Parallel Programming

Performance Metrics

Performance Speedup: $speedup(p_n) = \frac{performance(p_n)}{performance(p_1)}$ (n processors / 1 processor)

In HPC (high perf. computing) $performance = \frac{work}{time}$

$$speedup(p_n) = \frac{time(p_1)}{time(p_n)}$$

Speedup based on Throughput:

$$speedup\_t(p_n) = \frac{tpm(p_n)}{tpm(p_1)}$$ — transactions per minute

Parallel efficiency: $efficiency(p_n) = \frac{speedup(p_n)}{n}$

Amdahl's law

Execution time (assuming parallel regions have full speed):

$$T(p) = (1-f) \cdot T + \frac{f \cdot T}{p}$$

f - fraction of parallel execut.
p - n of parallel tasks/threads/processes
T - sequential time

$25\% = 0{,}25$

$(1-0{,}25) \cdot T$ — сколько выч. seq.

$\frac{f \cdot T}{p}$ — ост. - паралл., максим.

We assume that state (resources) stays the same

↓

Maximal speedup

$$SU(p) = \frac{T}{T(p)} = \frac{T}{(1-f)\cdot T + \frac{f \cdot T}{p}} = \frac{1}{1-f+\frac{f}{p}}$$

Superlineal speedup - more than linear speedup, can happen when with num. of processes resources increase

Types of parallelism

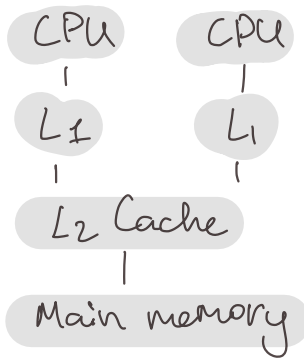
- **Data parallelism**
  Делим данные на кусочки, выполняем одни и те же операции на всех кусочках.
- **Functional parallelism**
  Делим все на таски, таски могут выполнять разные задачи

Flynn's Classification

	Single Data	Multiple Data
Single Instruction	SISD	SIMD - single instr. mult. data
Multiple Instruction	MISD	MIMD

Shared Memory Models

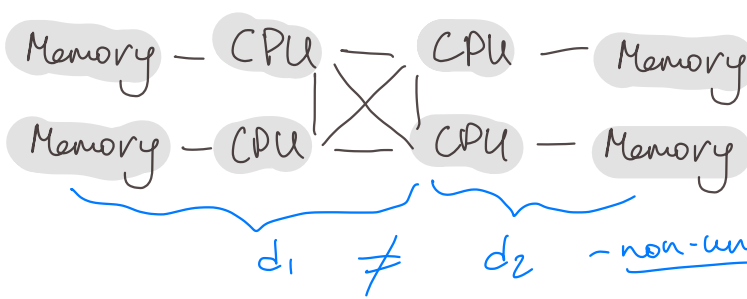
- **Uniform Access Memory (UMA)** (also symmetric multiprocessors - SMP)



Одинаково расстояние от всех CPU до памяти, поэтому uniform

- **Non-uniform Memory Access (NUMA)**

(also Distributed Shared Memory Systems - HW-DSM)



$d_1 \neq d_2$ - non-uniform

d_2 - much faster

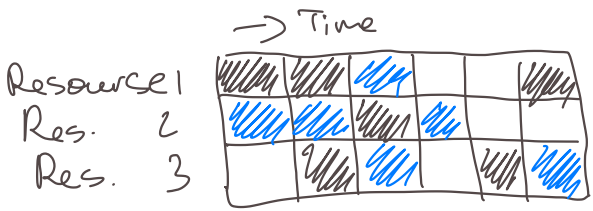
Cache coherent!

Thread is a stream of execution

Software threads are abstracting execution streams

↙ user-level ↘ system-level

Hyper-Threading (SMT)



Thread 1
Thread 2

POSIX Threads

```
int pthread_create(*thread, *attr, *start_routine, *arg)
```

ret. thread id

properties of thread behaviour (bound/unbound, scheduling policy...)

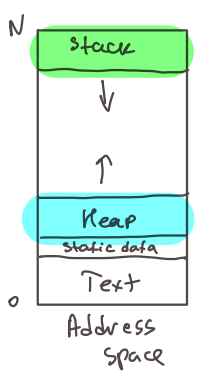
```
int pthread_join(pthread_t thread, void **retval)
```

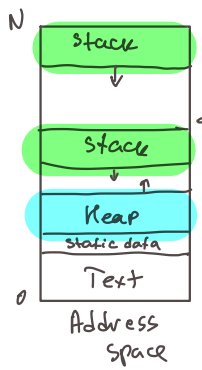
id

return value

Нужно (можно) set&get stack size & stack addresses

Stack Structure





X - needs to be determined

Process - a running instance of a program with it's own memory address space

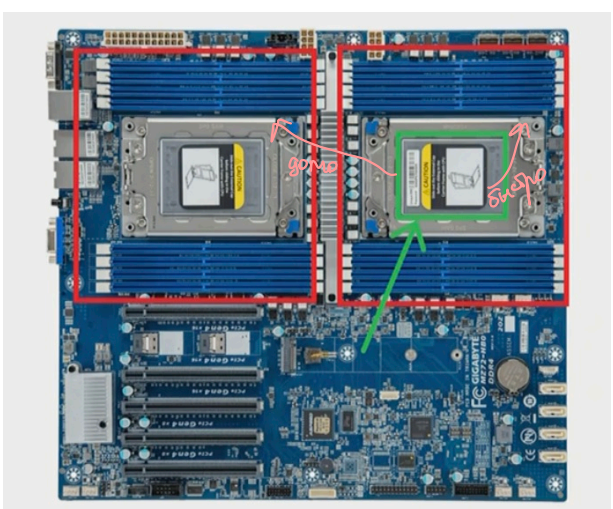
Core - an independent processing unit

Thread - independent stream of execution

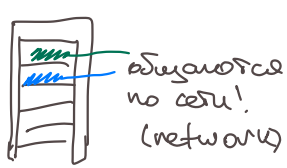
Hardware thread - implementation of an execution stream in hardware

Software thread - programming abstraction that represents a stream of execution

Node (multiple per rack)



Rack



NUMA Socket
(CPU + memory)

Physical CPU

Threads creation

```
...
// Start numThreads threads
std::thread threads[numThreads];
for (int i = 0 ; i< numThreads ; ++i){
    threads[i] = std::thread(oops_kernel, &i);
}
for (int i = 0 ; i< numThreads ; ++i){
    threads[i].join();
}
...
```

```
void oops_kernel (int *args) {
    int argument = *args;
    ...
}
```

Этот цикл может завершиться раньше, чем тред запустится, и i станет out of scope

Лучше делать так

```
...
// Start numThreads threads
int ids[numThreads];
std::thread threads[numThreads];
for (int i = 0 ; i< numThreads ; ++i){
    ids[i] = i;
    threads[i] = std::thread(a_kernel, &ids[i]);
}
for (int i = 0 ; i< numThreads ; ++i){
    threads[i].join();
}
...
```

```
void a_kernel (int* args) {
    int argument = *args;
    ...
}
```

Lock Implementations

Критерии:

1. Correctness : Guarantees mutual exclusion
2. Fairness: Every process eventually gets the mutex

Implementations:

- **Spin Lock**
  Spin on a flag, if lock is taken
  Typicly via atomic operations

  Advantage: fast response time

  Disadvantage: • blocks the hardware threads, uses resources, costs energy
  • generally bad for concurrent execution (если хотим использ. тред для чего-то еще)
  (касается, имеется в виду прочно concurrent, не parral)

- **Yielding Locks**
  If lock is taken - do nothing, give back hardware thread. Usually implemented via runtime system.

  Advantage: low resource usage

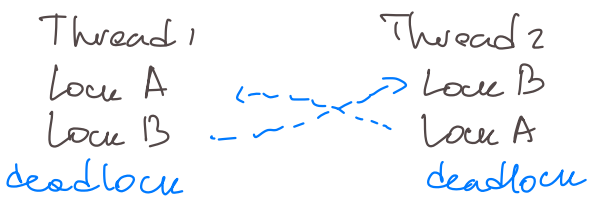
  Disadvantage: • slow response time
  • generally bad for HPC (High performance computing)

Lock Granularity

+: Locum не все переменные, а только часть (данных)
- Maximizes parallelism

-: requires multiple locks management

Danger: Deadlocks



Deadlock avoidance Strategies

- only hold one lock at a time - bad :(
- have a locks manager, ask permission to lock
- order the locks - сначала A, потом взяла B

Threading Performance

Overheads:
- Thread creation & destruction can be expencive

That's why we have threadpools - create once in the beg.

Locks:

– low overhead

– large overhead

↖ more threads accessing more often

Choose wisely between locking all data under one lock VS lock granularity

Thread pinning:

Иногда хотим "запинить" тред на одном ядре, чтобы overhead на cache move ушел

OpenMP

MP - multiprocessing

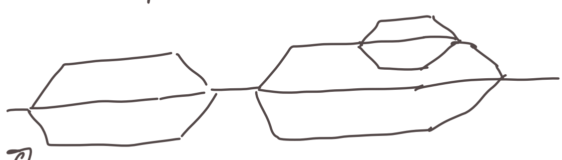
все равно fork-join под ковром

+ использует pthreads

```
export OMP_NUM_THREADS=2
gcc -O3 -fopenmp openmp.c
```

```
#include <omp.h>

...
#pragma omp parallel
{
        ......

}
```

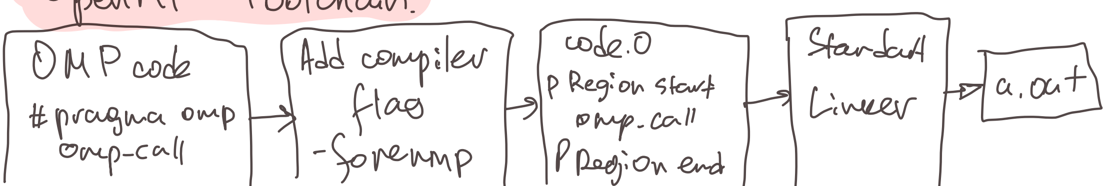
Nested parallelism:



Но не все вендоры его поддерживают

На самом деле под ковром оно создаёт max_n тредов, но исп. 3. Потом не удаляет, а из ОС переиспользует.

OpenMP Toolchain:



Нигде нельзя исп. имена, начин. с omp_...

↗ namespace omp

```
main(){
int a[1000], b[1000];

#pragma omp parallel sections
 {


  #pragma omp section
  for (int i=0; i<1000; i++)
    a[i] = 100;
  #pragma omp section
  for (int i=0; i<1000; i++)
    b[i] = 200;

 }
}
```

} выполняются параллельно, каждый одним потоком

```
main (){
int a[100];
#pragma omp parallel
 {
   #pragma omp for
    for (int i= 1; i<n;i++)
       a[i] = i;
 }
}
```

} should be very simple + independent iteration, т.е. order is not guaranted

here guaranteed the sync (unless) #pragma omp for nowait

Можно настроить разбиение в цикле:

`#pragma omp for schedule(static)`

нумерация с 0

`omp_get_thread_num()`

- **static**
  Fixed sized chunks, distributed in round-robin fashion
  (default = num_iter / num_thr) ← по зависит от имплементации! Не запоминаться
  lower overhead
- **dynamic**
  Fixed size chunks (default = 1), distr. one by one at runtime as chunks finished (один закончил, пошел новый)
  schedule(dynamic, 4)
- **guided**
  Start with large chunk size, then exponent. decrease
- **runtime**
  Controlled at runtime using control variable
- **auto**
  Compiler/runtime can choose

This are preferred

Shared data

По умолч. все глобальное.

Чтобы сделать локальное:

```
int t;
#pragma omp parallel private(t)
```

можно написать shared(t)
или default(private) → shared/none

Но! Это создает новые переменные, не копии!

Т.е.

```
int t=3;

# .... private(t)
{
   t=t+1;
}
```

private(t) - создаст переменную с именем t для каждого потока, но прошлым мусором

← содержит мусор! + создается, кажется, один раз на один тред

firstprivate(t) - сделает копию t до первой итерации для всех тредов

lastprivate(t) - делает новые переменные для t, но в t=t+1 запишется результат последнего треда

reduction(op,t) - инициализ. t нулем, в конце каждого треда берет значение t и применяет оператор op.

- Reductions Операции:
  - Aggregate results of region to master thread
  - reduction(operator: list)
    - operator: +, *, -, &, ^, |, &&, ||

инициализ. 0 → инициализ. 1

```
int a=0;
#pragma omp parallel for reduction(+: a)
for (int j=0; j<4; j++)
{
  a = j;
}
printf("Final Value of a=%d\n", a);
```

инициализируется нулем, потому что + своя копия

← но у каждого треда

OMP_NUM_THREADS=4

Final value of a:

Складываются финальное значение а каждого треда

Если итераций 4, а треда 2, то будет сумма всего 2х чисел!

T_1 T_2

1 2

(3) + (4)

Лучше делать так:

```
int a=0;
#pragma omp parallel for reduction(+: a)
for (int j=0; j<100; j++)
{
  a += j;
}
printf("Final Value of a=%d\n", a);
```

CV кушается в гонке за race!

guard_lock {error} - unlocks!

просто лок - { lock() error } - doesn't unlock!

```
int data=0;
#pragma omp parallel for
for (int i=0; i<2; i++){
    data++;
}
```

race condition

результат неизвестен, может быть 1, может быть 2

- `int data = 0` — копирует data = 0 для всех тредов

```
#pragma omp parallel for firstprivate(data) lastprivate(data)
for (int i=0; i<2; i++) {
    data++;
}
```

lastprivate(data) — записывает в data последнее значение из каждого треда

`output(data)` → data = 1

Почему лучше всего писать `reduction(+, data)`

- `#pragma omp critical { ... }` — почти как мьютекс

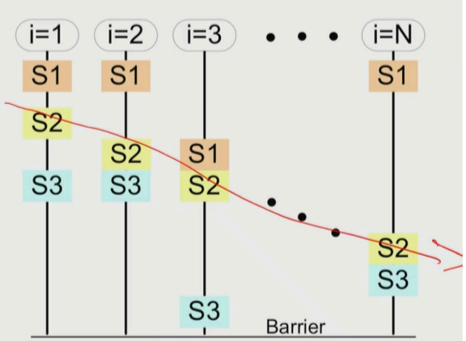
```
#pragma omp parallel - создаёт треды
{
  #pragma omp for - параллелит
  {} - тут барьер
}
```

  - omp barrier - explicit "join", можно пропустить с nowait
  - omp master - старая версия, новая - masked(threads_ids) - то же, но исп. любой тред
    - Only master thread executes code (within parallel region)
    - Others skip and continue
  - omp single
    - Only one [redacted] thread executes code — тот, который пришел первее
    - Barrier at end

- **omp critical [Name]** – просто выполняет по одному
  - can only be executed by one thread at a time
  - block executed sequentially
- **atomic**
  - memory updated atomically – как критикал, но только простые операции a++, a--
  - only simple updates
    - increment/decrement, assignment
- **omp_lock_to object**
  - same as pthread mutex
  - init/destroy, set/unset, test
- **omp ordered** – конкретно этот кусок гарантированно выполнен в очередности
  - Define block within parallel region
  - Block executed in same order as if sequential
- **section** – выел. только одним рандомн. тредом

omp_init_lock(&lockvar)	initialize a lock
omp_destroy_lock(&lockvar)	destroy a lock
omp_set_lock(&lockvar)	set lock
omp_unset_lock(&lockvar)	free lock
logicalvar = omp_test_lock(&lockvar)	check lock and possibly set lock

```
#pragma omp for ordered
for (...)
  S1
  #pragma omp ordered
  { S2 }
  S3
 }
```



all implicit barriers:

- single
- for
- parallel
- sections
- workshare

Data Races

- Two unsynchronized access, and one of them is write
- Hard to detect
  - Debugging changes timing
  - Use tools (but they make code slow)

- Dependencies – guarantee program correctness
  - Control dependence
    - Need to synchronize control flow
  - Data dependence
    - Need to synchronize data
  - Types:
    - Flow dependence: write -> read
    - Anti dependence: read -> write
    - Output dependence: write -> write

Проблемы могут так же возникнуть в алгоритмах:

b = &a

True Dependence Flow Dependence

$O(S1) \cap I(S2) \neq \emptyset$

output input

$(S1\ \delta\ S2)$

The output of S1 overlaps with the input of S2

```
S1: X = ...
        ...
S2: ... = X
```

Anti Dependence

$I(S1) \cap O(S2) \neq \emptyset$

$(S1\ \delta^{-1}\ S2)$

The input of S1 overlaps with the output of S2

```
S1: ... = X
        ...
S2: X = ...
```

Output Dependence

$O(S1) \cap O(S2) \neq \emptyset$

$(S1\ \delta^{o}\ S2)$

The output of S1 and S2 write to same location

```
X = ...
X = ...
```

Loop Independent Dependencies

All dependencies are with iterations
No dependencies across iterations

Example:

```
for (i = 0; i < 4; i++)
   S1: b[i] = 8;
   S2: a[i] = b[i] + 10;
```

Loop Carried Dependencies

Dependencies across iterations
Computation in one iteration depends on data written in another

Example:

```
for (i = 0; i < 4; i++)
   S1: b[i] = 8;
   S2: a[i] = b[i-1] + 10;
```

Loops transformations

- Transformations:
  - Loop interchange (nested loops)
    - Interchange is safe if:
      - dependencies are within same iteration (i=i)
      - or depends on previous steps (i+1=i)
    - Move the chunkier loop outside & parallelize outer loop

for 1
for 2 => for 2
for 1

- Loop distribution/fission
  - If loop has multiple statements, one has inter-iteration dependencies, statements have no dependence cycle
  - Split statements into two separate loops
  - Increases granularity, useful if it makes serial loop parallelizable

```
do j=2,n
S1:  a(j)= b(j)+2
S2:  c(j)= a(j-1) * 2
enddo
```

```
do j=2,n
S1:  a(j)= b(j)+2
enddo

do j=2,n
S2:  c(j)= a(j-1) * 2
enddo
```

- Loop fusion
  - Combine loops
  - Can introduce loop carried dependencies & reduce parallelism
  - Increases granularity, reduce thread overhead (?)

for
for } => for

- Loop alignment
  - Shift loop carried dependencies -> independence
  - Peel off first and last iteration

```
do i=2,n
S1:  a(i)= b(i)+2
S2:  c(i)= a(i-1) * 2
enddo
```

```
do i=1,n
S1:  if (i>1) a(i)= b(i)+2
S2:  if (i<n) c(i+1)= a(i) * 2
enddo
```

  - Can eliminate loop carried dependencies if:
    - Loop has no dependence cycle
    - Distance of dependence is constant throughout iterations

Memory Models

Memory model - describes the interactions of threads through memory and their shared use of data

- **Memory/Cache Coherency**: reasoning about updates to one memory location
- **Memory Consistency**: reasoning about updates to several memory locations

Sequential Consistency – A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program

Processor consistency — Ensures writes from a single processor are seen in order by all processors. - yme Hardware Thread consistency

No global order (given)

Requires cache coherency. Writes issued by a single processor are seen by all processors in the order they were issued.

Weak consistency — Guarantees memory consistency only at synchronization points.

Release consistency — Ensures memory consistency at acquire and release. (Same as weak but with more relaxed)

Memory synchronization points (or flush points)

- Entry and exit of parallel regions
- Barriers (implicit and explicit)
- Entry and exit of critical regions
- Use of OpenMP runtime locks
- Every task scheduling point (see Segment 19)

However, note the following are **NOT synchronization points:**

- Entry and exit of work sharing regions
- Entry and exit of masked regions

generic

Sequential Consistency:

1. Program order

Within a single thread, memory operations (reads and writes) must appear to be executed in the order specified by the program.

T.e. read x
write t
⇕ should not be shifted!
А так иногда случается из-за оптимизаций компилятора и из-за задержек (в сети, например)

write atomicity

2. Atomicity

Memory operations must be atomic.

3. Visibility

A write by one processor must be visible to all processors.

← если один тред что-то записал, то все последующие треды должны прочитать именно эту версию, а не старую

4. Total order

All memory operations must be globaly ordered.
Т.е. можно составить последовательность записей и чтения среди всех тредов, причем так, чтобы не нарушался программный порядок внутри треда

Weak Consistency:

Two categories of memory operations:

- **Synchronization Operations**

base scheduler, entries and exits of parallel regions, barriers, etc. Cannot be reordered.

All preceding instructions must be completed before the execution of a synchronization operation.

Accesses to synchronisation instructions are sequentially consistent

Requires explicit sync. operations to ensure that memory operations are consistent and visible.

- **Data Operations** (memory instructions)
  Just reads and writes. Can be reordered

Release Consistency:

Same as weak but more relaxed
Instead of synchronization barrier, provides 2 barriers – Acquire and Release operations.

Acquire ↗ lock
Release ↗ unlock

Acquire doesn't have to wait for the preceding instructions!
Release doesn't have to be completed before the following instructions
(But if there are instructions that need to wait until Release completes, we need to Acquire before them)

Why is the program order requirement hard?

- Network can reorder
- Use of write-buffers
- Compiler optimizations

Why is the in-sequence/atomicity requirement hard?

- A write has to take place instantly with respect to all processors
- Can lead to a system-wide serialization of writes

We need to relax consistency requirements

- Necessary to achieve well performing systems
  - Increase the level of concurrency
- Compensate with synchronization primitives
  - Need them anyway for proper synchronization of contents
  - Can implicitly execute consistency operations

Create the "illusion" of sequentially consistency

Manual Synchronisation

Thread 1	Thread 2
a = foo();	while(flag);
flag = 1;	b = a;

2 separate variables, compiler doesn't know they belong

What can go wrong?

- Compiler can reorder memory accesses
- Compiler can keep variables in registers

#pragma omp flush [list of variables]

Thread 1

```
a = foo();
#pragma omp flush(a,flag)
flag = 1;
#pragma omp flush (flag)
```

чтобы не поменялись местами

Thread 2

```
while (flag)
{
  #pragma omp flush (flag)
}
#pragma omp flush(a,flag)
b = a;
```

flag gets updated

ensures there is no reordering

Explicit Tasking

Tied Tasks – once started, will remain on the same thread.
Default behaviour, easy to reason about

Untied Tasks – can move to a different thread. Execution can be interrupted and task moved. More flexibility for runtime system and better resource utilization che)
(but you can loose your ca

```
#pragma omp parallel {
  #pragma omp single {
    for (elem = l->first; elem; elem = elem->next)
      #pragma omp task
        process(elem)
  } ← barrier
}
```

создает таски только один поток

по умолчанию их все потоки! Оптимизирование

by default - tied tasks

```cpp
#include <iostream>
#include <omp.h>

int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp task
        std::cout << "Hello World from task"
                  << std::endl;
    }
    return 0;
}
```

создаются 4 треда
создаются 4 таски, каждый тред создаёт таску

```cpp
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp task
        {
            #pragma omp critical
            std::cout << "Hello World from task, "
                      << "executed by thread: "
                      << omp_get_thread_num()
                      << std::endl;
        }
    }
    return 0;
}
```

все треды создают таски, но по очереди (sequentially)

```cpp
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int t = 0; t < omp_get_num_threads(); t++)
            {
                #pragma omp task
                {
                    #pragma omp critical
                    std::cout << "Hello World from task, "
                              << "executed by thread: "
                              << omp_get_thread_num()
                              << std::endl;
                }
            }
        }
    }
    return 0;
}
```

один тред создаёт таски

тут они создаются и берутся другими тредами сразу же после создания

#pragma omp task [clause list]

Select clauses:

if (scalar-expression) ← если False, то это не будет task, и блок сразу не начнет выполняться (например if n>10, пусть будет разбито на таски)

untied – чтобы не привязать к нит-предку

Default (shared|none), private, firstprivate, shared
default is firstprivate! ← внутри таска

priority (value)
execute this task before another task
Must NOT be used to rely on task ordering

Example: Tree Traversal

```
struct node {
  struct node *left;
  struct node *right;
};

void traverse( struct node *p ) {
  if (p->left)
    #pragma omp task // p is firstprivate by default
    traverse(p->left);
  if (p->right)
    #pragma omp task // p is firstprivate by default
    traverse(p->right);
  process(p);
}

#pragma omp parallel
#pragma omp single
   traverse(root);
```

#pragma omp taskwait - чтобы проходились по порядку

`#pragma omp taskwait` - ждет completion выполнения всех детей (i.e. tasks, created since the beginning of the current parallel scope)

`#pragma omp taskyield` - specifies that the current task can be suspended
This is an explicit task scheduling point

Implicit task scheduling points:
- task creation
- end of task
- taskwait
- barrier sync

Tasks dependencies:

```
#pragma omp task shared(x) depend(out:x)
        task1();
#pragma omp task shared(x) depend(in:x)
        task2();
```

← оно само поймет, в каком порядке выполнить tasks

Considerations:
- task granularity
- NUMA optimization

```c
int fib(int n) {
    int i, j;

    if (n < 2) return n;

    #pragma omp task shared(i) firstprivate(n)
    i = fib(n - 1);

    #pragma omp task shared(j) firstprivate(n)
    j = fib(n - 2);

    #pragma omp taskwait
    return i + j;
}
```

```c
int main(int argc, char** argv) {
    int n = 30;

    if(argc > 1)
        n= atoi(argv[1]);

    omp_set_num_threads(4);

    #pragma omp parallel shared(n)
    {
        #pragma omp single
        printf("fib(%d) = %d\n", n, fib(n))
    }
}
```

Orphaned task uses firstprivate by default
Non-orphaned task inherit variables as shared

doesn't have parent, first task that was created
task invoked within a task

```c
#define T 30 // THRESHOLD

int fib(int n)
{
    int i, j;

    if (n < 2)
        return n;

    #pragma omp task shared(i) firstprivate(n) final(n < T)
    i = fib(n - 1);

    #pragma omp task shared(j) firstprivate(n) final(n < T)
    j = fib(n - 2);

    #pragma omp taskwait
    return i + j;
}
```

говорит, что больше не будем создавать task-ов, а будем запускать рекурсию внутри этой таски

omp_set_nested(1); – не исп. для tasks! Только parallel
– если хотим создавать nested parallelism – parallel region inside parallel region

#pragma omp parallel num_threads(2)
{
  #pragma omp parallel num_threads(2) ← создаст по 2 треда внутри каждого треда
    { ... }
}

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a // Shared
              // a= 1
            b // Firstprivate
              // b= ?
            c // Shared
              // c= 3
            d // Firstprivate
              // d= 4
            e // Private
              // e= 5
        }
    }
}
```

2) private - создаем в скоупе новую
← по умолч. firstprivate для всех перемен., объявл. внутри parallel скоупа
Shared - объявили данную
Firstprivate - создали в скоупе новую
Shared - объявили до скоупа
Firstprivate – объявили в скоупе
Private - объявили внутри task

Important Runtime Routines

Determine the number of threads for parallel regions
- `omp_set_num_threads(count)`

Query the maximum number of threads for team creation
- `numthreads = omp_get_max_threads()`

Query number of threads in the current team
- `numthreads = omp_get_num_threads()`

Query own thread number (0..n-1)
- `iam = omp_get_thread_num()`

Query the number of processors
- `numprocs = omp_get_num_procs()`

internal control variables

Relevant Environment Variables (ICVs):

`OMP_NUM_THREADS=4`
- Number of threads in a team of a parallel region

`OMP_SCHEDULE="dynamic" OMP_SCHEDULE="GUIDED,4"`
- Selects scheduling strategy to be applied at runtime
- Schedule clause in the code takes precedence

`OMP_DYNAMIC=TRUE`
- Allow runtime system to determine the number of threads

`OMP_NESTED=TRUE`
- Allow nesting of parallel regions
- If supported by the runtime

Runtime routines to query and set ICVs

Typical performance considerations

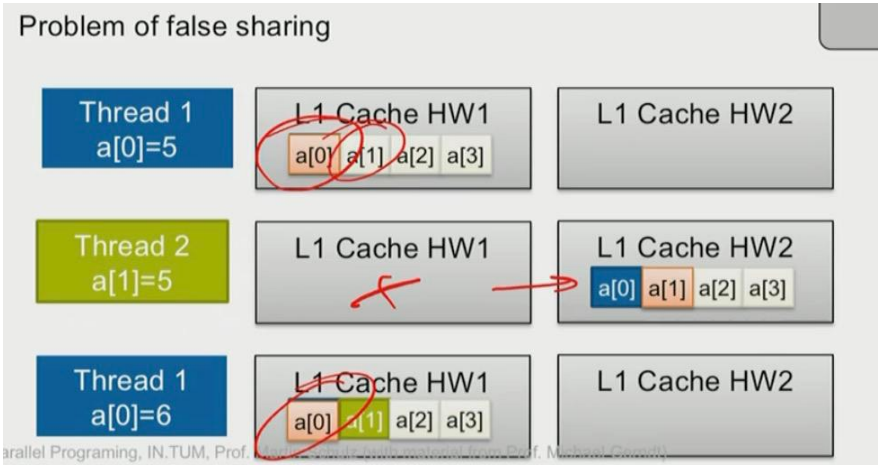
- Threading overhead
  - Expencive to starting/ending a thread - better do once (as omp does)
  - Scheduling is also not that cheap - better not to have too little work per thread
- Syncronisation overhead
  - Lock too often
  - Lock too much (lots of waiting)
- Cache behaviour and locality
  - Have to keep cache to reuse
  - Locality - better take from nearest cache
- Thread mapping
  - How fast to put thread from memory

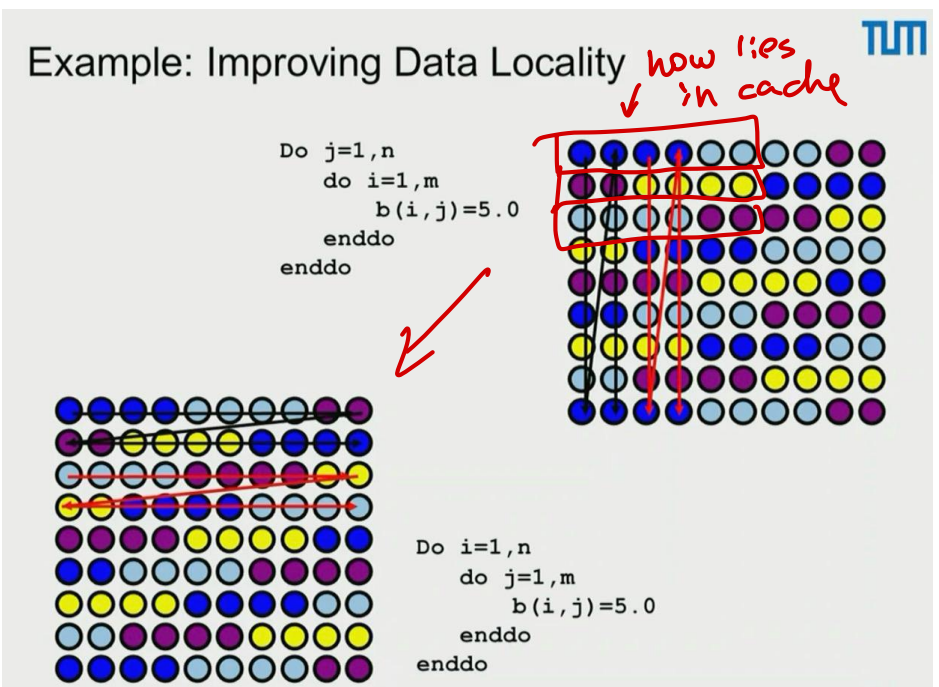
another solution

Compare_and_swap ← atomic operation - in loop

ABA Problem

- A tries to remove an element
  - Has to wait
- B removes the element
- Frees it
- Reallocates it (new element, same address)
- B inserts element again
- A compares against the old/new element


Problem of false sharing
Thread 1
a[0]=5
L1 Cache HW1
a[0] a[1] a[2] a[3]
L1 Cache HW2
Thread 2
a[1]=5
L1 Cache HW1
L1 Cache HW2
a[0] a[1] a[2] a[3]
Thread 1
a[0]=6
L1 Cache HW1
a[0] a[1] a[2] a[3]
L1 Cache HW2

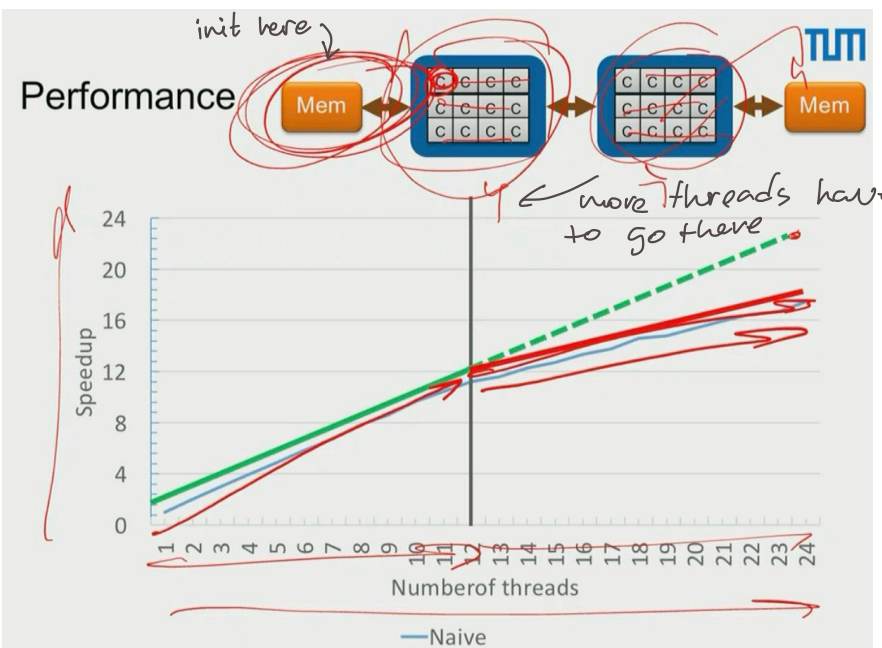

Example: Improving Data Locality
how lies in cache
Do j=1,n
do i=1,m
b(i,j)=5.0
enddo
enddo
Do i=1,n
do j=1,m
b(i,j)=5.0
enddo
enddo

Placing Memory

First touch:

- Allocation creates virtual space
- First access allocate physical memory located close to accessing thread
- No later migration

That's why this happens:



Control by ICV **OMP_PROC_BIND**

- **true:** threads are bound to cores/hw-threads
- **false:** threads are not bound and can be migrated
- **master:** new threads are always co-located with master
- **close:** new threads are located "close" to their master threads
- **spread:** new threads are spread out as much as possible

Description of available locations, also known as places: **OMP_PLACES** ICV

- Describes ordered list and hierarchy of all available hardware threads
- Example: `{0:4},{4:4},{8:4},{12:4}`
- Master thread executes on first place

SIMD

To support SIMD, processor needs special vector registers and instruction set support

SSE - streaming SIMD extentions - 128 bit registers available on all x64_86 intel processors

AVX - SSE+SSE = 256 bit registers

<immintrin.h> - intrinsics define datatypes and operations with them

"_mm256_xxxx_ps"

AVX gcc

AVX datatypes:

__m256 - 8 x 32bit float
__m256d - 4 x 64 bit double
__m256i - 8x32 bit int or 4x64bit int

__mm256_loadu_ps
- u: unaligned
- ps: packed single — т.е. операция на всем векторе, не на части

__mm_add_ps - add packed floating point nums

__mm_movelh_ps - byte operations

_sd - scalar operations on the least significant data element (0-31 bits for floats)

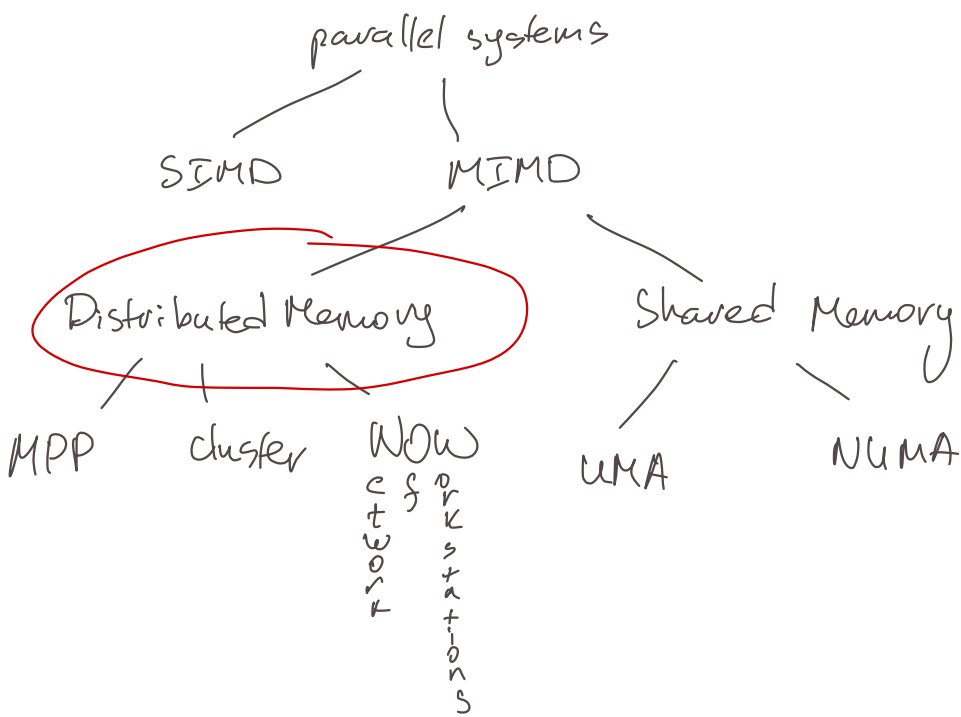
Things to consider:
- Latency - number of clock cycles
- Throughput - rate of executions of instructions

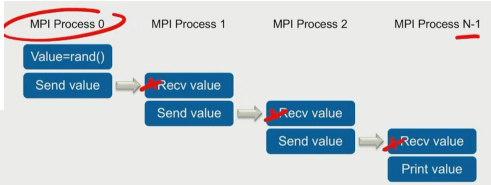
MPI: Message Passing Interface

aiming 10 microsec latency max ← network transportation

Types of operations:

Point to Point | Collectives | One Sided






```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main (int argc, char** argv)
{
  double value;
  int size, rank;
  MPI_Status s;
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  value=MPI_Wtime();
  printf("MPI Process %d of %d (value=%f)\n", rank, size, value);
  if (rank>0)
    MPI_Recv(&value, 1, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, &s);
  if (rank<size-1)
    MPI_Send(&value, 1, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD);
  if (rank==size-1)
    printf("Value from MPI Process 0: %f\n",value);
  MPI_Finalize ();
}
```

Should be called by all MPI processes! exactly once!

how many proc. in my world

init the job. Передаем параметры тут сами аргументы strips MPI args from argc, argv

returns error code

my number

from

what to send how many type where to send tag

context

shut down MPI - resources released

Parallel Programing, IN.TUM, Prof. Martin Schulz (with material from Prof. Michael Gerndt)

Немного подробнее переменные MPI_...

Communicators: MPI_COMM_WORLD

- Group of MPI processes
- Work with communication context - always!

Communications across communicators are not possible!
- Different context :(

But! one MPI process can be in multiple communicators

Default:
MPI_COMM_WORLD: all initial MPI processes
MPI_COMM_SELF: only own MPI process

MPI Processes vs OS Processes vs Rank

MPI concept!
Communication endpoint that can send and receive messages. Members of MPI groups and Communicators.

But!
Usually implemented as processes (OS)

Rank is identifier of an MPI process relative to an MPI communicator

MPI_Send – initiate message send and blocks until send buffer can be reused
(i.e. nonparallel nonparallel)
possible deadlocks!

MPI_Send

Send one message to another MPI process on a given communicator

```
int MPI_Send (void *buf, int count, MPI_Datatype dtype,
              int dest, int tag, MPI_Comm comm)
```

IN buf	Address of the send buffer
IN count	Number of data elements of type `dtype` to be sent
IN dtype	Data type
IN dest	Receiver (rank of target MPI process in `comm`)
IN tag	Message tag
IN comm	Communicator

Initiates message send and blocks until send buffer can be reused.

- Note: possibilities of deadlocks (!)

MPI_Recv

Receives of a message from another MPI process on a given communicator

```
int MPI_Recv (void *buf, int count, MPI_Datatype dtype,
int source, int tag, MPI_Comm comm, MPI_Status *status)
```

have to match with sender

OUT buf	Address of the receive buffer
IN count	Number of data elements of type dtype to be received
IN dtype	Data type
IN source	Sender (rank of source MPI process in comm)
IN tag	Message tag
IN comm	Communicator
OUT status	Status information

Blocking receive of a message

- Blocks execution until a matching message has been received
- Note: possibilities of deadlocks (!)

Sender and tag can be specified as wild cards

- MPI_ANY_SOURCE and MPI_ANY_TAG
- Can be used instead of source and tag arguments
- There is no wild card for the communicator/context

We have a guaranteed messages order!
(but this can be overwritten)

Messages are supposed to have matched types
Casting types can be dangerous!

Message buffers must be at least as long as the message

Most common MPI types

MPI_CHAR	char (treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_C_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_BYTE	
MPI_PACKED	

Custom datatypes can be defined through special routines.

In C, MPI_Status is a structure that includes the following fields:
- MPI_SOURCE: sender of the message
- MPI_TAG: message tag
- MPI_ERROR: error code

Important for wild card communication
- Identifies the actual channel the message was transmitted on

The actual length of the received message can be determined via

```
int MPI_Get_count(const MPI_Status *status,
                  MPI_Datatype dtype, int *count)
```

IN status	MPI_Status object returned from the receive call
IN dtype	Data type used in receive
OUT count	Number of data elements actually received

```
double MPI_Wtime(void)
```

Return wall clock time since a reference time stamp in the past

- Unit: seconds
- Reference time stamp is fixed for one execution
- Time can be global, but doesn't have to be
  - In fact, in most cases it is not
  - Users have to synchronize, if needed
- Note: return value is not an error code

Collective Operations

Properties

- Must be executed by all processes of the communicator
- Must be executed in the same sequence
- Arguments across all MPI processes must be consistent (or equal)

MPI provides three classes of collective operations

- Synchronization
  - Barrier
- Communication, e.g.,
  - Broadcast
  - Gather – collect from all nodes
  - Scatter – send chunks to all nodes (distribute work)
- Reduction, e.g.,
  - Global value returned to one or all processes.
  - Combination with subsequent scatter.
  - Parallel prefix operations

MPI_Barrier

```
int MPI_Barrier (MPI_Comm comm)
```

IN comm: *Communicator*

This operation synchronizes all MPI processes, i.e., no MPI process can return from the call until all MPI processes have called it.

```
#include "mpi.h"

main (int argc,char *argv[]){
...
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);

MPI_Barrier(MPI_COMM_WORLD);
...
```

Broadcast Communication: MPI_Bcast

```
int MPI_Bcast (void *buf, int count, MPI_Datatype dtype,
               int root, MPI_Comm comm)
```

IN buf: *Adress of send/receive buffer*
IN count: *Number of elements*
IN dtype: *Data type*
IN root: *Rank of send/broadcasting MPI process* – our rank (send to everyone but root)
IN comm: *Communicator*

The contents of the send buffer is copied to all other MPI processes

Collective vs P2P operations
- No tag
- Number of elements sent must be equal to number of elements received

The routines do not necessarily synchronize processes wrt. P2P

Broadcast doesn't guarantee everyone recieved the message!

Collective vs P2P operations
- No tag !!
- Number of elements sent must be equal to number of elements received

any collectives

```
switch (rank) {
case 0:
  MPI_Bcast (buf1, ct, tp, 0, comm);
  MPI_Send (buf2, ct, tp, 1, tag, comm);
  break;
case 1:
  MPI_Recv (buf2, ct, tp, MPI_ANY_SOURCE, tag, ...);
  MPI_Bcast (buf1, ct, tp, 0, comm);
  MPI_Recv (buf2, ct, tp,MPI_ANY_SOURCE, tag, ...);
  break;
case 2:
  MPI_Send (buf2, ct, tp, 1, tag, comm);
  MPI_Bcast (buf1, ct, tp, 0, comm);
  break;
}
```

Communicators

Why want anything but MPI_COMM_WORLD
- Creation of subgroups for collective communication
- Isolation between communication operations

Duplicating Communicators

Isolation important for libraries
- Different libraries should use different communicators
- Avoids “cross-talk” and side effects

Typical strategy for libraries
- Libraries get a communicator passed at initialization (often WORLD)
- The library then clones the communicator and stores the handle

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

IN comm	communicator (handle)
OUT newcomm	copy of comm (handle)

↑ you can even send!

Collective operation

Communicators should be freed, when no longer in use

```
int MPI_Comm_free(MPI_Comm *comm)
```

← so application doesn't starve

Creating Subcommunicators

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

	IN comm	*Parent communicator*
	IN color	*Subset color*
highest/lowest values →	IN key	*Key to determine rank order in new communicator*
	OUT newcomm	*New communicator*

Creates new communicator(s)

- Collective operation over parent communicator
- All MPI processes that pass the same ***color***, will be in same new communicator
- ***Key*** argument determines the rank order in the new communicator

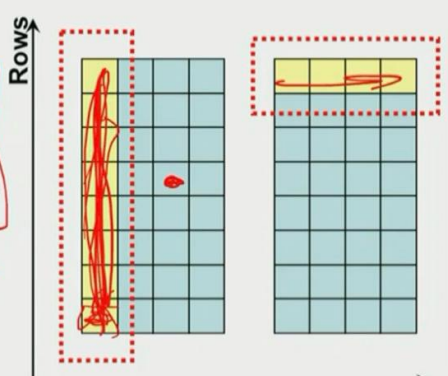
MPI processes can opt out

- Pass `MPI_UNDEFINED` as color
- Will return `MPI_COMM_NULL` as new communicator

Collective operation

Example: Row and Column Communicators

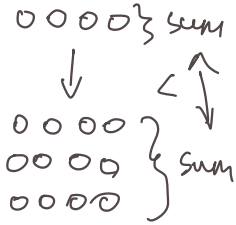
```
int rank, size, col, row;
MPI_Comm row_comm, col_comm;

MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

Row = rank % N;
Col = rank / N;

MPI_Comm_split(MPI_COMM_WORLD,
               row, col, &row_comm);

MPI_Comm_split(MPI_COMM_WORLD,
               col, row, &col_comm);
...
MPI_Bcast(buf, 1, MPI_INT, 0, col_comm);

(works also for Send & Recv)
```



Weak Scaling

Keeping the problem size per HW thread/core/node constant

- Larger machine -> larger problem
  - Expanding the problem domain
  - Increasing refinement
- Traditionally most common way to deal with scaling in HPC



Assumptions

- Machines are never big enough
- Fully loading a node/core/HW thread is the right thing to do
  - Exploiting the resources on one unit and keeping that constant

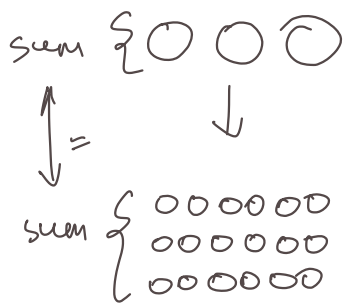
Advantages

- Execution properties (like cache behavior) often stay fixed
- Easier to scale, as overheads stay roughly constant as well

Challenges

- Keeping load constant can be tricky for complex applications (2D/3D problems)
- Multiple ways to repartition a workload

Strong Scaling



Keeping the total problem constant
- Larger machine -> (hopefully) faster execution
- Need to adjust problem distribution
- Traditionally not the most common type of scaling
  - But becoming rapidly more and more relevant

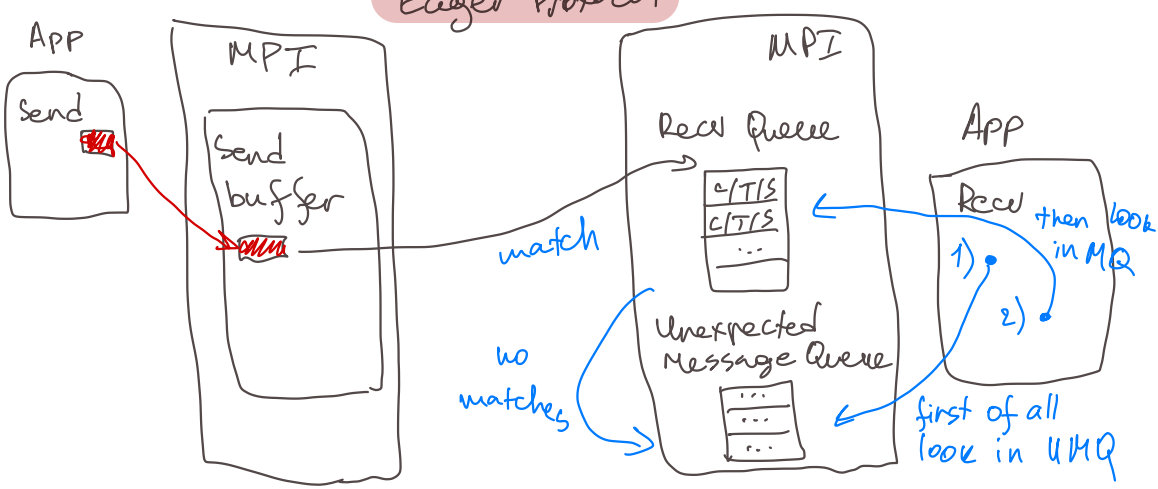
Assumptions:
- The machine is big enough for the problem
- Goal: reducing time to solution and increasing throughput

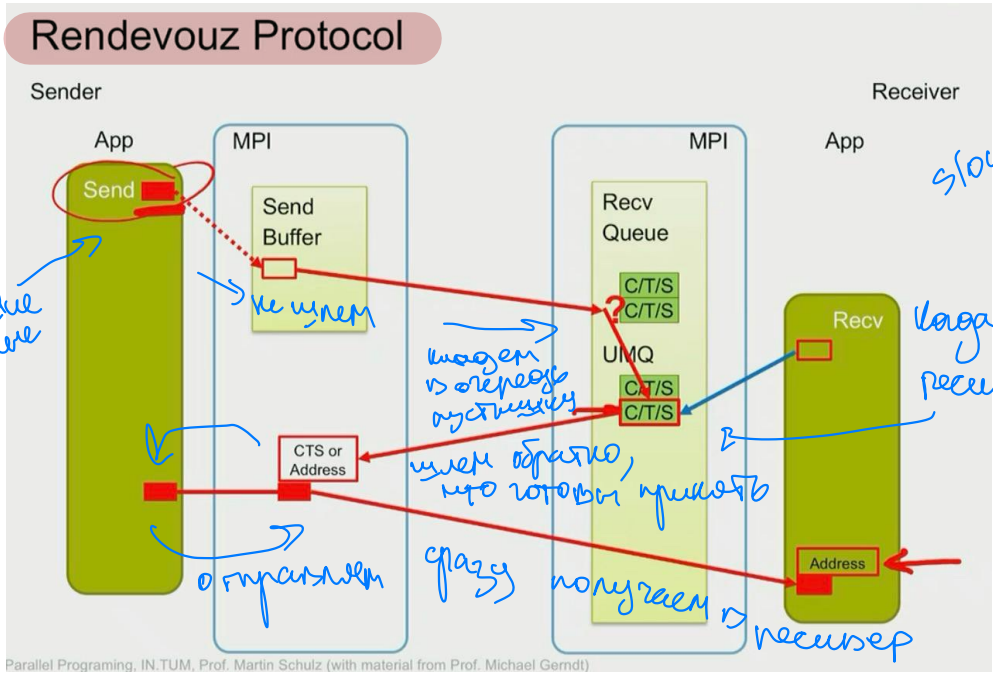
Needed for time critical tasks
- Emergency response to natural disasters
- Real-time simulations
- Large-scale ensemble calculations

Challenges
- Harder to scale, as overheads grow with smaller per HW thread workloads
- Changing executing characteristics (cache miss rates, etc.)

How Messages are Transfered (typically)

Eager Protocol





Eager vs. Rendezvous Protocol

Eager protocol

- Send messages directly
  - Advantage: avoids extra handshake
  - Disadvantage: adds extra copies
- Suitable for short messsages with low copy overhead

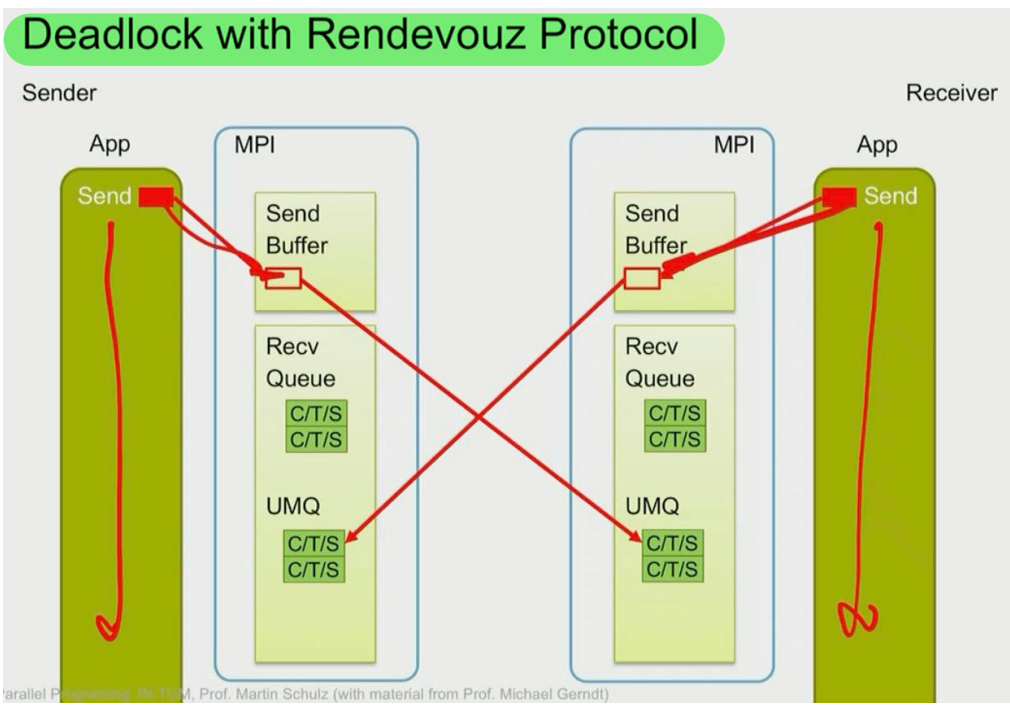
Rendezvous protocol

- Send header to retrieve address, then send message
  - Advantage: Can deposit message directly
  - Disadvantage: extra handshake
- Suitable for long messsages with high copy overhead

Cross-over point

- MPI implementations typically switch protocol based on message size
- Typically defined as eager limit
- Depends on system setup
- Can often be modified

Обычно исп. сразу оба, один для больших сообщ., другой для мал.
Поэтому если изм. поведение при изменении размера сообщ., возможно, это дедлок



Bidirectional Exchange

Transfer data elements between two "adjacent"
- Same or similar data exchanged between two MPI processes
- Example: halo exchange

MPI_Send(to: 1) — MPI_Send(to: 0)
MPI_Recv(from: 1) — MPI_Recv(from: 0)

Problem: deadlock possible
- MPI_Send may need resources on receiver side

Changing the order of send and receive

MPI_Send(to: 1) — MPI_Recv(from: 0)
MPI_Recv(from: 1) — MPI_Send(to: 0)

long before receive

Helps to avoid deadlock, but increases time

`MPI_Send`
- Standard send operation, no extra synchronization
- Sender side handling implementation defined

`MPI_Bsend`
- Buffered Send
- Force the use of a send buffer
- Returns immediately, but costs resources

`MPI_Ssend`
- Synchronous send
- Only returns once receive has started
- Adds extra synchronization, but can be costly

`MPI_Rsend`
- Ready Send
- User must ensure that receive has been posted
- Enables faster communication, but needs implicit synchronization

message is - guaranteed to be sent eventually

blocks until sent?

не блокируется, имеет в буфере и когда-нибудь отправит

Блокируется, пока reciever не получит (recieve that matches the message, was called)

Зуб даю, на стороне ресивера все будет хорошо и match найдется

оптимизация функции

Bidirectional Send/Recv

```
int MPI_Sendrecv (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                  int dest, int sendtag,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  int source, MPI_Datatype recvtag,
                  MPI_Comm comm, MPI_Status *status)
```

Combined blocking send and receive
- Can be matched with regular send/recv calls
- Can be matched with other Sendrecv calls with other destinations

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,
                         int dest, int sendtag,
                         int source, int recvtag,
                         MPI_Comm comm, MPI_Status *status)
```

Same principle, but combined send and receive buffer
- Requires same message types

Checking the UMQ with MPI_Probe/Iprobe

Allow polling of incoming messages without actually receiving them.

- `MPI_Probe`
  - Blocks until a matching message was found
  - Input: source, tag, communicator
  - Output: status object – size и tg. Можно создать подходящий reciever
- `MPI_Iprobe`
  - Terminates after checking whether a matching messages is available
  - Input: source, tag, communicator
  - Output: flag (message found yes/no) and status object

то же самое, но не блокер.

Usage

- Both operations inspect the Unexpected message queue
- Allows to wait for a message before issuing the receive
- Useful for wildcard receives
  - Issue probe call with wildcard
  - Call returns concrete source/tag pair
- MPI_Iprobe can be used to overlap wait time with useful work.

Nonblocking P2P Operations in MPI

Initiation routines as counterparts to blocking P2P operations

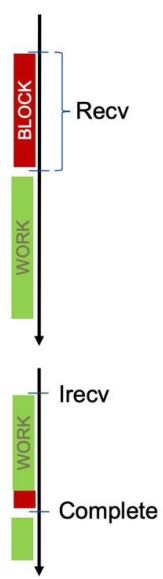
- Similar arguments, same semantics
- Can be matched with blocking operations
- Returns request object to track progress

Initiation call only starts operation

- Completion is indicated later
- Send and receive buffers are off limits before completion
- Semantics of operation

`MPI_Request`

- Request object
- User allocates object, but MPI maintains state
- Initiated operations must be completed



```
int MPI_Isend (void* buf, int count, MPI_Datatype dtype, int dest,
int tag, MPI_Comm comm, MPI_Request* request)
```

IN buf: *Send buffer*
IN count: *Number of elements*
IN dtype: *Data type*
IN dest: *Receiver*
IN tag: *Tag*
IN comm: *Communicator*
OUT request: *Reference to request*

No status param! As we do not know the status yet

```
int MPI_Irecv (void* buf, int count, MPI_Datatype dtype, int source,
int tag, MPI_Comm comm, MPI_Request* request)
```

IN buf: *receive buffer*
IN count: *number of entries in receive buffer*
IN dtype: *data type*
IN dest: *sender*
IN tag: *tag*
IN comm communicator
OUT request: *reference to request*

`MPI_Isend`

- Standard send operation, no extra synchronization
- Sender side handling implementation defined

`MPI_Ibsend`

- Buffered Send
- Force the use of a send buffer
- Completes immediately, but costs resources

Все возвраты быстро, но completion - как раньше

`MPI_Issend`

- Synchronous send
- Only completes once receive has started
- Adds extra synchronization, but can be costly

`MPI_Irsend`

- Ready Send
- User must ensure that receive has been posted at Irsend time
- Enables faster communication, but needs implicit synchronization

All blocking collectives in MPI have a non-blocking counterpart

Completion Operations

Option 1: Blocking completion

```
int MPI_Wait (MPI_Request *request, MPI_Status *status)
```

INOUT request: *request*
OUT status: *status of completed operation*

Returns if …
request is complete
request is `MPI_REQUEST_NULL`

Sets request to `MPI_REQUEST_NULL`

Frees all resources associated with request

```
{
   MPI_Request req;
   MPI_Status  status;
   int msg[10];
   ...
   MPI_Irecv(msg, 10, MPI_INT, MPI_ANY_SOURCE, 42,
             MPI_COMM_WORLD, &req);
   ...
       <do work>
   ...
   MPI_Wait(&req, &status);
   printf("Processing message from %i\n",
          status.MPI_SOURCE);
   ...
}
```

Option 2: Nonblocking/Polling completion

Не ждем, а просто возвращ. результат проверки

```
int MPI_Test (MPI_Request *request, int *flag,
              MPI_Status *status)
```

по сути bool

INOUT request: *request*

OUT flag: flag, whether complete or not

OUT status: *status of completed operation*

Returns if immediately

flag set to True if request is complete

request is `MPI_REQUEST_NULL`

If completed,

Sets request to `MPI_REQUEST_NULL`

Frees all resources associated with request

```
{
   MPI_Request req;
   MPI_Status  status;
   int msg[10], flag;
   ...
   MPI_Irecv(msg, 10, MPI_INT, MPI_ANY_SOURCE, 42,
             MPI_COMM_WORLD, &req);
   do
   {
       ...
       <do work>
       ...
       MPI_Test(&req, &flag, &status);
   } while (flag==0);

   printf("Processing message from %i\n",
          status.MPI_SOURCE);
}
```

например, в цикле

`MPI_Wait`

- Blocking wait for one request
- Output: status object

`MPI_Test`

- Checks completion of one request
- Output: flag, True if complete
- Output: status object

`MPI_Waitall` ← array of requests

- Blocking wait for several requests
  - Input: array of requests
  - Some can be `MPI_REQUEST_NULL`
- Returns once all are complete
  - Output: array of status objects

`MPI_Testall`

- Checks completion of several requests
  - Input: array of requests
  - Some can be `MPI_REQUEST_NULL`
- Returns immediately
  - Output: flag, True if all complete
  - Output: array of status objects

`MPI_Waitany (MPI_Waitsome)` ← also array

- Blocking wait for several requests
  - Input: array of requests
  - Some can be `MPI_REQUEST_NULL`
- Returns once at least one is complete
  - Output: "completed" index (array)
  - Output: (array of) status object(s)

`MPI_Testany (MPI_Testsome)`

- Checks completion of several requests
  - Input: array of requests
  - Some can be `MPI_REQUEST_NULL`
- Returns immediately
  - Output: flag, True if at least one complete
  - Output: "completed" index (array)
  - Output: (array of) status object(s)

Terminology before MPI 4.0

Blocking routines

- After return all buffers can be reused again
- E.g., after MPI_Send, the data in the send buffer has been consumed by MPI

Nonblocking routines

- After return buffers are still under MPI control
- E.g., after MPI_Isend, the data buffers cannot be written until MPI_Wait

MPI Blocking ≠ English Blocking

↙ может быть на той же машине

Non-Local routines

- Requires some (specific) other execution to run in another process
- E.g., for MPI_Recv to complete, another process must call MPI_(X)send

Local routines

- Not non-local

MPI Local ≠ Executes only locally in one MPI process

All MPI routines are "weak local", i.e., assume progress on all MPI processes

Terminology after MPI 4.0

Non-Local *procedures*
- Requires some (specific) other execution to run in another process
- E.g., for MPI_Recv to complete, another process must call MPI_(X)send

Local procedures
- Not non-local, now explicitly weak local

Completing procedures (was blocking)
- After return all buffers can be reused again
- E.g., after MPI_Send, the data in the send buffer has been consumed by MPI

Incomplete procedures (was non-blocking)
- After return buffers are still under MPI control
- E.g., after MPI_Isend, the data buffers cannot be written until MPI_Wait

Non-Blocking procedures
- incomplete and local

выполняем какую "локальную", нам ничего не мешает, а идем где-то на фоне чего-то non-local- и бог с ним

MPI Blocking
≠
English Blocking

Blocking procedures
- Not "non-blocking"

```
MPI_Request req[2];
MPI_Status  status[2];
int msg_in[10], msg_out[10];

MPI_Isend(msg_out, 10, MPI_INT, 1, 42, MPI_COMM_WORLD, &(req[0]);
MPI_Irecv(msg_in,  10, MPI_INT, 1, 42, MPI_COMM_WORLD, &(req[1]);


MPI_Waitall(2, req, status);

/* status[1] is the receive status */
```

массив массив

MPI_Reduce

```
int MPI_Reduce (void* sbuf, void* rbuf, int count, MPI_Datatype dtype,
                MPI_Op op, int root, MPI_Comm comm)
```

IN sbuf:	*Send buffer*
OUT rbuf:	*Receive buffer*
IN count:	*Number of elements*
IN dtype:	*Data type*
IN op:	*Operation*
IN root:	*Rank of MPI process receiving the reduced results*
IN comm:	*Communicator*

Combines the elements in the send buffer according to “op”
Delivers the result to root

MPI Reductions

The data of the processes are combined via a specified operation
- Predefined operators, like ’+’
- User defined operators (see `MPI_Op_create` and `MPI_Op_free`)

Often implemented in optimized way, e.g., using tree structures

`MPI_Reduce`
- Reduce elements from all processes based on operation

`MPI_Allreduce`
- Reduction followed by broadcast

`MPI_Reducescatter_block`
- Reduction followed by a scatter operation (equal chunks)

`MPI_Reducescatter`
- Reduction followed by a scatter operation (varying chunks)

`MPI_Scan` – первый процесс: x_0, второй x_0+x_1, третий $x_0+x_1+x_2$ ...
- Reduction using a prefix operation

Available Reduction Operators

`MPI_MAX`	maximum
`MPI_MIN`	minimum
`MPI_SUM`	sum
`MPI_PROD`	product
`MPI_LAND`	logical and
`MPI_BAND`	bit-wise and
`MPI_LOR`	logical or
`MPI_BOR`	bit-wise or
`MPI_LXOR`	logical exclusive or (xor)
`MPI_BXOR`	bit-wise exclusive or (xor)

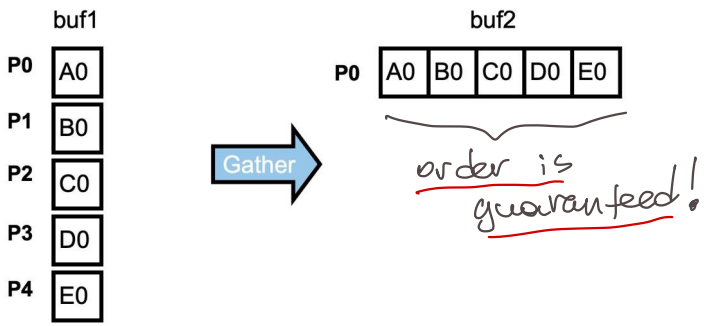
Standard has precise mapping of operations to allowable datatypes

Use with speciality datatype (tupel: <value> + <location>)

`MPI_MAXLOC`	max value and location
`MPI_MINLOC`	min value and location

Gather Operation

Get data from all MPI processes into one local array



MPI_Gather

```
int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

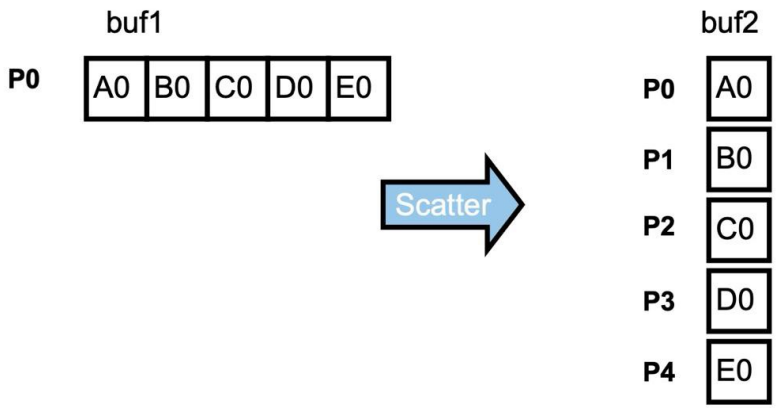
IN sendbuf:	*Send buffer*
IN sendcount:	*Number of elements to be sent to the root*
IN sendtype:	*Data type*
OUT recvbuf:	*Receive buffer*
IN recvcount:	*Number of elements to be received from each MPI process.*
IN recvtype:	*Data type*
IN root:	*Rank of receiving MPI Process*
IN comm	*Communicator*

The root receives from all processes the data in the send buffer.

It stores the data in the receive buffer ordered by the process number of the senders.

Scatter

почти то broadcast, но умеет разные данные



MPI_Scatter

```
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

IN sendbuf:	*Send buffer*
IN sendcount:	*Number of elements sent to each MPI process*
IN sendtype:	*Data type*
OUT recvbuf:	*Receive buffer*
IN recvcount:	*Number of elements to be received by a MPI process*
IN recvtype:	*Data type*
IN root:	*Rank of sending/scattering MPI process*
IN comm:	*Communicator*

The root sends a part of its send buffer to each process.

Process *k* receives *sendcount* elements starting with *sendbuf+k*sendcount*.

`MPI_Gatherv`

- Gather operation with different number of data elements per process

`MPI_Allgather`

- Gather operation with broadcast to all processes

`MPI_Allgatherv`

- Gather operation with different number of data elements AND broadcast

`MPI_Scatterv`

- Scatter operation with different number of data elements per process

`MPI_Alltoall`

- Every process sends to every other process (different data)

`MPI_Alltoallv`

- All to all with different numbers of data elements per process

`MPI_Alltoallw`

- All to all with different numbers of data elements and types per process

MPI_Gatherv

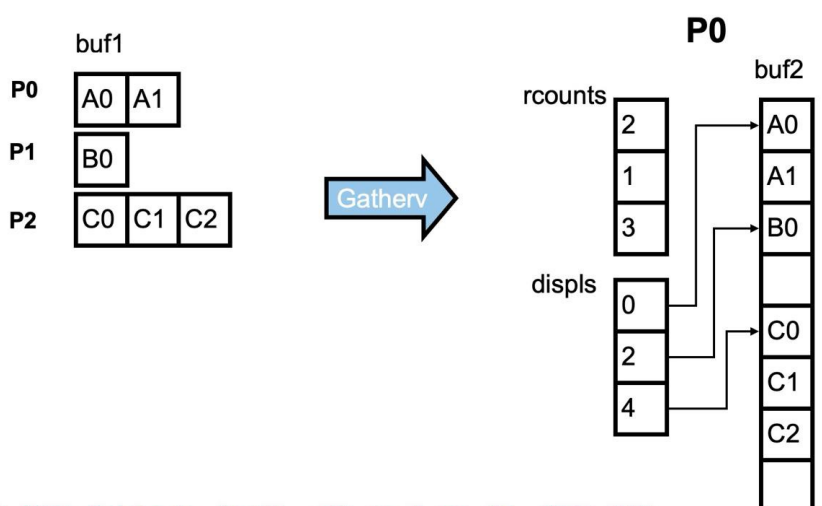
```
int MPI_Gatherv (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int *recvcount, int *displs,
                 MPI_Datatype recvtype, int root, MPI_Comm comm)
```

IN sendbuf	*Send buffer*
IN sendcount	*Number of elements sent*
IN sendtype	*Send type*
OUT recvbuf	*Receive buffer*
IN recvcounts	*Array with the number of elements to be received from each MPI process*
IN displs	*Array with the first index in the receive buffer for each MPI process*
IN recvtype	*Receive data type*
IN Root	*Receiving/Gathering MPI process rank*
IN comm	*Communicator*

In contrast to MPI_Gather, each process can send a different number of elements

The individual messages are stored according to *displs* in the receive buffer

MPI_Gatherv




```
int MPI_Gatherv (buf1, <2 or 1 or 3>, MPI_INT,
                 buf2, rcounts, displs, MPI_INT,
                 <rank of P0 in MPI_COMM_WORLD>,
                 MPI_COMM_WORLD)
```

MPI_Alltoall

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
```

IN sendbuf	*Send buffer*
IN sendcount	*Number of elements sent*
IN sendtype	*Send type*
OUT recvbuf	*Receive buffer*
IN recvcount	*Number of elements to be received by a MPI process*
IN recvtype	*Data type*
IN comm	*Communicator*

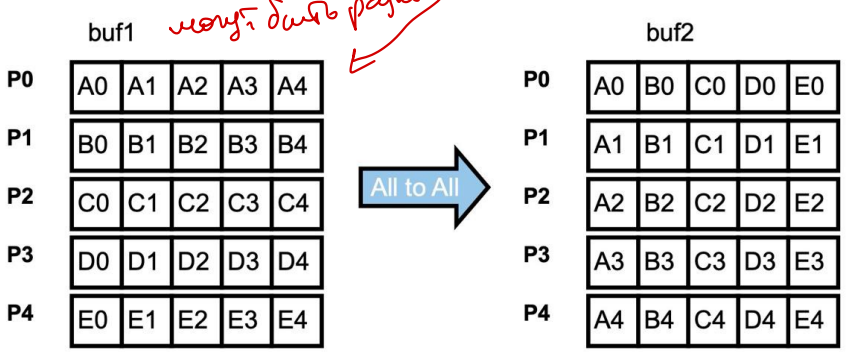
Sends data from each MPI process to each other MPI process

Warning: this is often a scaling bottleneck

MPI_Alltoall

Потому что scatter и gather

могут быть разные




```
int MPI_Alltoall (buf1, 1, MPI_INT,
                  buf2, 1, MPI_INT,
                  MPI_COMM_WORLD)
```

Nonblocking Collectives (since MPI 3.0)

Same concept as non-blocking P2P

- Defined for all collective operations
  - Include `MPI_Ibarrier`, the non-blocking barrier
  - Enables overlap with extra computation
- Returns request object
- Request can be used in `MPI_Wait` and `MPI_Test` operations

Cannot be mixed with blocking collectives

The initiation routines have to be executed in the same order on all processes

Example:

```
int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype,
               int root, MPI_Comm comm, MPI_Request *request)
```

INOUT buffer	starting address of buffer (choice)
IN count	number of entries in buffer (non-negative integer)
IN datatype	data type of buffer (handle)
IN root	rank of MPI process acting as broadcast root (integer)
IN comm	communicator (handle)
OUT request	communication request (handle)

MPI_Ialltoall

```
int MPI_Ialltoall(void* sendbuf, int sendcount, MPI_Datatype
                  sendtype,
                    void* recvbuf, int recvcount, MPI_Datatype recvtype,
                    MPI_Comm comm, MPI_Request *req)
```

IN sendbuf	*Send buffer*
IN sendcount	*Number of elements sent*
IN sendtype	*Send type*
OUT recvbuf	*Receive buffer*
IN recvcount	*Number of elements to be received by a MPI process*
IN recvtype	*Data type*
IN comm	*Communicator*
OUT req	*MPI Request to complete operation*

Starts the send of data from each MPI process to each other MPI process
Had to be complete later with Test or Wait

Warning: this is still often a scaling bottleneck, therefore a good target for overlap

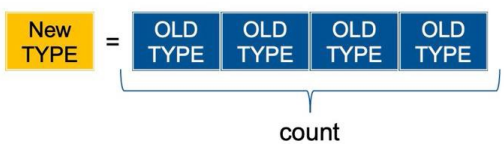
Derived MPI Datatypes

Several MPI functions available to create arbitrary layouts
- Recursive specification possible
- Declarative specification of data-layout

Constructor functions
- Additional arguments to describe new type
- Input: existing datatype (basetype for this operation)
- Output: newly created derived datatype

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
```

New datatype is “count” contiguous elements of ”oldtype”



Datatype handle

```
MPI_Datatype
```

Datatype construction

Set of routines (see previous and next slides)

Datatype commit

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

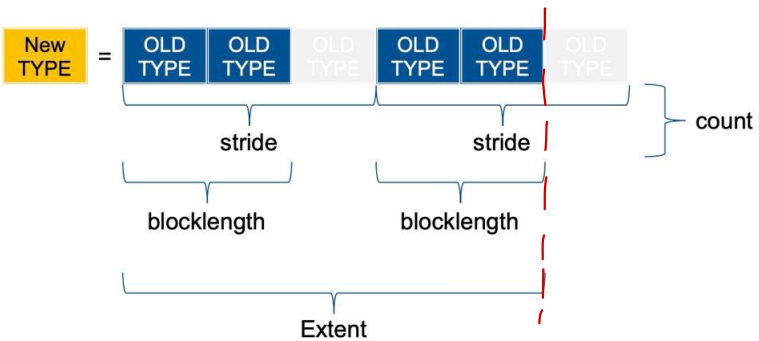
necessary before type can be used,
but uncommitted types can be used to create new types

Datatype free

```
int MPI_Type_free(MPI_Datatype *datatype)
```

Constructing a Vector Datatype

```
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

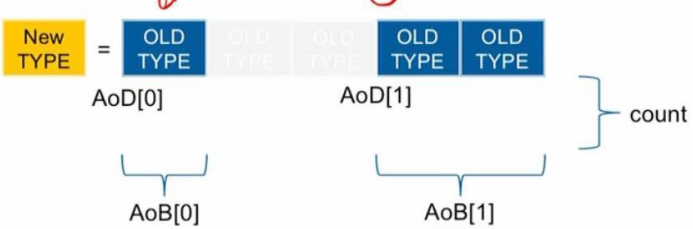


Constructing an Indexed Datatype

```
int MPI_Type_indexed(int count,
                     const int array_of_blocklengths[],
                     const int array_of_displacements[],
                     MPI_Datatype oldtype,
                     MPI_Datatype *newtype)
```

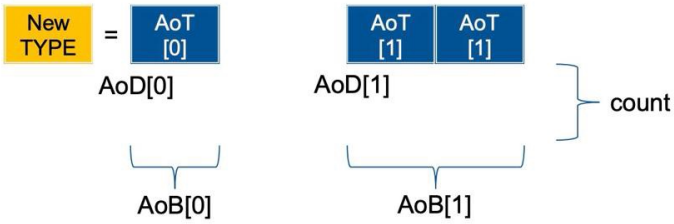
(AoB)

(AoD)



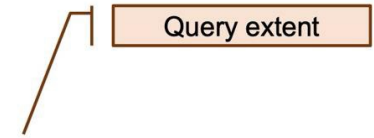
Creating a Struct Datatype

```
int MPI_Type_create_struct(int count,
              const int array_of_blocklengths[],          (AoB)
              const MPI_Aint array_of_displacements[],    (AoD)
              const MPI_Datatype array_of_types[],        (AoT)
              MPI_Datatype *newtype)
```

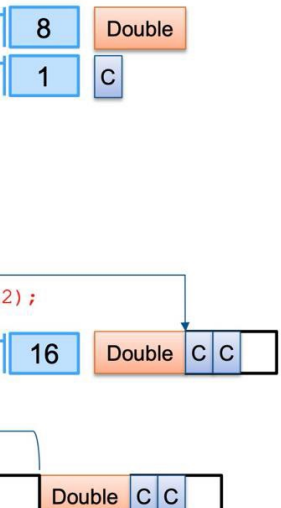


Note: Displacements in Bytes (`MPI_Aint`) - address integers

Example




```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define INSPECT(dt) {MPI_Type_get_extent(dt,&lb,&ex); printf("%li\n",ex);}

int main(int argc, char **argv)
{
        MPI_Datatype dt1[2],dt2,dt3;
        int blens[2];
        MPI_Aint ex,lb,disps[2];

        MPI_Init(&argc,&argv);

        dt1[0]=MPI_DOUBLE; INSPECT(dt1[0]);
        dt1[1]=MPI_CHAR;   INSPECT(dt1[1]);
        blens[0]=1;  blens[1]=2;
        disps[0]=0;  disps[1]=sizeof(double);
        MPI_Type_create_struct(2,blens,disps,dt1,&dt2);
        INSPECT(dt2);

        MPI_Type_vector(2,1,2,dt2,&dt3);
        INSPECT(dt3);

        MPI_Finalize();
}
```



MPI Threading

New MPI initialization routine

```
int MPI_Init_thread(int *argc, char ***argv,
                    int required, int *provided)
```

1) call_init_thread

3) finalize

Application states needed level of thread support
MPI returns which one it supports

4) check

`MPI_THREAD_SINGLE` – один и все, до конца

- Only one thread exists in the application

`MPI_THREAD_FUNNELED` - outside parallel regions, during master region

- Multithreaded, but only the main thread makes MPI calls

`MPI_THREAD_SERIALIZED` - critical & single regions

- Multithreaded, but only one thread at a time makes MPI calls

`MPI_THREAD_MULTIPLE`

- Multithreaded and any thread can make MPI calls at any time

Consequences of using `MPI_THREAD_MULTIPLE`

Each call to MPI completes on its own

- Effect is as if they would have been called sequentially in some order
- Blocking calls will only block the calling thread

```
T1: MPI_Send( to: 1 )        T1: MPI_Send( to: 0 )
T2: MPI_Recv( from: 1 )      T2: MPI_Recv( from: 0 )
```

User is responsible for making sure racing calls are avoided

- Example: access to an already freed object
- Collective operations must be ordered correctly among threads

Collective operations should be called in the same order!

```
T1: MPI_Bcast( comm )        T1: MPI_Bcast( comm )
T2: MPI_Barrier( comm )      T2: MPI_Barrier( comm )
```

order is not determined

Possible performance impact, e.g., caused by locking in MPI

MPI - SPMD (single process, multiple data)

MPI Persistent Communication

setup communication

```
int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype,
                  int dest, int tag, MPI_Comm comm, MPI_Request *request);
```

```
int MPI_Start(MPI_Request *request);
```
every loop iteration

```
int MPI_Request_free(MPI_Request *request);
```
after loop

Also available for collectives

when send-rs

Sender: fill partition and move as ready
Receiver: notification for each arriving partition

Partitions do not need to be finished in order!
Each thread fills one or more partitions.

Sender:

1. `MPI_Psend_init` sets up the communication.
2. `MPI_Start` indicates the coming transfers to MPI.
3. `MPI_Pready` indicates that a partition is ready. - filled one partition
4. `MPI_Wait` completes the operation.
5. `MPI_Request_free` frees the resources.

Receiver:

1. `MPI_Precv_init` sets up the communication.
2. `MPI_Start` indicates the coming transfers to MPI.
3. `MPI_Parrived` indicates whether a partition arrived.
4. `MPI_Wait` completes the operation.
5. `MPI_Request_free` frees the resources.

true or false

```
if (rank == 0) {
  MPI_Psend_init(message, sendbuf, arraysize, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_INFO_NULL, &request);

  for (int k = 0; k < num_steps; k++) {
    MPI_Start(&request);

    #pragma omp parallel
    {
      int i = omp_get_thread_num();
      // Do computation on i-th block of data
      MPI_Pready(i, &request);
    }

    MPI_Wait(&request);
  }

  MPI_Request_free(&request);
} else if (rank == 1) {
```