SPP   zB: Performance Modeling, Neural Networks, Parallelism Discovery, Scheduling in HPC & cloud, Scalable Algorithms

C

conditions evaluated to ⟨0 → false / sonst → true⟩

struct name {
  type attribute;
  ...
}

using mit struct name
instance.attribute
referenz → attribute
a→b = (*a).b

Arrays:
  type name[length]; (zu Compilezeit)   length = sizeof(name)/sizeof(type)

Pointer
  type *name;
  pointer++ berücksichtigt sizeof(type of pointer)

& reference
* dereference

workaround for string
  char * with \0 at the end

%c char
%i int
%s char*

I/O
  number Written Characters printf (const char* format, ...)
  Read scanf (const char* format, ...) — adressen Ziel z.B. &my_int

main function
  int main (int argc, char ** argv){ ... }

break; cont,
switch (number)
  case 0: case 1:
   ... break
  default:
}

automatic ↔ dynamic lifetime
sonst malloc (# bytes) : referenz erster byte
  free (ptr erstellt mit malloc) else memory leak
  new ... delete ..., new[], delete[] ...

compiler reads from top to bottom
  ⇒ #include "file.h" from current directory
type func (param); < > from system directory

common library
  <stdlib.h> malloc, free, system, abs, div, rand
  <string.h> memcpy, memcmp, strcpy, strlen
  <stdio.h> printf, scanf

to prevent mehrfaches reinkopieren
  #pragma once
oder
  #ifndef KEY
  #define KEY
   ... code ...
  #endif

shallow copies of everything
arrays decay to pointers
explicit casting like java (can cast pointers arbitrarly

[const] int * [const] pointer
can't change reference value

type aliases typedef existing_typename alias;

C++ - Zero-overhead principle
- don't pay for what not used
- better than if done reasonably by hand

std::cout << val << std::endl; (also flushes)
typecopy; std::cin >> copy;

references
  type& name = other; no own value, always initialized

other modifiers:
begin func [[nodiscard]] - Warning if value not safed
  constexpr - can be evaluated during compile
Ende kopf noexept
  explicit, friend
  std::size_t can store size of any possible object

throw and catch everything

templates (previously inefficiently done by void* — tests at runtime)
requires std::is_arithmetic_v<T>
  template <typename _Ty, int n, ...> no floats allowed
  returntype func(Param){...}   keine expliziten Typeinschränkungen
  calling: func<int, 3>(...); compile produces func for needed types

[] allow multiple param

z.B. type& operator=(const type& other){ ... return *this}
can overload operators

classes - connect data and functionality, resource management
using value_type
  class name {
   public: Bereiche
   private: ← starts with
  ...
  }
  constructor: name (param) : attribute{Param}, ... {...code...}
   ↑ unified constructor
  copy constructor - default shallow copy
  ~name deconstructor

auto - matches greedely every type
  const auto, const auto&

standard containers std::
  vector - managed memory block; const acces, amortized const add (using value)
  list: linear access, const merging
  array<_Ty, _s> fixed size
  string
  queue, stack,
  deque - contineous memory, const access, push, pop
  map<_Ke, _Val> uses operator [] — logarithmic, stored in red-black tree
  unordered_map hash-map — amortized constant

Iterating Containers
  for(auto it = cont.begin(); it != cont.end(); ++it){...}
  kurz for(const auto& [key, val] : m) ← bei map

Lambdas (closure)
  [identifiers, ...] (args) { rumpf }

Parallel Architectures

Flynn's classification 1966

Brick Wall { Power wall, Memory wall, ILP wall
↑ Instruction Level Parallelism

Frequency ⇒ Voltage ⇒ Heat

#data streams \ #instruction streams	Single	Multiple
Single	SISD classical uniprocessor	MISD no commercial
Multiple	SIMD Data parallelism z.B. extensions for multimedia, vector processing ↕ GPU	MIMD Thread-level parallelism + flexibility

Functional ⟩ parallelism
Data
Control ⟩ dependence
Data

popular execution model: Single Program Multiple Data SPMD
potentially different control flow

Single instruction multiple threads SIMT z.B. GPU
~~manages~~ manages threads in groups of parallel threads (warps)
instruction unit broadcasts to all
threads may do nothing instead

Memory Architecture

shared memory = single shared address space
address

UMA - Uniform memory access
∀ CPUs same time ∀ memory
implementation +simple
Bus
Switch

NUMA - Non-uniform memory access
Memory has affinity to a CPU
faster with local memory ~~+scalable~~
+scalable

distributed memory

(Interconnecting)
Network architecture
physical links of nodes (Processors, memory)
protocol
Topology, Routing technique
Bandwidth, Latency
Maximum rate └ sending overhead + time of flight
Aggregated total + Packetsize/bandwidth + receiving overhead
Effective to application

Cache coherence protocol

read newest version
serialize writes (∀ see same order)

Snooping — serialized: only 1 message at time
∀ caches on broadcast medium
∀ copies have sharing status on cached locations
⇒ no centralized state
∀ caches snoop for sharing status
- Write-through caches
  data can be retrieved from main memory
- Write-back caches (dirty)
  data from processor with only copy
  − longer than main memory + bandwidth scalable

z.B. MESI protocol z.B. i7

states:	read from memory if able	memory caches all heard	invalidate	What if cache full and block
Modified	- no other copies			modified?
Exclusive	- memory up to date			
Shared				
Invalid				

indirect/dynamic-interconnection

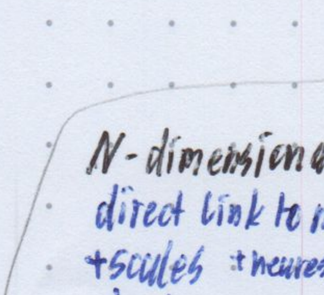
Centralized switched networks

Several links, intermediate switches
for shared and distributed Memory

Crossbar switch
+ Non blocking
− scale N^2 switches

Multistage Interconnection network MIN

z.B. Omega Network
- blocking
perfect shuffle permutation of each stage
blocking
$\log_k N$ stages $\frac{N}{k} \cdot \log_k N$ switches
k×k switches



Fat tree
Leaves: end nodes
vertices: switches
often composed of multiple smaller
same total bandwidth
+ popular for cluster interconnections.

Shared-media

decentralized
listen if free
- only 1 message at time
- collisions degrade performance
- broadcasts — all listen to all receiving expected if nest.
− not scalable
+ cheap

Switched

concurrent messages
of different node pairs
+ scales

BRUNNEN

Lichtenberg

Processor - shared memory for cores
Compute node - typically 2 processors
shared memory fast connection
Server
Cluster - nodes as battalion by scheduler
no shared memory

Lichtenberg II
Big-Mem
Accelerator nodes
└ Nvidia Tesla
infiniband
non-blocking fat tree
Ethernet

Heterogeneity, #Customisation

CPU, GPU - data parallelism
FPGA - problem specific compute and mem
Neuromorphic - asynchronous CMOS circuits
Quantum

Write invalidate ↔ update/broadcast

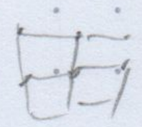
copies
+common
- bandwith

Coherence misses
- true sharing misses — invalidated writing same word
- false — diff. word prevented by smaller blocks

directory-based - für jeden Block 1 location for status
centralized (UMA) ↔ distributed (NUMA) + scalability
- fast hardware & query at ∀ memory reference

z.B. NUMA split address: Node | Line | Offset
directory: Line, cached: bool, where: Node
- only 1 copie saved by directory
Optimization: k entries Bitmap LinkedList
request forwarded to owner, changes owner
Status of Line (dirty ↔ clean) - traffic - useless with only 1 copy
copy invalidates previous

N-dimensional mesh
direct link to neighbors
+scales +nearest neighbor
- borders



Torus wraparound ⇒ no borders
direct (static)
typically 3-6 dimensions
Hank

↔ Distributed switched networks
∀ switches direct link to End-node
switches 1:1 nodes
ratio
{ processor(s) memory

fully connected ↔ ~~Bus~~ Ring
+performance +cost
Dragonfly

Order of updates

- between threads (through network) — requests
- compiler instruction reordering

coherence ↔ Consistency

coherence: single memory

Consistency: ordering access of diffrent memory locations
When to update?
observed order?

Sequential

result = if processors execute sequential
and for individual processor order of programm

\+ simple programming paradigma
− potential performance degregation

Relaxed consistency models

only synchronize important − hard to debug

programm synchronized = Access to shared data ordered by sync. operations

data races − not orderd access to single location

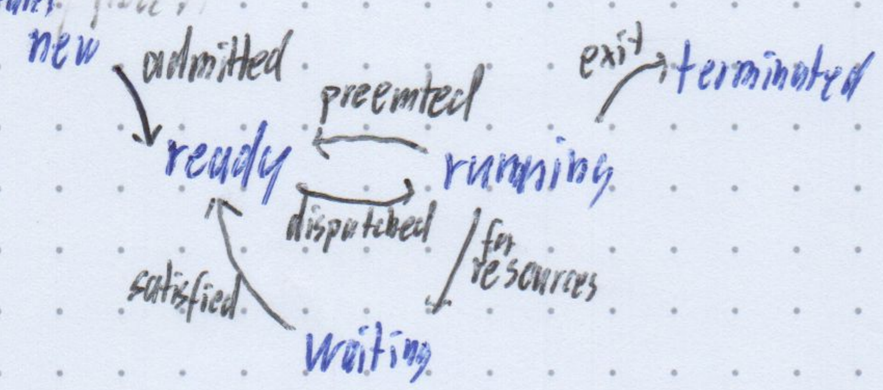
hardware supplied instructions
atomic, detect problems, can be bottleneck

Time sharing / multitasking

multiprogramming
Processor switches process

Process

States process:



Memory

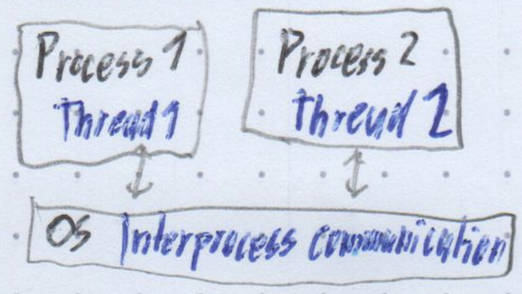
max
- Stack — top, param, return addr, local var
- heap — dynamic allocated
- data — global var
- text — code

Thread

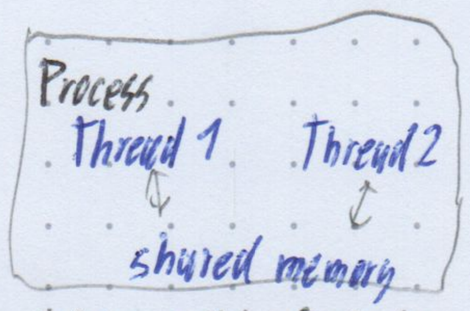
light weight process

shares: Code, data, files
own: Registers, Stack

Communication



explicit, often copies
protected address space
− profound design
− hard writing debugger
\+ scaleable



\+ space, switch efficiently
\+ convienient
− hard to debug race conditions

On-chip multithreading

hides Latency
boosts performance
CPU runs other thread if stalled z.B. Memory

- Fine ↔ Coarse grained
  - Fine: Round robin — − switch overhead
  - Coarse: only switch upon stall — − Locken

z.B. i7 Hyperthreading — up to 4 threads per core
2 threads or processes per core
looks like 2 CPU with shared Memory
\+ concurrency
− conflicts

BRUNNEN TH

3/94

Programming Models – abstraction computer allowing expression parallel algorithms/data structures

- Implementation
  - languages + extensions
  - API
  - compiler directives
- Objectives
  - Productivity
    - code
    - debugging
    - optimizing
  - Portability (comparable performance)

Nodes

+ scale
− re-design

MPI – Message Passing Interface z.B. for Compute Cluster, distributed Memory

ranks as addresses

```
MPI_Send( buffer, count, datatype, RECEIVER, ... );
MPI_Recv                           SENDER
```

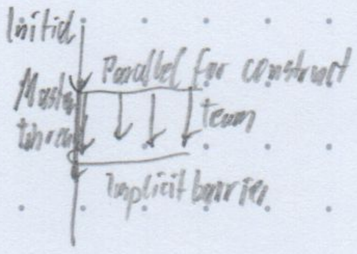
Cores

+ Incremental Parallelisation
− scale
− debug

OpenMP – Multithreading

multiple threads communicate via shared variables

sync. through barriers, lock-style methods, atomic operations

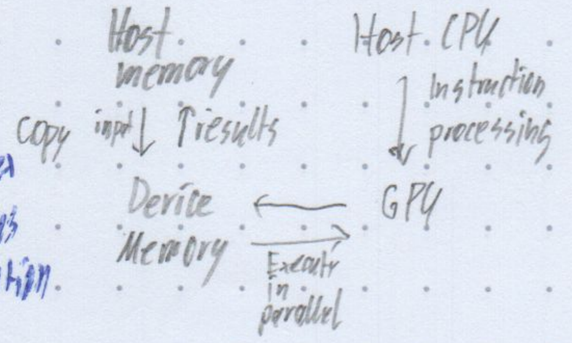


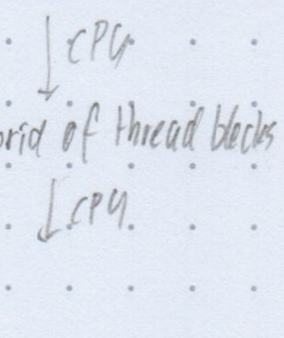

```
#pragma omp parallel for
for(int i=0;i<n;i++){
  z[i]=a*x[i]+y[i];
}
```

GPU

+ efficient
+ scalable
− complex
− laborious optimization

CUDA – GPU computing (with NVIDIA extension for data-parallel workloads






```
__global__ __parallel__
void func(..){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i<n) ...
}
```

needs not full block

```
int nblocks = (n+255)/256; // 256 threads per block
func_parallel<<<nblocks, 256>>>(..);
```

Performance $= \frac{1}{\text{Resources to solution}}$ often goal of parallelism

response, execution time
Throughput
Energy

Peak Performance – guaranteed limit
Actual – limited by critical resource

Speedup $S = \frac{\text{time before enhancement for entire task}}{\text{after}}$

Amdahl's law

Fraction $f_{enhanced}$ = code teil der besser wurde
$S_{enhanced}$ = speedup für it

$$S = \frac{1}{(1-f_{enhanced}) + \frac{f_{enhanced}}{S_{enhanced}}}$$

for parallelism

$$S = \frac{1}{f_{seq} + \frac{1-f_{seq}}{P}} \quad \text{Limit} \quad < \frac{1}{f_{seq}}$$

cost of parallelism
communication demand

$$E(P) = \frac{S(P)}{P}$$

BRUNNEN

if < 1 super-linear speedup
z.B. through aggregated cache

Law of Gustafson – consider problem size

Assumption
sequential part const runtime t_f
parallel part $t_v(n,p)$

$$S(n,p) = \frac{t_f + t_v(n,1)}{t_f + t_v(n,p)}$$

if perfectly parallelized $\lim_{n\to\infty} S(n,p) = P$

$t_v(n,p) = \frac{t_v(n)}{P}$ denke ich

Scalability

Weak: larger problems with more recourses solvable
problem size per processor constant
harder

Strong same problems faster with more recourses
~~harder~~ limited by Amdahls law and $E(p)$

Asymptotic

z.B. dot product

$$T(N,P) = \Theta(N/P + \log(P)) \quad S = \frac{T_1}{T_P} = \frac{\Theta(N)}{\Theta(N/P + \log P)} = \Theta\left(\frac{N}{\frac{N}{P} + \log(P)}\right)$$

$$E(p) = \frac{T_1}{P \cdot T_P} = \Theta\left(\frac{N}{N + P\log(P)}\right)$$

Little's formula

$$C = R \cdot L$$

- L Latency
- desired throughput (items / time)
- # items concurrently in system

Concurrency required to hide Latency

OpenMP - Open specification for Multi Processing

~~shared program~~
shared-memory programming interface
- Compiler directives #pragma Express parallelism
- Library functions get/set parameters, sync
- Environment variables preset parameters

Predecessors Proprietary
developers wanted portability

+ incremental parallelization
+ code size
+ readability
+ supports single CPU
+ shared memory

- limited scale
- special compiler
- implicit communication hard to consider
- danger of incorrect sync.

Syntax
#pragma omp ...

für code
#ifdef _OPENMP
code
#endif

ignored by non openMP compiler

Fork-join execution model
↖ SPMD

Initial thread
Master thread

parallel construct
team of threads - implicit tasks
implicit barrier

loop level
→ Work sharing: gets distributed => every thread runs
Tasking Task pool, ~~threads choose~~
existing team of threads?

Nested parallelism
Offloading computations
SIMD Parallelism
...

Controll structures
can be nested - parallel construct - create team
- work-distribution construct
- task construct - generate task
- target construct - offload to device
- SIMD construct - vector code
- Synchronize thread

Data Environment design
Private
locals private vars

— T1 ...

Sync, order, mutual exclusion

shared
global vars
Heap

Memory model relaxed
Private Memory — T1 — Temporary view of Memory
Registers & cache

Memory

↕ Flush to enforce consistency

Construct ⇔ region
- lexical/static extend
- dynamic ⊆ + sub routines @ called from within

⚠ Variable access may not be atomic

ICVs - Internal Control Variables
control behaviour
init by implementation
assign through environments, routines
API "only way to retrieve"

env	↓ Priority	setenv OMP_NUM_THREADS n
API		omp_set_num_threads(n)
clause	high	numthreads(n)

Loop-level parallelism ← local concept

Worksharing construct in parallel construct

```
#pragma omp parallel   new line
{
  #pragma omp for   clause [,] clause ... newline
     for loop(s) → must complete all iterations
}
```

No break, no outside caught exceptions
continue allowed, exit programm allowed

abgekürzt #pragma omp for
only to immediate loop unless collapse clause

clauses
- if
- num-threads
- copyin - between private threads
- allocate - for private var
- schedule
- collapse - nested loops collapse(#for) - combined iteration space
  z.B. i:j 0:0,0:1, ...
- ordered
- nowait - z.B. für for, single

Data ⚠ pointers

shared (default)
- all vars before entry
- static vars
- Heap allocated memory

private
- local vars
- loop index of work-shared loop
- undefined at begin & end

clauses for memory sharing
default(shared | firstprivate | private | none)
↳ all must be explicitly specified
shared | [first- | last-] private(x, y, z)
- sequential last in masters copy
- init with masters

reduction(operator : varName) z.B. reduction(+:sum) ... sum = sum + ...; ...
↳ +, *, &, |, ^, &&, ||, max, min, - als + (-value)
⚠ should be commutativ and assoziativ

manually achievable by
#pragma omp critical in for loop
z.B. sum = sum + ...;

Data dependence - ~~multiple~~ two accesses at least one writing
≥1 m-access
1-location
≥1-write

race condition - outcome depends on ordering

loop-carried - between iterations

Detection
- automatic (if possible)
  - z.B. PLUTO
- semi-automatic
  - if indizes linear expression
  - solve system
- manual

Classification Action 1 sequentially before 2

A2 \ A1	Read	Write
Read	RAR no dependence	RAW flow dependence ⇒ no parallel possible
Write	WAR anti-dependence only reuse ⇒ ~~make firstprivate~~ 1 read only copy - overhead	WAW output dependence ⇒ lastprivate

Remove techniques
- reduction clause
- Induction-variable elimination
  - if value only dependent on loop index
  - replace with expression z.B. Fibonacci mit Matrizen berechnen
- loop skewing
  - index für teile verschieben (Like Wall Trapezmethode)

Non removable
- alternative algorithm ?
- parallelise different loop
- Fissioning
  - split in parallelizable and non-∥ part
  - z.B. scalar Expansion if non-parallelizable part is only scalar

Parallel overhead => conditional parallelization
(create, distribute, think)
clause z.B. if(n > MINTRIP_COUNT)

loop level parallelisation scheduling - distribution of iterations in chunks ~~good - finish even~~

schedule ([modifier[, modifier]:] kind[, chunk_size])
- modifier
  - monotonic
  - nonmonotonic
  - simd
- kind
  - static
  - dynamic
  - guided
  - auto
  - runtime

static - distribution at loop-entry
- without chunk size
  - #threads ≈ #chunk (nearly equal size)
- with chunk size - assigned in round robin fashion
  ~~assign if thread is free (round robin fashion)~~

Dynamic - threads request new chunks
- Dynamic
  - default chunk size = 1
- Guided
  - implementation specific start size
  - then exponential decrease
  - min. = chunk size

+ Load balance
- overhead

Runtime - via environment var/API call (good for +testing)
Auto (no chunk size) - impl. decides

- complex
+ generic

SPMD-style parallelism - network distribution construct (only ~~single~~ / divided execution)
=> work replication (same)
- omp_get_thread_num() for master -- (team size - 1)
- omp_get_num_threads() = team size

) Domain decomposition
- assignment based on thread num

#pragma omp single [clause's] { work distr. construct ?
structured block
only exec. by 1 thread
- [(first-)private
- copyprivat
- nowait

BRUNNEN

∀ work distr. constructs incl single must be reached by ∀ threads exec in same order
have implicit border except nowait clause
=> single entry/exit
no nesting
no explizit barrier

private clause only construct

get it in region ?
- pass as arguments
- Threadprivat
  - #pragma omp threadprivate (var, ..)
  - z.B. inter var definition
  - once initialized persist (across parallel regions)
  - => same thread number => same var
  - no data-sharing clauses for them but
    - bei for c.: copyin(var) master → workers @entering parallel construct
    - bei single c. copyprivate(var) executing → other @after single construct

orphaned - not in parallel construct
reached from
- serial behaves like 1 master thread ✓
- parallel like inside
  - but different sharing semantics

Synchronization - implicit communication
(of)/through shared memory

does not order - Mutual exclusion ↔ Event - can be used to order

Race condition ↔ Data race
(often solved by private)

Result:

- Race condition: non-deterministic
  - unenforced relative timing of concurrent operations
  - acceptable
    - z.B. round-off errors
- Data race: undefined behavior
  - 2 unorderd conflicting actions
    - min 1 not atomic
  - probably error
    - but z.B. n threads searching first writes 1 → likely works, still bad

Sync mechanisms

Mutual Exclusion z.B. for access to shared data

critical construct

#pragma omp critical [(name)] [hint(hint-expr)]]

- ∠ lock impl., (un-)contended, (non-)speculative
- ⇒ mutual exclusion of same name
  - wait until no other thread inside
- no branches in/out
- no fairness guranteed
- Forward progress garanteed
  - └ eligible thread always gets access
- ⚠ nesting in regions → gurantee same order to prevent deadlock

+ for single construct: multiple update
  - + one sync
  - + severall locations

don't mix ⟷ for conflicting use / + overlap

atomic construct

#pragma omp atomic [clauses]

- expression-statement / structured block
  - ↑ allowed depend on clause; ↑ not covered
- + may use hardware instructions (only numeric/skalar vars)
  - ⇒ performance
- single update

types of clauses → atomic:

- read
- write
- update - default
- extended-atomic
- ...
- memory-order
  - seq-cst ⟸ to comply with
  - release
  - relaxed - default
  - aquire
  - aqu_rel

Deadlock - Threads waiting for event, that can only be caused by one waiting

as if operations interleaved consistent with thread order

if

- No non-sequential consistent atomic
- No relying on false result from lock
- No data races

lock routines through/by runtime libary

- omp_set_lock(<reference to lock>); [-nest]
  - blockiert bis Lock acquisition
- omp_test_lock("); [-nest]
  - returns true if lock aquired else false
- omp_unset_lock("); [-nest]
  - releases it

```
omp_lock_t myLck;          [-nest]
omp_init_lock(&myLck);     [-nest]
...
omp_destroy_lock(&myLck);  [-nest]
```

+ flexibilbillity
  - no block structure
  - dynamic lock
  - compute while waiting with test
  - nestable locks (z.B. for recursion)
    - same thread can set lock multiple times, release

Event synchronization

barrier construct

#pragma omp barrier

- ∀ threads in parallel region must reach barrier before passing

⚠ not inside work-sharing construct

identicle to implicit barrier

taskwait, taskgroup → siehe Tasking

Ordered, Flush → not covered

BRUNNEN

Tasking - idea: seperate problem decomposition, concurency

Task pool ← create tasks — Thread pool
Task pool → fetch tasks → Scheduler → assign to threads

\+ over-decomposed ⇒ load balance
− task overhead if too small (creation, [sync] pool)

explicit ↔ implicit task (makes standard more compact)
explicit: created by task construct
implicit task: generat for ∀ Thread (Task) on begin parallel construct

#pragma omp task [clauses]
structured block

only works in parallel region
undefined order

- untied
- data sharing attributes: default (↳ relative to implicit task) — inherited from creation context → firstprivat created inside → private
- if(condition) (only 1 per task)
- final(condition) (optimization, z.B. for rekursion getting smaller and smaller) — if true → task becomes final, ignore task constructs inside ind. itself
- depend([in|out]inout: var-list) — runtimes enforces dependencies between siblings in order of code (Abhängigkeitsgraph); in/out/inout: read-written by task

barrier inside explicit task → deadlock

#pragma omp taskwait
waits for ∀ direct child tasks
for the grandchildren → use taskwait in children too

#pragma omp taskgroup
{ structured block }
at end waits for all tasks created inside and their descendents

Task scheduling

thread may switch task (can be suspended/resumed)

implied ↔ explicit scheduling points
↳ #pragma omp taskyield
implied: after task generation; last instruction task region; implicit, explicit barriers; taskwait, group constructs

tied ↔ untied task
↳ default: same task resumes suspended task
↳ untied clause: on task gen ↔ any task in team

Parallel Algorithms - faster, larger

energy efficient

$T^*(n)$ = time of best seq. algorithm (known)

$T_p(n)$ = time with p processors

$T_\infty(n)$ = ∞ processors
limited by - seq part
- grain size

$T^*(n) \leq T_1(n)$

$T_p(n) \geq T_\infty(n) \Rightarrow E_p(n) \leq \frac{T_1(n)}{p * T_\infty(n)}$

$\Rightarrow$ drops quickly after $p > \frac{T_1(n)}{T_\infty(n)}$

Speedup $S_p(n) = \frac{T^*(n)}{T_p(n)}$ desired p

$> p \Rightarrow$ superlinear

Parallel Efficiency $E_p(n) = \frac{T_1(n)}{p * T_p(n)}$

≤ 1 without superlinear
↳ may with aggregate cache

Parallel Performance on PRAM

algorithm in $T(n)$ mit $P(n)$ processors
$C(n) = T(n) * P(n)$ - cost of algorithm

$p > P(n) \Rightarrow$ only run $P(n)$
$p < P(n) \Rightarrow O(T(n) \cdot P(n)/p)$

$p = \infty \Rightarrow O(\frac{C(n)}{p} + T(n))$

asymptotically equivalent

$T(n)$ with Landau Symbols

number of operations $\Rightarrow$ needs model

sequential
RAM - Random access machine
+simple language

Input $\Rightarrow$ CPU ⇄ Memory $\Rightarrow$ output

Parallel models simplicity ↔ accuracy
↳general, portabel ↳realistic
$\Rightarrow$ theoric

DAG - Directed acyclic graph
+architecture independent - complex
+data dependence - communication, Allocation missing

= input size
Inputs $\Rightarrow$ nodes without incoming
Operations $\Rightarrow$ inner node = 1 time unit
Outputs $\Rightarrow$ without outgoing arcs

No branches
$\Rightarrow$ e.g. loop unrolling

Algorithm $\Rightarrow$ family of DAGs $\{G_n\}$

Scheduling p processes, shared data
node $i \sim (j_i, t_i)$ are consistend
↳processor ↳time $1_j, \ldots, 1_i$ at a time

$\{(j_i, t_i) \mid i \in N\}$ = schedule
↳set of nodes

execution time = $\max_{i \in N}(t_i)$

$T_p(n) = \min_{\text{schedule}}\{\max_{i \in N}(t_i)\} \geq$ depth of DAG

Shared-memory model - MIMD

C-concurrent
E-exclusive
R-Read
W-Write

for Communication $\Rightarrow$ shared memory
has local memory $\rightarrow P_1 \ldots P_n$
↑ processor ID

variants
- EREW
  no simultan
- CREW
  only read
- CRCW
  - common
    write same value
  - arbitrary
  - priority - min index succeeds

our Modes of Operation
- synchronous ↔ asynchronous
  ↳common clock ↳seperate clock
  auch PRAM ↳parallel
  sync points
  programmer responsible


```
begin
  global_read   i := processNum
  write
  Compute
end
```

Tree $\Rightarrow$ for $h = 1$ to $\log_2 n$ do
if $(i \leq n/2^h)$ then
begin ... end

Network model

- difficult
- topology dependent

asymmetric graph of processors
no shared memory

synchronous ↔ asynchronous
↳ sync via messages
message-passing model II
~~blockierend~~
send(X, j)
blockierend → receive(X, j)
with routing

Topology e.g. Linear Array, Ring, mesh, torus, hypercube, fully connected
evaluation param
diameter - max dist between pair
Maximum degree of a node
Edge or Node connectivity - min removed to disconnect peer
Bisection width - # nodes removed to cut network so roughly in half

#p tradeoff (T_{comp} = computation time
T_{comm} = communication time

$comm(n) = \alpha + n/\beta$ ⌈ latency ⌊ Bandwidth

Work-depth model

DAG, but workers with greedy scheduling

T_p = time to run with p workers
T_1 = work (seq)
T_∞ = depth / critical path / span

no superlinear speedup
more processors dont slow down
depth ~ scalability

Brents theorem

$p \leq$ processors for max concurrency $\Rightarrow \frac{T_1}{p} \leq T_p \leq \frac{(T_1 - T_\infty)}{p} + T_\infty$

perfectly ↔ imperfectly Parallelizable

$\Rightarrow \frac{T_1}{(\frac{T_1}{p} + T_\infty)} \leq S_p \leq \min(p, \frac{T_1}{T_\infty})$

estimate if $T_\infty \ll T_1 \Rightarrow T_p \simeq \frac{T_1}{p} + T_\infty$
↳ decent parallelism

Over-decomposition

Greedy Scheduling if $p \ll \frac{T_1}{T_\infty} : S_p = \frac{T_1}{T_p} \simeq p$

Parallel slack = $\frac{S_\infty}{p} = \frac{T_1}{p * T_\infty}$
↳ excess parallelism

\+ usually tighter than Amdahl's law
\+ lower bounds, notion of imperfect parallelization
\- assumes ∞ memory bandwidth
greedy scheduling (no locks or task pinning)

nur erwähnte Modelle:

BSP - Bulk-synchroneous parallel model
- Processor-/Memory-modules
- Network
- Synchronizer (barrier)

Superstep = compute, send, receive

LogP model - cluster abstraction distributed memory
parameters
- L = latency communication medium
- o = overhead messages
- g = required gap between messages
- P = # processing units

BRUNNEN

GPU - Graphical Processing Unit

applications
graphics, deep learning, ...

1980s VGA - video graphics array controller
Late 1990s + 3D accelerator functions
[Triangle setup, rasterization
[Texture mapping, shading
2000 GPU became own processor
↗ programmable
precise floating point
scalable
support for general-purpose programming languages (C, C++)

CPU ↔ GPU
instruction level parallelism ⟷ ↗ cores, threads (lightweight) - data parallelism
↘ control logic + power efficient
↗ computational intensity
└ arithmetic : memory access operations
caches – hide memory latency
+ to compute
- copy CPU → GPU (and)

Logical graphics pipeline - programmable - physically in Unified processor array
- fixed functions

Input assembler → Vertex shader → Geometry shader → Setup & rasterizer → Pixel shader → Raster operations / output merger

GPU programming interface

Heterogenes system
1 CPU ≥1 GPU
Heterogenes cluster
each processor has ≥ 1 GPU

Vulcan low overhead cross platform (for both)
- Graphics: Open GL, Direct3D
- General Purpose: CUDA, HIP extensions of C++
SYCL; OpenCL C framework
└ programms for heterogenous platforms
OpenACC, OpenMP directive based

CUDA - Compute Unified Device Architecture
C with NVIDIA extensions

slides 4 page 22f.
CUDA software stack, compute capability

Grace Hopper Superchip ↔ GPU as coprocessor to CPU, own DRAM
CPU, GPU can access each others memory transparently

{ CPU serial code
SPMD
Parallelism: GPU parallel kernel
- coarse-grained task: Grid (computes one) ~ (Streaming) multiprocessors (SM) with global, constant (divided into banks) ~~shared~~ memory
  \ partitioned into
- coarse-grained data: Blocks — with shared memory, L1 cache - any order; to sync only atomic memory operations
  \
- fine-grained data: Elements ~ threads executed on SP - Streaming Processore Core — with registers, local memory (└ in global memory ⇒ slow) - managed in Hardware; barriers to sync (only on ∀ of block; ab CUDA 9 cooperative groups)
└ Virtualizes multiprocessors ↔ Problem Decomposition Blocks
⇒ portability

same hardware
cudaDeviceSetCacheConfig(), cudaFuncSetCacheConfig()
default
size of shared known at compile time
splits automatic
dynamic
third param of kearnel call

/ independent - concurrently given sufficient hardware
\ dependent - sequential with implicit inter-kernel barrier

CUDA variable type qualifiers

Declaration	Scope	Memory	Lifetime
automatic scalar vars	Thread	Register	Kernel
" array vars	"	Local	"
__shared__ type name;	Block	Shared	"
__device__	Grid	Global	Application
__constant__	"	Constant	"

BRUNNEN

combines SIMD performance & Multithreading productivity

SIMT in SM

partitioned after linearized threadIdx
last warp padded - results nullified

SM manages warps - groups of 32 parallel threads (↳ do same / nothing)
↳ scheduled like threads in normal processes
many for latency hiding
waiting kept in register files
⇒ fast

different
⇒ data dependent conditional branching in warp (e.g. diff #iterations in for loop) ⇒ serializes - performance

Compilation workflow (NVCC - compiler)

need same word length (32/64 bit)

Host Code
replacing CUDA syntax ↦ runtime function calls
compiled (by another tool)

Device Code compiled to: — only subset of C++
- cubin object (binary code)
  - architecture specific
  - compatibility between minor revisions
- PTX - Parallel Thread Execution (scalar instructions) ← stable target Instruction Set Architecture for many compilers (incl. Direct 3D vector instructions)
  - compiled just in time by device driver (and cached)
  - - load time
  - + aufwärtskompatible (benefit from new optimizations)

CUDA programming interface (C++ extension)

implicit initialization for runtime @ first runtime call
- CUDA context f. devices on system
- (just-in-time compilation device code)

```
cudaError_t cudaMalloc(void **devPointer, size_t size)
```
↳ returns pointer to location in M. ↳ in bytes; sizeof(type) · number

```
cudaError_t cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
```
dst, src don't overlap
↳ cudaMemcpyHostToHost
" HostToDevice
" DeviceToHost
" DeviceToDevice

```
cudaError_t cudaFree(void *devPointer)
```

Error handling

```
const char* cudaGetErrorString(cudaError_t error)
```

Extension to function declaration (before return type)
- kombinierbar: __device__ (nur von entsprechendem Aufrufbar)
- default → __host__
- __global__

Built-in vector types of basic integer, floating types
- type1 (fields x)
- 2 (x, y)
- 3
- 4 (x, y, z, w)

constructor: make_type1/2/3/4(type x, ...)
↳ nicht angegebene 1

dim3 - integer vector for dimensions

Kernel call (dim3 als var oder initialized mit {...} übergeben)

```
name <<< gridDim, blockDim >>> ();
```

Thread organization

vars for device code ⇒ global coordinates
- dim3 gridDim
- uint3 blockIdx — x = blockIdx * blockDim.x + threadIdx.x; y = ...; z = ...
- dim3 blockDim
- uint3 threadIdx
- int warpSize

BRUNNEN

concurrently to each other

Streams of GPU instructions
- execute in issue-order
- 1 Default Stream

Sync with CPU
- asynchronous / non-blocking
  - kernel launch
  - asychronous cuda calls (allow overlapping kernel, memcopy)
- synchronous / blocking
  - memcopies (except async ones, within same device, ≤ 64kB) → H2D
  - cudaDeviceSynchronize()
    - blocks until completion of cuda calls

Sync threads in block
- Barrier
  - __syncthreads();

---

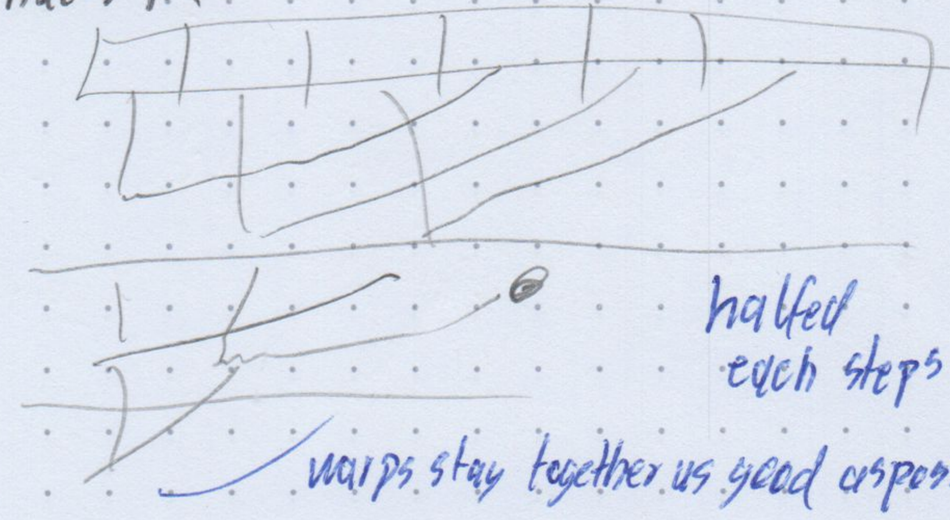
max threads pro block

more ⇒ multiple blocks $\lceil \frac{N}{\text{threads pro blk}} \rceil$ Randbehandlung, Randthreads tun nichts

may be usefull to collaborative
load data in shared, syncthreads, then work
z.B. Matrix multiplication parts required multiple in block) change order, phases plan multiple ~~stages~~ to reduce memory
= Locality

Reduction Algorithm / Tournament Reduction – "divide and conquer"

```
int t = threadIdx.x;
for(unsigned int stride = blockDim.x >> 1; stride > 0; stride >>= 1){
  __syncthreads();
  if (t < stride)
    array[t] = op(array[t], array[t+stride]);
}
```



Further performance aspects
- threadblocks to warps
- coalescing memory access
- prevent bank conflicts
- prefetching data
- balance per thread resources
- unroll loops ⇒ reduce cost of loop control
  - typically compiler thing
- thread granularity

BRUNNEN

Advanced CUDA features
- Nvidia Nsight
  - performance view
- GPUDirect multiple
  - copy data between GPUs
  - even remote
- Dynamic parallelism
  - kernel launching kernel
  - recursive, data dependent parallism
- Unified memory CPU, GPU
  - runtime handels it
- Cooperative groups of threads to sync, ..
  - kind: block, tile, coalesced, multi-grid, grid
- CUDA graphs of GPU operation
  - launched as one from CPU
- Tensor cores
  - specialized hardware execution units
  - z.B. matrix, convolution operations

Parallel Patterns

- name
- context
- forces (goals & constraints)
- solution

pattern language
to solve problem only choose one ~~Pfeil~~ pattern after another
=> pattern web

1. Finding concurrency

Decomposition patterns

Task (functional)
break stream of instructions
⚠ not to fine granular or irregular
group based on memory
Cutoff point

Data
= same task

Forces:
Efficiency
Simplicity ↔ Flexibility

Dependence analysis
graph
group tasks
sharing constraints
order tasks
temporal/data dependencies
Data sharing
read only, accumulate
n read 1 write, effective local

Design Evaluation
good enough?
target platform?
Forces?

=> concurren tasks with local duty
grouping, dependencies

2. Algorithm structure

16.1.24

Rest nicht testat relevant

BRUNNEN