared memory, one of accesses is a write

Critical Sections, MutEx, Locks

- lock is programming model to guarantee mutual exclusion on a critical section
- threads try to acquire lock, if granted they enter critical section, release lock when done
  - threads are blocked till lock is acquired
- locks have to be deadlock free
- lock is starvation free if thread will not be starved
- threads at lock will *serialize*, one thread after another will pass critical section
- locks where many threads are competing are called *contended*
- *try-locks* allow execution of some code when lock could not be acquired
- in *spin-locks* threads test by busy waiting
- in *blocking-locks* waiting threads are blocked and freed by the OS
- condition variables are associated with mutex
  - thread waits on variable till a *signal* occurs
  - a thread can *signal* one (maybe arbitrary) thread only
  - a *broadcast* signals all threads at once
- concepts with conditions with signals and waits are called *monitor*

- *barriers* are similar to mutex with conditions
- *concurrent initialization* is where first thread executes initialization code before other threads begin

Locks in data structures

- trivially make data structures work with parallel algorithms by using a single lock for all operations on structures
- other way is to construct *concurrent data structures*

Problems with Locks

- deadlocks can happen easily with locks
  - can also happen when the same thread tries to acquire same lock again
- locks around long critical section cause harmful serialization
- threads crashing during critical section also cause deadlocks
- threads can starve
- priority threads and locks can lead to lower priority threads preventing higher priority threads from continuing

Atomic Operations

- atomic instruction carry out instructions that cannot be interfered with by other threads
- `a = a+27` can be implemented as atomic instruction through `fetch-and-add`
- crashing threads during atomic instructions will not cause deadlocks
- instructions are *wait-free* because they always execute in $O(1)$
- instructions are *lock-free* if any thread will be able to execute instruction in a bounded amount of time (not always given)

OpenMP

Programming Model

- fork-join thread model
  - master thread forks and creates working threads
  - when finished working threads join again back to master thread
- all threads execute same program (SPMD)
- each thread has uníque id

- more threads than processors are possible
- shared and private variables in threads are possible
- synchronization constructs to prevent race conditions

OpenMP in C

- has to be compiled in gcc with `-fopenmp`
- has to be included with `#include <omp.h>`

Parallel Regions

- forking starts at parallel region
- designated by `#pragma omp parallel [...]`
- number of threads in parallel region cannot be changed after start
- thread number set by the runtime environment or library call or `num_threads()` pragma

Library Calls

- `omp_get_thread_num(void)` returns thread id
- `omp_get_num_threads(void)` returns number of threads
- `omp_get_max_threads(void)` returns maximum possible threads
- `omp_set_num_threads(int num_threads)` sets maximum
- `omp_get_wtime(void)` returns wall clock time in seconds
- `omp_get_wtick(void)` returns tick resolution of timer

Sharing variables

- default is all vairables before region are shared, all variables declared in region are private
- sharing of variables set by clause in pragma
  - `private(vars...)` makes uninitialized copies of variables
  - `firstprivate(varis...)` makes copies and initializes value to value before parallel region
  - `shared(vars...)` declares variables as globally shared
  - `default(shared|none)` makes variables shared or not shared by default
- use like `#pragma omp parallel private(a, b, c) shared(d) default(none)`
- good practice to set no variables to shared per default (none)

- shared variables make *data races* possible

Work Sharing: Master and Single

`#pragma omp master`

- declares statement to be executed by master thread only (thread id of 0)
  - other threads will simply skip

`#pragma omp single`

- statement is executed by either one of the threads
  - other threads will wait at end of statement until all threads have reached end
  - can be eliminated by `nowait` clause after `single`
    - might cause race conditions
  - allows making variables private or firstprivate (unlike master construct)

explicit Barrier

`#pragma omp barrier`

- no thread can continue until all threads have reached barrier

Sections

- code is split into small pieces that can be executed in parallel by available threads

`#pragma omp sections`

- outer structure
- end of block is implicit barriers
  - nowait possible
- variables can be designated (first)private

`#pragma omp section`

- marks an independent code section
- which thread executes which section is designated by runtime system
- best case: each thread executes a section, all threads run in parallel

Loops of Independent Iterations

- *loop scheduling* is assignment of iteration blocks to threads
- each iteration has to be executed exactly once
- data races have to be avoided

`#pragma omp for [clauses...] for (loop range...)`

- all threads must have same start and end values for i and same step
- loop ranges have to be finite and determined (no while loops possible)
- end condition has to be in the form of: `i<n, i<=n, i>n, i>=n, i!=n`, with n as an expression or value
- steps have to be in form of `i++, i--, i+=inc, i=i+inc, i-=inc, i=i-inc`
- these kinds of loops are in *canonical form*

`#pragma omp parallel for [clauses...]`

- shorthand for parallel region with one loop
- next statement is for loop
- always barrier after

Loop Scheduling

- how is each thread assigned to each iteration
- loop range is divided in consecutive chunks
- static schedule: chunks have same size and are assigned like round-robin (each thread gets chunk one after another)
  - computation of chunk assignments is very fast (low overhead)
- dynamic: each thread dynamically grabs next chunk it can
- guided: dynamic assignment but chunk size changes, determined by number of iterations divided by threads
- set like `schedule(static|dynamic|guided[, chunksize])`
  - for guided `chunksize` is minimum size
  - if no size given then default size is used
- also possible `schedule(auto|runtime)`
  - runtime: scheduling is set at runtime
  - auto: OpenMP compiler determines

Collapsing nested loops

- transform multiple loops into one

`#pragma omp parallel for collapse(depth) [clauses...]`

- depth is the amount of nested loops to be parallelized

Reductions

`#pragma omp for reduction(operator: variables)`

- is clause of `for`
- operators are ‘+ - * & | ^ && || min max‘

Tasks and Task Graphs

`#pragma omp task [clauses...]`

- task is like a function call
- completion of task may not happen immediately
  - at latest when completion is requested (e.g. at end of parallel region)
- tasks can access any variables of the thread
  - if those variables are changed in the task data races may occur
- task can be designated final and thus not generate any additional tasks

`#pragma omp taskwait [depend(...)]`

- wait for completion of generated tasks
- only dependency clause allowed

Mutual Exclusion Constructs

`pragma omp critical [(name)]`

- threads will wait and only one thread will execute code of critical section
- section can have name
- shared variables can be updated by task in critical region

`#pragma omp atomic [read|write|update|capture]`

- for simple critical sections
- allow fetch and add (FAA) atomic operations
- update is `x++, x = x op ..., ...`
- capture is `y = x++, ...`

Locks

- locks don't have condition variables
- recursive (nested) locks are possible in OMP

Special Loops

- OMP can try to utilize vector operations (SIMD)

`#pragma omp simd [clauses...]`

- followed by canonical for loop
- one thread executes loop but with SIMD instructions

`#pragma omp for simd [clauses...]`

- followed by canonical for loop
- loop executed by multiple threads
- each chunk executed with SIMD instructions

`#pragma omp parallel for simd [clauses...]`

- same as previous but with parallel region declared

`#pragma omp taskloop [clauses...]`

- recursively break iteration range into smaller ranges
- smaller ranges are then executed as tasks
- should be initiated by a single thread
- size of range can be adjusted by `grainsize()`

Loops with Hopeless Dependencies

`#pragma omp ordered`

- loops with dependency patterns that cannot be handled normally can be marked as ordered
- only one possible ordered block in a parallel loop
- other parts of iteration can still be performed in parallel

Cilk

- OMP inspired by Cilk
- since 2018 not supported anymore
- supports constructs `cilk_spawn` (like task), `cilk_sync` (like taskwait) and `cilk_for` (like taskloop)
- executes threads through *work-stealing algorithm*
  - each thread has local task queue
  - when threads run out of local tasks, they steal tasks from other threads until there are no more tasks to steal

3 Distributed Memory Parallel Systems and MPI

Network Properties: Structure and Topology

- distributed memory introduces interconnection network (or *interconnect*)
- entities (cores, multi-core CPUs or larger systems) are connected through *links* (in any form)
  - some entities are simply switches
- interconnect where processors are also communication elements (and without switches) is called *direct network*
- interconnect with switches is called *indirect network*
- *topology* of network can be modeled as an unweighted graph
  - each vertice is an network communication element
  - each arc denotes a direct link between two elements
- graphs are mostly undirected
- diam(G) graph *diameter* is longest shortest path between two nodes
  - is a lower bound on communication steps between two elements
- degree(G) graph *degree* is maximum degree of all nodes
- bisec(G) *bisection width* is minimum number of edges to remove so that graph is partitioned into two equally large subsets
- worst possible networks are *linear processor array* and *processor ring*
- *tree networks* are binary or k-ary trees
  - have bisec(T) = 1
- *d-dimensional mesh networks*
  - processors are identified by their integer vectors
  - special case is *torus network* where edges wrap around
- *hypercube network* is special case of torus network
- modern systems often built as torus networks with 3 to 6 dimensions, called *multi-stage networks*

Communication algorithms in networks

- *unidirectional* communication if only one direction
- *bidirectional* if in both directions can be communicated
  - most modern systems
- broadcast problem: how to transmit data from root to every other node in minimal communication steps
  - lower bound is $\lceil \log_{k+1}(p) \rceil$ in k-degree, network with p nodes
  - algorithm to solve partitions network in $k+1$ smaller networks, root sends data to "virtual roots" of these networks, solve problem for smaller subnetworks

Communication costs

- linear transmission cost assumes time of $\alpha + \beta m$, where α is start up latency and β is time per unit of data

Routing and Switching

- if network is not fully connected, routs make path between two nodes possible

Hierarchical, Distributed Memory Systems

- communication networks with different levels
- processors may have different characteristics and thus live on different compute nodes

Programming Models

- concrete network properties are abstracted
  - model of fully connected network
- processes are not synchronized and communicate with others through explicit or implicit transmission
  - message transmission is assumed to be deadlock free and correct
- distributed system programming models either:
  - *data distribution* centric: data structures distributed according to rules, if one processes changes data structure of another process then through communication
  - *communication* centric: focuses on explicit communication and synchronisation instead of data structures (MPI)

Message-passing Interface (MPI)

- message-passing programming model is to structure parallel processes through sending and receiving messages
- proccesses are *Communicating Sequential Processes*
  - cannot have data races
- MPI characteristics:
  - finite processes
  - each process identified by rank in domain
  - more than one domain possible
  - data is local
  - communication is reliable

MPI in C

- header `#include <mpi.h>`
- functions in `MPI_` "name space" (illegal and punishable by law to use prefix for own functions)
- `MPI_SUCCESS` return value for success (obviously)
- `mpicc` compiler to compile programs

Initializing MPI

- `MPI_Init` to initialize
- end with `MPI_Finalize`
  - `MPI_Abort` forces termination

Error Checking

- MPI only has limited error checking because it is expensive

Communicators

- `MPI_Comm_size` to get amount of processes in communicator
- `MPI_Comm_rank` to get rank in communicator
- all processes are in communication domain `MPI_COMM_WORLD`
  - data type `MPI_COMM`
- `MPI_Comm_split` splits processes of communicator into smaller groups with own communicators
- `MPI_Comm_create` also creates new communicators
- both functions are *collective* (have to be called by all processes)
- `MPI_Comm_free` to free communicators

Organizing Processes

- communicator with grid structure is called *Cartesian communicator*
- created through `MPI_Cart_create`
  - `reorder` flag to reorder ranks so neighboring processes are close on grid
- `MPI_Cart_coords` translate rank to coordinate vector
  - `MPI_Cart_rank` vice versa

Objects and Handles

- a *distributed object* is an object for which all processes that reference it can access it
- distributed objects
  - `MPI_Comm` for communicator object
  - `MPI_Win` for cummincation windows
- local objects
  - `MPI_Datatype` for local layout and structure of data
  - `MPI_Group` for ordered sets of processes
  - `MPI_Status` for communication
  - `MPI_Request` for open (not yet completed) communication
  - `MPI_Op` for binary operators
  - `MPI_Info` for additional info when creating objects

Process Groups

- objects of type `MP_Group`
- groups used locally by processes to order processes
- `MPI_Comm_group` returns an ordered group from a communicator
- `MPI_Group_(size|rank)` like `MPI_Comm_(size|rank)`
- set-like operations on groups possible, `MPI_Group_(union|intersection|difference|incl|excl...)`

Point-to-point Communication

- `MPI_Send` to send data from process to `dest`
- `MPI_Recv` to receive data from `source`
- combined operations `MPI_Sendrecv[_replace]` have argument for `source` and `dest`
- `MPI_Get_(count|elements)` to figure how much data was sent

Semantic terms

- *blocking* operations when function call returns when operation has been locally completed (irregardless of success of if data has been sent out etc.)
- *non-blocking* when function call returns immediately (specified by capital `I` in function name, e.g. `MPI_Irecv`)

Specifying Data

- `buffer` specifies starting adress
- `count` specifies number of elements
- `datatype` specifies MPI type
  - e.g. `MPI_(CHAR|INT|LONG|FLOAT|DOUBLE)`

One-sided Communication

- one process alone initiates communication
  - also specifies actions on both ends
- involved processes are called *origin* and *target*

Collective Communication

- processes communicating collectively
- `MPI_Barrier` blocks until all have reached routine
- `MPI_Bcast` to send data from `root` to all nodes
- `MPI_Scatter` distributes data from process `root` evenly to all other
  - `MPI_Scatterv` if data is not distributed evenly
- `MPI_Gather` gathers data from all other processes in `root`
  - `MPI_Gatherv` if not receiving the same number of elements from each process
- `MPI_Allgather` like Gather but all processes have all data
  - `MPI_Allgatherv` similar to `MPI_Gatherv`
- `MPI_Reduce` reduction of data in `root` with operator `MPI_Op op`
  - operations are assumed to be associative and commutative
- `MPI_Allreduce` like reduce but all processes receive reduction
- `MPI_Reduce_Scatter` first reduce vector in processes, then scatter vector in segments across processes
- `MPI_Alltoall` (could be named `Allscatter`) every process scatters their data to all other
- `MPI_Scan` performs a reduction over a group, where each subsequent process stores the result of the reduction of the current value and the previous values

  - e.g. Scan with SUM reduction processes with values in brackets: 1(1), 2(2), 3(3), 4(4)
    after Scan: 1(1), 2(3), 3(6), 4(10)
- `MPI_Exscan` like Scan but value of process is not used in reduction
  - same example with Exscan: 1(0), 2(1), 3(3), 4(6)