OpenMP, MPI and hybrid programming

An overview of the de-facto industrial standards

Vincent Keller
Nicolas Richart

May 11, 2015

Table of Contents

Week 5 (March 18) : OpenMP

Week 6 (March 25) : MPI basics

Week 7 (April 1) : MPI advanced

Week 8 (April 15) : Hybrid programming - Project proposal

Before you start your parallel implementation

- **You have no serial code :** design your application in a parallel way from scratch
- **You have a serial code :** follow a Debugging-Profiling-Optimization cycle before any parallelization

Debugging ?

- Find and correct bugs within an application
- Bugs can be of various nature : division by zero, buffer overflow, null pointer, infinite loops, etc..
- The compiler is (very) rarely able to recognize a bug at compilation time and the error is (very) rarely explicit regarding the bug ("syntax error")
- Use standard tools like `gdb`
- A multi-threaded code can be tricky to debug (race conditions, deadlocks, etc..)

Profiling ?

Where do I spend most of the time ?

- (good) using tools like `gprof` for a serial implementation (or the family of tools you tackled last week)
- (bad) "by hand" using timings and `printf`'s

Optimization ?

By order of complexity :

1. compiler and linker flags
2. “handmade” refactoring (loop reordering, usage of intrinsics, memory alignment, etc..)
3. algorithmic changes

Parallelization ?

1. Is it worth to parallelize my code ? Does my algorithm scale ?
2. Performance prediction ?
3. Timing diagram ?
4. Bottlenecks ?
5. Which parallel paradigm should I chose ? What is the target architecture (SMP, cluster, GPU, hybrid, etc..) ?

Lecture based on specifications ver 3.1

Releases history, present and future

- October 1997: Fortran version 1.0
- Late 1998: C/C++ version 1.0
- June 2000: Fortran version 2.0
- April 2002: C/C++ version 2.0
- June 2005: Combined C/C++ and Fortran version 2.5
- May 2008: Combined C/C++ and Fortran version 3.0
- **July 2011: Combined C/C++ and Fortran version 3.1**
- July 2013: Combined C/C++ and Fortran version 4.0

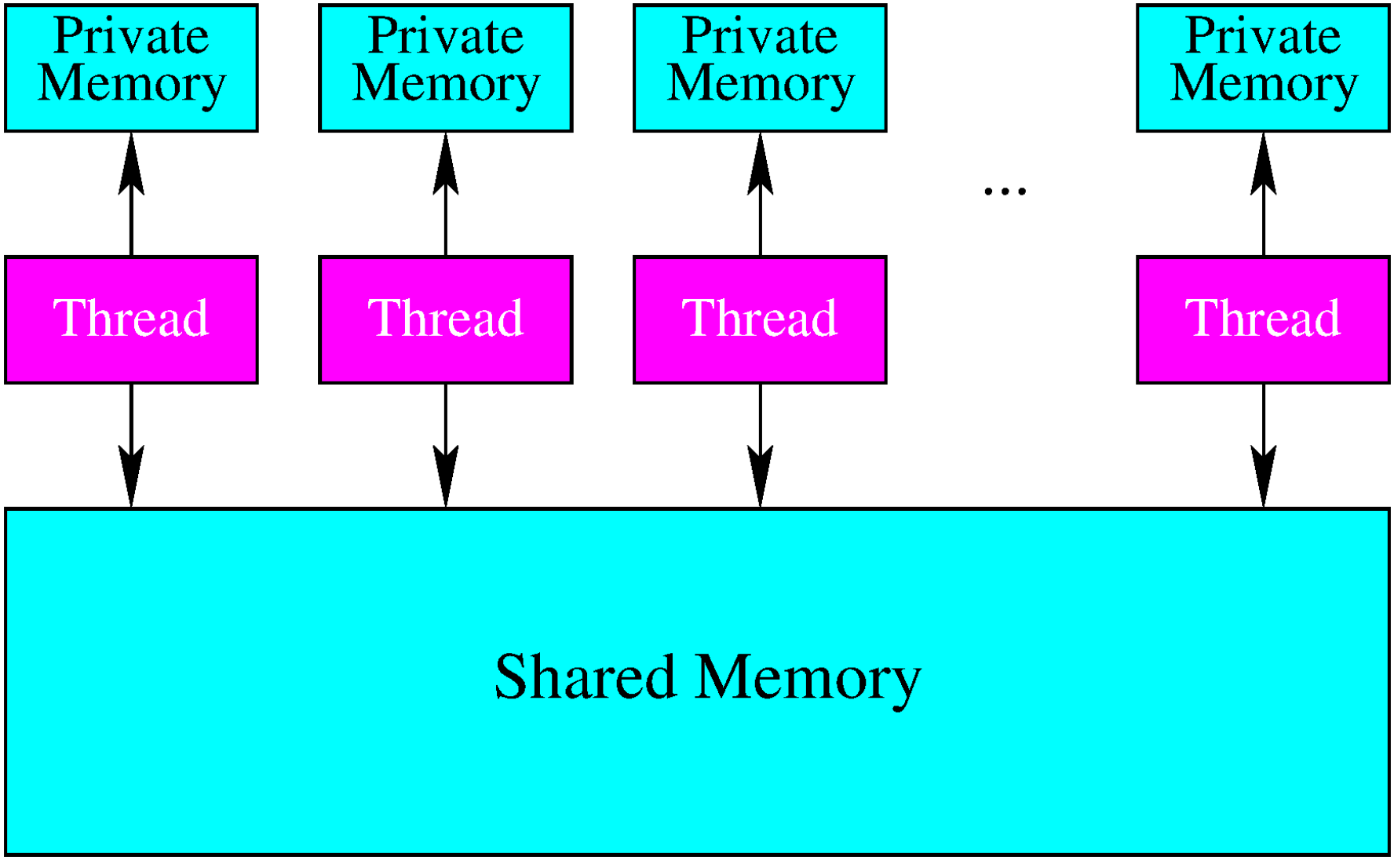
Terminology

- **thread :** an execution entity with a stack and a static memory (*threadprivate memory*)
- **OpenMP thread :** a *thread* managed by the OpenMP runtime
- **thread-safe routine :** a routine that can be executed concurrently
- **processor :** an HW unit on which one or more *OpenMP thread* can execute

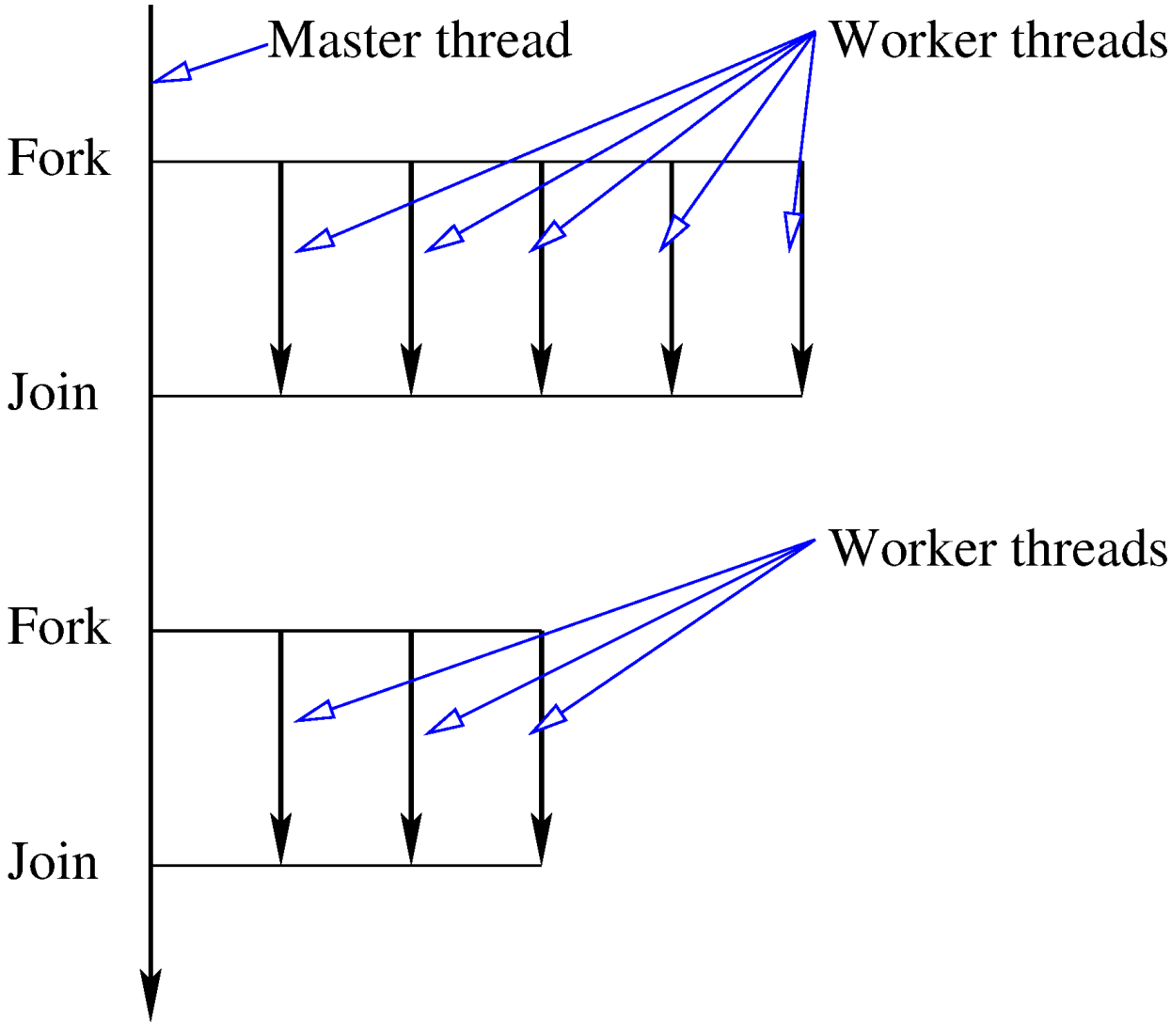
Execution and memory models

- Execution model : fork-join
- One heavy thread (process) per program (initial thread)
- leightweigt threads for parallel regions. threads are assigned to cores by the OS
- No implicit synchronization (except at the beginning and at the end of a parallel region)
- Shared Memory with shared variables
- Private Memory per thread with threadprivate variables

Memory model (simplified)



Execution model (simplified)



OpenMP and MPI/pthreads

- **OpenMP** $\neq$ OpenMPI
- All what you can do with OpenMP can be done with MPI and/or pthreads
- easier **BUT** data coherence/consistency

Syntax in C

OpenMP directives are written as pragmas: `#pragma omp`

Use the conditional compilation flag `#if defined _OPENMP` for the preprocessor

Compilation using the GNU gcc compiler:

```
gcc -fopenmp -lgomp  ex1.c -o ex1
```

Compilation using the Intel C compiler:

```
icc -openmp ex1.c -o ex1
```

Hello World in C

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main(int argc, char *argv[]) {
4     int myrank=0;
5     int mysize=1;
6  #if defined (_OPENMP)
7  #pragma omp parallel default(shared) private(myrank,
       mysize)
8  {
9     mysize = omp_get_num_threads();
10    myrank = omp_get_thread_num();
11 #endif
12    printf("Hello from thread %d out of %d\n", myrank,
          mysize);
13 #if defined (_OPENMP)
14 }
15 #endif
16    return 0;
```

Syntax in Fortran 90

OpenMP directives are written as comments: `!$omp omp`

Sentinels `!$` are authorized for conditional compilation (preprocessor)

Compilation using the GNU gfortran compiler:

```
gfortran -fopenmp -lgomp  ex1.f90 -o ex1
```

Compilation using the Intel Fortran compiler:

```
ifort -openmp ex1.f90 -o ex1
```

Hello World in Fortran 90

```
1  program ex1
2     implicit none
3     integer :: myrank, mysize
4  !$ integer, external :: omp_get_num_threads,
       omp_get_thread_num
5     myrank=0
6     mysize=1
7  !$omp  parallel default(shared) private(myrank,
       mysize)
8  !$      mysize = omp_get_num_threads()
9  !$      myrank = omp_get_thread_num()
10    print *, "Hello from thread",myrank,"out of",
          mysize
11 !$omp  end parallel
12 end program ex1
```

Number of concurrent threads

The number of threads is specified in a hardcoded way (*omp_set_num_threads*()) or via an environment variable.

BASH-like shells :

```
export OMP_NUM_THREADS=4
```

CSH-like shells :

```
setenv OMP_NUM_THREADS 4
```

Components of OpenMP

- Compiler directives (written as comments) that allow work sharing, synchronization and data scoping
- A runtime library (libomp.so) that contains informal, data access and synchronization directives
- Environment variables

The `parallel` construct

Syntax

This is the mother of all constructs in OpenMP. It starts a parallel execution.

```
1 #pragma omp parallel [clause[[,] clause]...]
2 {
3    structured-block
4 }
```

where *clause* is one of the following:

- `if` or `num_threads` : conditional clause
- `default(private | firstprivate | shared | none)` : default data scoping
- `private(`*list*`)`, `firstprivate(`*list*`)`, `shared(`*list*`)` or `copyin(`*list*`)` : data scoping
- `reduction({` *operator | intrinsic_procedure_name* `} :` *list*`)`

Example : the `if` conditional clause

The `if` clause

A `if` test specifies if a parallel region must be executed in parallel or not:

```
if (n<2) then
   execute test(n) serial
else
   execute test(n) in parallel
endif
```

```
1 #pragma omp parallel if (n<2)
2    test(n);
```

The `if` clause [output]

```
vkeller@mathicsepc13:~/OpenMP/exercises/C$ ./ex3
 var =       1  : Code is executed by only one thread
 var =       2  : Code is executed by only one thread
 Parallelized with          3 threads :           3
 Parallelized with          4 threads :           4
```

Data scoping

What is data scoping ?

- most common source of errors
- determine which variables are **private** to a thread, which are **shared** among all the threads
- In case of a private variable, what is its value when entering the parallel region **firstprivate**, what is its value when leaving the parallel region **lastprivate**
- The default scope (if none are specified) is **shared**
- most difficult part of OpenMP

The data sharing-attributes `shared` and `private`

Syntax

These attributes determines the scope (visibility) of a single or list of variables

```
1 shared(list1) private(list2)
```

- The `private` attribute : the data is private to each thread and non-initiatilized. Each thread has its own copy. Example : `#pragma omp parallel private(i)`
- The `shared` attribute : the data is shared among all the threads. It is accessible (and non-protected) by all the threads simultaneously. Example : `#pragma omp parallel shared(array)`

The data sharing-attributes `firstprivate` and `lastprivate`

Syntax

These clauses determines the attributes of the variables within a parallel region:

```
1 firstprivate(list1) lastprivate(list2)
```

- The `firstprivate` like `private` but initialized to the value before the parallel region
- The `lastprivate` like `private` but the value is updated after the parallel region

The `reduction(...)` clause (Exercice)

How to deal with

```
vec = (int*) malloc (size_vec*sizeof(int));
global_sum = 0;
for (i=0;i<size_vec;i++){
   global_sum += vec[i];
}
```

A solution with the `reduction(...)` clause

```
vec = (int*) malloc (size_vec*sizeof(int));
global_sum = 0;
#pragma omp parallel for reduction(+:global_sum)
   for (i=0;i<size_vec;i++){
      global_sum += vec[i];
   }
```

But other solutions exist !

Worksharing constructs

Worksharing constructs are possible in four "flavours" :

- `sections` construct
- `single` construct
- `master` construct
- `workshare` construct (only in Fortran)

The sections construct

Syntax

```
1 #pragma omp [parallel] sections [clause]
2 {
3    #pragma omp section
4    {
5       code_block
6    }
7 }
```

where *clause* is one of the following:

- `private(`*list*`)`, `firstprivate(`*list*`)`, `lastprivate(`*list*`)` : data scoping
- `reduction({` *operator* | *intrinsic_procedure_name* `} :` *list*`)` : data scoping
- Each `section` within a `sections` construct is assigned to one and only one thread

A sections construct example (Exercice

Example

```
#pragma omp parallel
#pragma omp sections
{
  #pragma omp section
       do something from a thread
  #pragma omp section
       do something from another thread
  #pragma omp section
       do something from another thread
}
```

The `single` construct

Syntax

```
1 #pragma omp single [clause[[,] clause] ...]
2 {
3    structured-block
4 }
```

where *clause* is one of the following:

- `private(`*list*`)`, `firstprivate(`*list*`)`

Only one thread (usualy the master thread) executes the `single` region. The others wait for completion, except if the `nowait` clause has been activated

The master construct

- Same as single, but only the master thread execute the section

```
1  #pragma omp parallel default(shared)
2  {
3  ...
4     #pragma omp master
5     {
6        print *, "I am the master"
7     }
8  ...
9  }
```

The `workshare` construct (**only Fortran**)

Syntax

```
1 !$omp workshare
2    structured-block
3 !$omp end workshare [nowait]
```

The enclosed structured block must consist of only the following:

- array and scalar assignements
- `FORALL` statements and constructs
- `WHERE` statements and constructs
- `atomic` constructs
- `critical` constructs
- `parallel` constructs

A workshare construct example

```
1  !$omp parallel shared(A,B) private(i,j)
2  !$omp workshare
3  forall (i=1:N, j=1:N, A(i,j).NE.0.)
4     A(i,j) = 3*A(i,j)/.23 + 6.
5     B(i,j) = A(i,j) ** 2
6  end forall
7  !$omp end workshare
8  !$omp end parallel
```

The `for` directive

Parallelization of the next loop

Syntax

```
1 #pragma omp for [clause[[,] clause] ... ]
2 {
3    for-loop
4 }
```

where *clause* is one of the following:

- `schedule(`*kind[, chunk_size]*`)`
- `collapse(`*n*`)`
- `ordered`
- `private(`*list*`)`, `firstprivate(`*list*`)`, `lastprivate(`*list*`)`,`reduction()`

The `schedule` clause

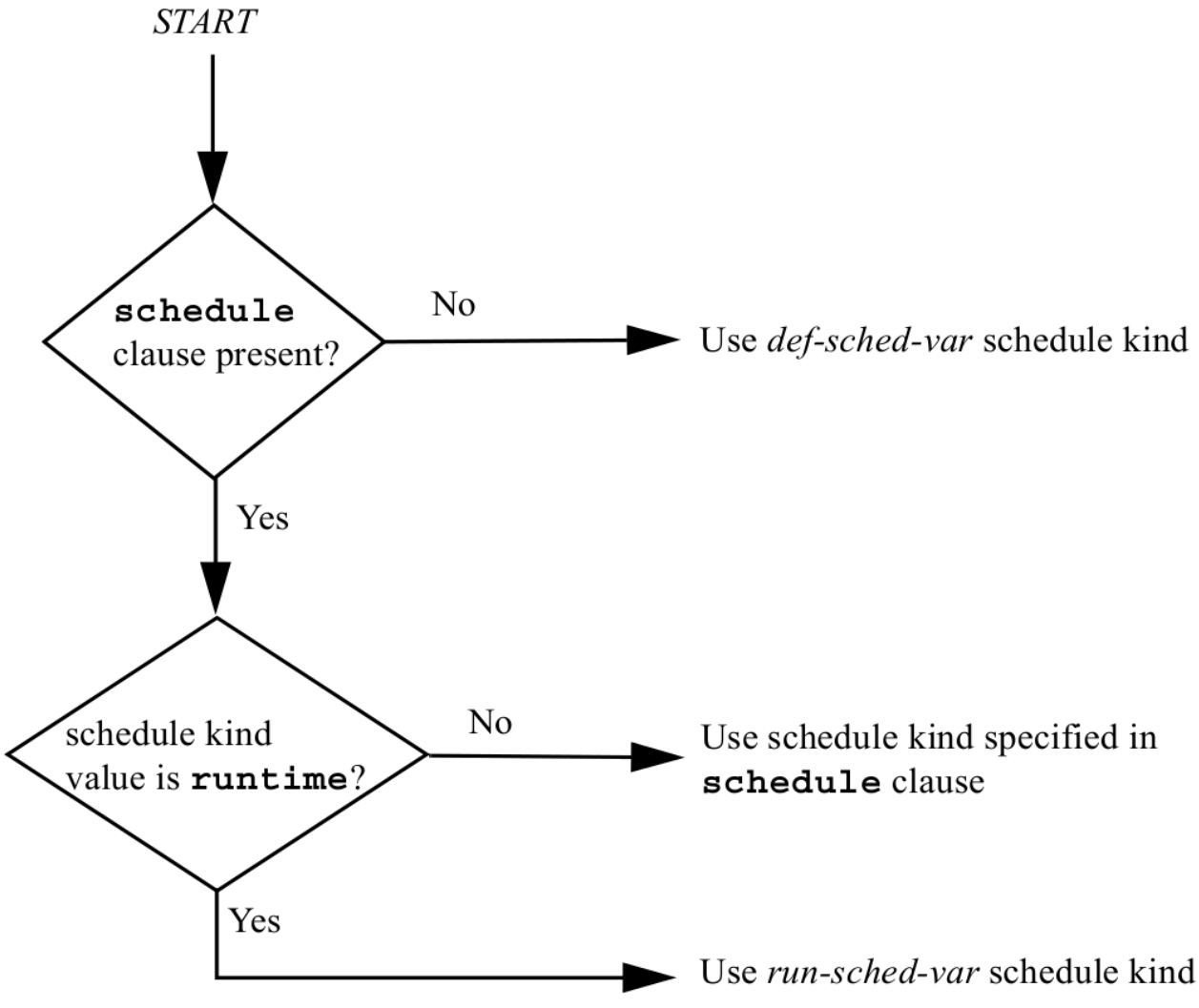
Load-balancing

clause	**behavior**
schedule(static [, chunk_size])	iterations divided in chunks sized *chunk_size* assigned to threads in a round-robin fashion. If *chunk_size* not specified system decides.
schedule(dynamic [, chunk_size])	iterations divided in chunks sized *chunk_size* assigned to threads when they request them until no chunk remains to be distributed. If *chunk_size* not specified default is 1.

The `schedule` clause

clause	**behavior**
schedule(guided [, chunk_size])	iterations divided in chunks sized *chunk_size* assigned to threads when they request them. Size of chunks is proportional to the remaining unassigned chunks. If *chunk_size* not specified default is 1.
schedule(auto)	The decisions is delegated to the compiler and/or the runtime system
schedule(runtime)	The decisions is delegated to the runtime system

The `schedule` clause



A parallel `for` example

How to...

... parallelize the dense matrix multiplication $C = AB$ (triple for loop $C_{ij} = C_{ij} + A_{ik}B_{kj}$). What happens using different `schedule` clauses ?)

A parallel `for` example

```
1   #pragma omp parallel shared(A,B,C) private(i,j,k,
        myrank)
2   {
3      myrank=omp_get_thread_num();
4      mysize=omp_get_num_threads();
5      chunk=(N/mysize);
6      #pragma omp for schedule(static, chunk)
7      for (i=0;i<N;i++){
8         for (j=0;j<N;j++){
9            for (k=0;k<N;k++){
10              C[i][j]=C[i][j] + A[i][k]*B[k][j];
11           }
12        }
13     }
14  }
```

Loop order is important (see slide # 73)

A parallel `for` example

```
vkeller@mathicsepc13:~$ export OMP_NUM_THREADS=1
vkeller@mathicsepc13:~$ ./a.out
 [DGEMM] Compute time [s]    : 0.33388209342956
 [DGEMM] Performance  [GF/s]: 0.59901385529736
 [DGEMM] Verification        : 2000000000.00000
vkeller@mathicsepc13:~$ export OMP_NUM_THREADS=2
vkeller@mathicsepc13:~$ ./a.out
 [DGEMM] Compute time [s]    : 0.18277192115783
 [DGEMM] Performance  [GF/s]: 1.09425998661625
 [DGEMM] Verification        : 2000000000.00000
vkeller@mathicsepc13:~$ export OMP_NUM_THREADS=4
vkeller@mathicsepc13:~$ ./a.out
 [DGEMM] Compute time [s]    : 9.17780399322509E-002
 [DGEMM] Performance  [GF/s]: 2.17917053085506
 [DGEMM] Verification        : 2000000000.00000
```

The `collapse` clause

Intel view

Use the collapse clause to increase the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread.

You can improve performance by avoiding use of the collapsed-loop indices (if possible) inside the collapse loop-nest (since the compiler has to recreate them from the collapsed loop-indices using divide/mod operations AND the uses are complicated enough that they don't get dead-code-eliminated as part of compiler optimizations)

A collapse directive example

```
1  #pragma omp parallel for collapse(2) shared(A)
       private(k,l)
2      for (k=0;k<kmax;k++) {
3          for (l=0;l<lmax;l++){
4              do_some_work(&A,k,l,N);
5          }
6      }
```

where do_some_work(A,k,l,N) looks like:

```
1      for(i=0;i<N;i++) {
2          for (j=0;j<N;j++) {
3              A[i][j] = A[i][j]*s+A[i][j]*t
4          }
5      }
```

A `collapse` directive example [output]

Here we compare the collapsed result with the standard parallel loop (on `k`)

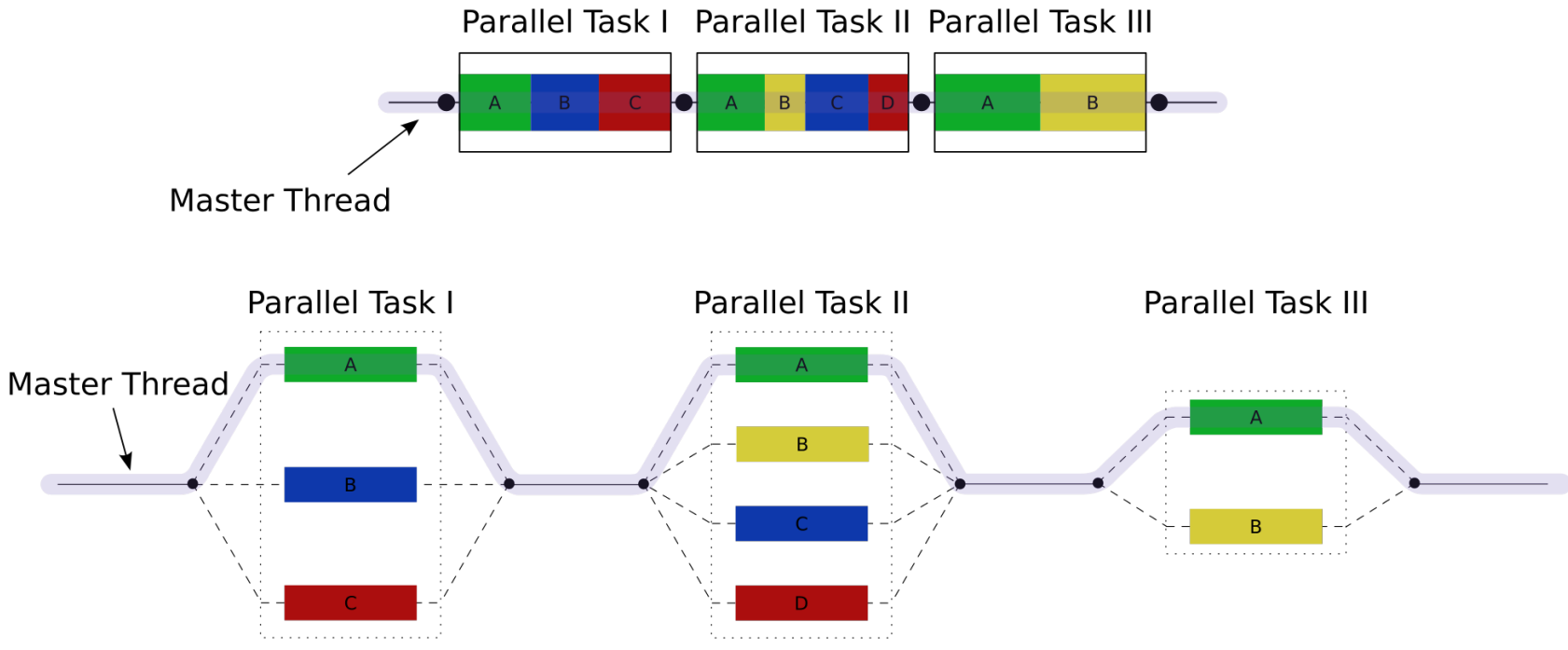
OMP_NUM_THREADS	**1**	**2**	**4**	**8**	**16**
standard // loop	2.17	1.69	1.69	1.44	1.22
collapsed(2) // loop	2.13	1.60	1.02	0.83	0.70

Table : `N`=20000, time is in seconds

Remark

It is mandatory that the $n-$collapsed loops are perfectly nested and with a rectangular shape (nothing like `do i=1,N ... do j=1,f(i)`) and that their upper limits are "small".

The `task` directive



Source : wikipedia.org

The `task` directive

What is an OpenMP task ?

- Offers a solution to parallelize irregular problems (unbounded loops, recursives, master/slave schemes, etc..)
- OpenMP tasks are composed of
  - **code** : what will be executed
  - **data** : initialized at task creation time
  - **ICV's** : specific task's ICV's

Synchronization

- All tasks created by a thread of a team are garanteed to be completed at thread exit (end of block)
- Within a task group, it is possible to synchronize through `#pragma omp taskwait`

The `task` directive

Syntax

```
1 #pragma omp task [clause[[,] clause] ...]
2 {
3   structured-block
4 }
```

where *clause* is one of the following:

- `if(`*scalar-logical-expression*`)`
- `final(`*scalar-logical-expression*`)`
- `untied`
- `default (private | firstprivate | shared | none)`
- `mergeable`
- `private(`*list*`)`, `firstprivate(`*list*`)`, `shared(`*list*`)`

The `task` directive

Execution model

- A task `t` is executed by the thread `T` of the team that generated it. Immediately or not (depends on the implementation)
- A thread `T` can suspend/resume/restart a task `t`
- Tasks are **tied** by default:
  - tied tasks are executed by the same thread
  - tied tasks have scheduling restrictions (deterministic creation, synchronization, destruction)
- It is possible to untie tasks with the directive `untied`

A `task` directive (stupid) example

You probably know this popular example to compute the `n` first Fibonacci numbers (Fib = [1,1,2,3,5,8,13,21,34,..]):

```
1  int fibonacci(int n){
2     int x,y;
3     if (n < 2) {
4        return n;
5     } else {
6        x = fibonacci(n-1);
7        y = fibonacci(n-2);
8        return (x+y);
9     }
10 }
```

A `task` directive (stupid) example (Exercise)

Idea

The idea is using the `#pragma omp tasks` construct by spawning a new task whenever the recursive function is called.

```
! start the following call in a task
   x=fibonacci(n-1)
! start the following call in a task
   y=fibonacci(n-2)
! a synchronization must be done here !
   fibonacci = (x+y)
```

Warning: number of tasks and number of physical cores

Pay attention to the number of tasks with respect to the number of cores you have on the node (here 48).

A `task` directive (stupid) example

```
1  int fibonacci(int n){
2     int x,y;
3     if (n < 2) {
4        return n;
5     } else {
6  #pragma omp task shared(x)
7        x = fibonacci(n-1);
8  #pragma omp task shared(x)
9        y = fibonacci(n-2);
10 #pragma omp taskwait
11       return (x+y);
12    }
13 }
```

A `task` directive example [output]

```
vkeller@mathicsepc13:~$ ./ex12 45
```

Warning !

Waaaaaaaaaaaaaaaaaaay too long : too many tasks are created.

A `task` directive (stupid) example with cutoff

```
1  int fibonacci(int n, int level, int cutoff){
2     int x,y;
3     if (n < 2) {
4        return n;
5     } else if (level < cutoff) {
6  #pragma omp task shared(x)
7        x = fibonacci(n-1, level+1,cutoff);
8  #pragma omp task shared(x)
9        y = fibonacci(n-2, level+1,cutoff);
10 #pragma omp taskwait
11       return (x+y);
12    } else {
13       x = fibonacci(n-1);
14       y = fibonacci(n-2);
15       return (x+y);
16    }
17 }
```

A `task` directive (stupid) example [output]

OMP_NUM_THREADS	**1**	**2**	**4**	**8**	**16**
sequential	27.4	27.4	27.4	27.4	27.4
without cutoff	27.4	>>60	>>60	>>60	>>60
with cutoff (level=10)	27.4	14.5	7.4	4.1	3.9
DO loop	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}

Table : `N`=45, time is in seconds

Remark

We get a beautiful speedup by cutting off the tree search. But another simplier algorithm (do loop) performs by 6 orders of magnitude.

Tasks remarks

- targeted for many-cores co-processors (like Intel Phi)
- Can be used to solve non symetrical problems and algorithms
- Avoid them on a multi-core CPU

Synchronization

Synchronization constructs

Those directives are sometimes mandatory:

- `master` : region is executed by the master thread only
- `critical` : region is executed by only one thread at a time
- `barrier` : all threads must reach this directive to continue
- `taskwait` : all tasks and childs must reach this directive to continue
- `atomic (read | write | update | capture)` : the associated storage location is accessed by only one thread/task at a time
- `flush` : this operation makes the thread's temporary view of memory consistent with memory
- `ordered` : a structured block is executed in the order of the loop iterations

Nesting regions

Nesting

It is possible to include parallel regions in a parallel region (i.e. nesting) under restrictions (cf. sec. 2.10, p.111, *OpenMP: Specifications ver. 3.1*)

Runtime Library routines

Usage

- The functions/subroutines are defined in the lib `libomp.so`. Don't forget to include `#include <omp_lib.h>`
- These functions can be called anywhere in your programs

Runtime Library routines

General purpose routines

routine	behavior
`omp_set_num_threads` `omp_get_num_threads`	sets/gets number of threads to be used for subsequent parallel regions that do not specify a num_threads clause by setting the value of the first element of the nthreads-var ICV of the current task
`omp_get_max_threads`	returns an upper bound on the number of threads that could be used to form a new team if a parallel region without a num_threads clause were encountered after execution returns from this routine

Runtime Library routines

General purpose routines

routine	behavior
`omp_get_thread_num`	returns the thread number, within the current team, of the calling thread.
`omp_get_num_procs`	returns the number of processors available to the program.
`omp_in_parallel`	returns true if the call to the routine is enclosed by an active parallel region ; otherwise, it returns false
`omp_set_dynamic` `omp_get_dynamic`	gets/sets the dynamic adjustment of the number of threads available for the execution of subsequent parallel regions by getting/setting the value of the dyn-var ICV.

Runtime Library routines

General purpose routines

routine	behavior
`omp_set_nested` `omp_get_nested`	gets/sets nested parallelism, by getting/setting the nest-var ICV.
`omp_set_schedule` `omp_get_schedule`	gets/sets the schedule that is applied when runtime is used as schedule kind, by getting/setting the value of the run-sched-var ICV.
`omp_get_thread_limit`	returns the maximum number of OpenMP threads available to the program.

Runtime Library routines

General purpose routines

routine	behavior
`omp_set_max_active_levels` `omp_get_max_active_levels`	limits the number of nested active parallel regions, by getting/setting the max-active-levels-var ICV.
`omp_get_level`	returns the number of nested parallel regions enclosing the task that contains the call.
`omp_get_ancestor_thread_num`	returns, for a given nested level of the current thread, the thread number of the ancestor or the current thread

Runtime Library routines

General purpose routines

routine	behavior
`omp_get_team_size`	returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs
`omp_get_active_level`	returns the number of nested, active parallel regions enclosing the task that contains the call.
`omp_in_final`	returns true if the routine is executed in a final task region; otherwise, it returns false.

Remark on lock routines

The following routines are rarely used. They are mentionned for exhaustivity purposes

routine	behavior
`omp_init_lock`	initializes a simple lock
`omp_destroy_lock`	uninitializes a simple lock
`omp_set_lock`	waits until a simple lock is available and then sets it
`omp_unset_lock`	unsets a simple lock
`omp_test_lock`	tests a simple lock, and sets it if it is available

Runtime Library routines

Remark on lock nestable routines

The following routines are rarely used. They are mentionned for exhaustivity purposes

routine	behavior
`omp_init_nest_lock`	initializes a nestable lock
`omp_destroy_nest_lock`	uninitializes a nestable lock
`omp_set_nest_lock`	waits until a nestable lock is available and then sets it
`omp_unset_nest_lock`	unsets a nestable lock
`omp_test_nest_lock`	tests a nestable lock, and sets it if it is available

Runtime Library routines

Timing routines

routine	behavior
`omp_get_wtime`	returns elapsed wall clock time in seconds.
`omp_get_wtick`	returns the precision of the timer used by `omp_get_wtime`

Environment variables

Usage

- Environment variables are used to set the ICVs variables
- under `csh` : `setenv OMP_VARIABLE "its-value"`
- under `bash` : `export OMP_VARIABLE="its-value"`

Environment variables

variable	**what for ?**
`OMP_SCHEDULE`	sets the run-sched-var ICV that specifies the runtime schedule type and chunk size. It can be set to any of the valid OpenMP schedule types.
`OMP_NUM_THREADS`	sets the nthreads-var ICV that specifies the number of threads to use in parallel regions
`OMP_DYNAMIC`	sets the dyn-var ICV that specifies the dynamic adjustment of threads to use for parallel regions.

Environment variables

variable	**what for ?**
`OMP_PROC_BIND`	sets the bind-var ICV that controls whether threads are bound to processors
`OMP_NESTED`	sets the nest-var ICV that enables or disables nested parallelism
`OMP_STACKSIZE`	sets the stacksize-var ICV that specifies the size of the stack for threads created by the OpenMP implementation.
`OMP_WAIT_POLICY`	sets the wait-policy-var ICV that controls the desired behavior of waiting threads.

Environment variables

variable	**what for ?**
`OMP_MAX_ACTIVE_LEVELS`	sets the max-active-levels-var ICV that controls the maximum number of nested active parallel regions.
`OMP_THREAD_LIMIT`	sets the thread-limit-var ICV that controls the maximum number of threads participating in the OpenMP program.

The apparent "easiness" of OpenMP

"Compared to MPI, OpenMP is much easier"

In the reality

- Parallelization of a non-appropriate algorithm
- Parallelization of an unoptimized code
- Race conditions in shared memory environment
- Memory coherence
- Compiler implementation of the OpenMP API
- (Much) more threads/tasks than your machine can support

Parallelization of an unoptimized code

(... or parallelization of a non-appropriate algorithm)

Back in 1991, David H. Bailey from Lawrence Berkeley National Laboratory released a famous paper in the Supercomputing Review: *"Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers"*.

Number 6 was: *Compare your results against scalar, unoptimized code on Crays.*

The compiler issue

The choice of the compiler is **very** important. Different version can lead to different performance on the same code.

```
1       for (i=0;i<N;i++){
2          for (j=0;j<N;j++){
3             for (k=0;k<N;k++){
4                C[i][j]=C[i][j] + A[i][k]*B[k][j];
5             }
6          }
7       }
```

We change the loop order and the compiler version: `i-j-k`, `i-k-j`, `j-i-k`, `j-k-i`, `k-i-j`, `k-j-i` and the two compilers: `gcc-4.8.2`, `icc 15`.

gcc version 4.8.2, N=2000

```
gcc -O3 -ftree-vectorize dgemm.c -lgsl
    -lgslcblas -lm -o dgemm
```

```
ijk          0.133   [GF/s]
ikj          2.128   [GF/s]
jik          0.132   [GF/s]
jki          0.064   [GF/s]
kij          1.914   [GF/s]
kji          0.064   [GF/s]
DGEMM OPT    1.781   [GF/s]
```

Remark

"DGEMM OPT" stands for the GNU Scientific Library (gsl)

icc version 15, N=2000

```
icc -DMKL_IN_USE -DMKL_ILP64 -openmp -I${MKLROOT}/include
    -O3 -xHost dgemm.c  -L${MKLROOT}/lib/intel64
    -lmkl_intel_ilp64 -lmkl_core -lmkl_intel_thread
    -lpthread -lm -o dgemm
```

```
ijk          2.746   [GF/s]
ikj          2.784   [GF/s]
jik          2.586   [GF/s]
jki         10.233   [GF/s]
kij          2.747   [GF/s]
kji          9.815   [GF/s]
DGEMM OPT   27.593   [GF/s]
```

Remark

"DGEMM OPT" stands for the Intel Math Kernel Library (version 11.2).

“OpenMP-ization” strategy

- **STEP 1** : Optimize the sequential version:
  - Choose the best algorithm
  - “Help the (right) compiler”
  - Use the existing optimized scientific libraries
- **STEP 2** : Parallelize it:
  - Identify the bottlenecks (heavy loops)
  - “auto-parallelization” is rarely the best !

Goal

Debugging - Profiling - Optimization cycle. Then parallelization !

Tricks and tips

- **Algorithm**: choose the "best" one
- **cc-NUMA**: no (real) support from OpenMP side (but OS). A multi-CPU machine is not a real shared memory architecture
- **False-sharing**: multiple threads write in the same cache line
- **Avoid barrier**. This is trivial. Bus sometimes you can't
- **Small number of tasks**. Try to reduce the number of forked tasks
- **Asymetrical problem**. OpenMP is well suited for symetrical problems, even if tasks can help
- **Tune the schedule**: types, chunks...
- **Performance expectations**: a theoretical analysis using the simple Amdahl's law can help
- **Parallelization level**: coarse (SPMD) or fine (loop) grain ?

OpenMP Thread affinity

Show and set affinity with Intel executable

By setting the `export KMP_AFFINITY=verbose,SCHEDULING` you are able to see where the OS pin each thread

Show and set affinity with GNU executable

By setting the
`export GOMP_CPU_AFFINITY=verbose,SCHEDULING` you are able to see where the OS pin each thread

OpenMP Thread affinity with compact

```
vkeller@mathicsepc13:~$ export KMP_AFFINITY=verbose,compact
vkeller@mathicsepc13:~$ ./ex10
OMP: Info #204: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #202: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11 info
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
OMP: Info #156: KMP_AFFINITY: 16 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 4 cores/pkg x 2 threads/core (8 total cores)
OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 8 maps to package 0 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 9 maps to package 0 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 10 maps to package 0 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 11 maps to package 0 core 10 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 4 maps to package 1 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 12 maps to package 1 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 5 maps to package 1 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 13 maps to package 1 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 6 maps to package 1 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 14 maps to package 1 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 7 maps to package 1 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 15 maps to package 1 core 10 thread 1
OMP: Info #144: KMP_AFFINITY: Threads may migrate across 1 innermost levels of machine
OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set {0,8}
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {0,8}
OMP: Info #147: KMP_AFFINITY: Internal thread 2 bound to OS proc set {1,9}
OMP: Info #147: KMP_AFFINITY: Internal thread 3 bound to OS proc set {1,9}
 [DGEMM]   Compute time [s]    :   0.344645023345947
 [DGEMM]   Performance  [GF/s]:   0.580307233391397
 [DGEMM]   Verification        :    2000000000.00000
```

OpenMP Thread affinity with scatter

```
vkeller@mathicsepc13:~$ export KMP_AFFINITY=verbose,scatter
vkeller@mathicsepc13:~$ ./ex10
OMP: Info #204: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #202: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11 info
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
OMP: Info #156: KMP_AFFINITY: 16 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 4 cores/pkg x 2 threads/core (8 total cores)
OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 8 maps to package 0 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 9 maps to package 0 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 10 maps to package 0 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 11 maps to package 0 core 10 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 4 maps to package 1 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 12 maps to package 1 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 5 maps to package 1 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 13 maps to package 1 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 6 maps to package 1 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 14 maps to package 1 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 7 maps to package 1 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 15 maps to package 1 core 10 thread 1
OMP: Info #144: KMP_AFFINITY: Threads may migrate across 1 innermost levels of machine
OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set {0,8}
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {4,12}
OMP: Info #147: KMP_AFFINITY: Internal thread 2 bound to OS proc set {1,9}
OMP: Info #147: KMP_AFFINITY: Internal thread 3 bound to OS proc set {5,13}
 [DGEMM]   Compute time [s]    :   0.204235076904297
 [DGEMM]   Performance  [GF/s]:   0.979263714301724
 [DGEMM]   Verification        :    2000000000.00000
```

OpenMP Thread affinity with explicit (a kind of pining)

```
vkeller@mathicsepc13:~$ export KMP_AFFINITY='proclist=[0,2,4,6],explicit',verbose
vkeller@mathicsepc13:~$ ./ex10
OMP: Info #204: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #202: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11 info
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
OMP: Info #156: KMP_AFFINITY: 16 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 4 cores/pkg x 2 threads/core (8 total cores)
OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 8 maps to package 0 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 9 maps to package 0 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 10 maps to package 0 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 11 maps to package 0 core 10 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 4 maps to package 1 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 12 maps to package 1 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 5 maps to package 1 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 13 maps to package 1 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 6 maps to package 1 core 9 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 14 maps to package 1 core 9 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 7 maps to package 1 core 10 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 15 maps to package 1 core 10 thread 1
OMP: Info #144: KMP_AFFINITY: Threads may migrate across 1 innermost levels of machine
OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set {0,8}
OMP: Info #147: KMP_AFFINITY: Internal thread 3 bound to OS proc set {6,14}
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {2,10}
OMP: Info #147: KMP_AFFINITY: Internal thread 2 bound to OS proc set {4,12}
 [DGEMM]   Compute time [s]    :   0.248908042907715
 [DGEMM]   Performance  [GF/s]:   0.803509591990774
 [DGEMM]   Verification        :    2000000000.00000
```

A simple conclusion ...

The message

FIRST OPTIMIZE ON ONE CORE, THEN PARALLELIZE (the right algorithm)

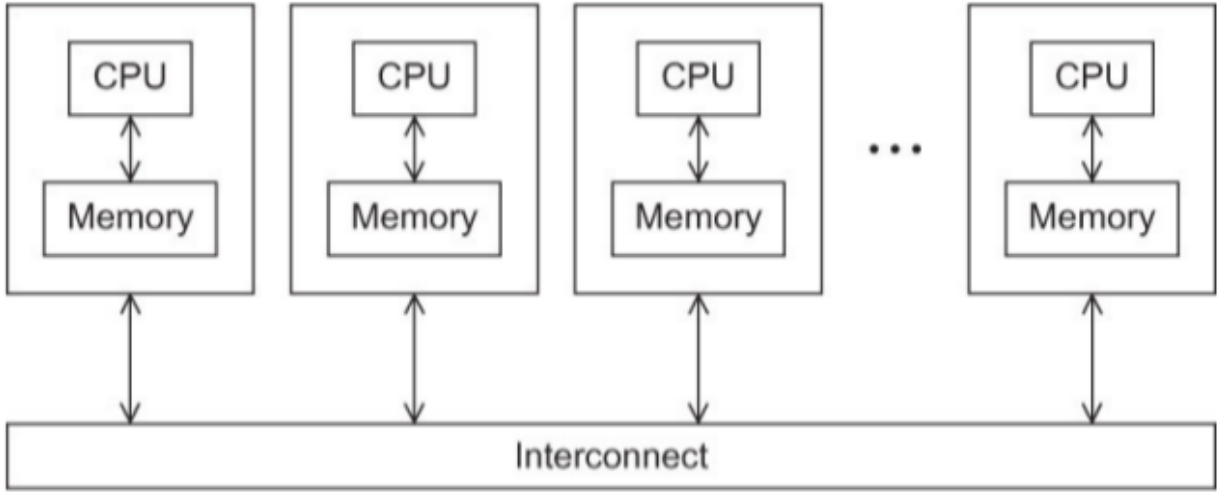
What's new with OpenMP 4.0 ?

- Support for new devices (`Intel Phi`, `GPU`,...) with `omp target`. Offloading on those devices.
- Hardware agnostic
- League of threads with `omp teams` and distribute a loop over the team with `omp distribute`
- SIMD support for vectorization `omp simd`
- Task management enhancements (cancelation of a task, groups of tasks, task-to-task synchro)
- Set thread affinity with a more standard way than `KMP_AFFINITY` with the concepts of `places` (a thread, a core, a socket), `policies` (spread, close, master) and `control settings` the new clause `proc_bind`

What will we learn today ?

- The distributed memory programming paradigm MPI
- Point-to-Point communications
- Collective communications
- Synchronizations

Reminder



Goals and scope of MPI

- Provide a source-code portability
- efficiency across different architectures
- easier debugging
- parallel I/O

MPI

- Run multiple instances of the same program : `"mpiexec -n p myApp myArgs"` starts `p` instances of the program `"myApp myArgs"`
- Instances exchange information by sending messages to each other
- Communications take place within a `communicator` : a set of processes indexed from 0 to *communicatorSize* − 1. A special communicator named `MPI_COMM_WORLD` contains all the processes

Hello World

```
1  int main(int argc, char *argv[]){
2     int size, rank;
3     MPI_Init(&argc, &argv);
4     MPI_Comm_size(MPI_COMM_WORLD, &size);
5     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6
7     printf("I'm process %d out of %d\n", rank, size);
8
9     MPI_Finalize();
10 }
```

- Compile with `mpicc hello.c -o hello`
- Run with `mpiexec -n 4 ./hello`

Types of communications in MPI

- Point-to-Point (One-to-One)
- Collectives (One-to-All, All-to-One, All-to-All)
- One-sided (One-to...)
- Blocking and Non-Blocking of all types

MPI_Send() and MPI_Recv()

- `MPI_Send(buf, count, datatype, destination, tag, communicator)`
- `MPI_Recv(buf, count, datatype, source, tag, communicator, status)`
- Sends (receives) `count` elements of type `datatype` to (from) `destination` (`source`) buffer `buf`.
- **Each send must be matched by a receive!**
  - You must know `source`, `tag`, `communicator` and size (`count * sizeof(datatype)`) of incoming message
  - If you don't know, use `MPI_Probe` / `MPI_Iprobe` to get that information

Mismatches cause race conditions or deadlocks

ping.c

```
1  int main(int argc, char *argv[]) {
2     int myrank, mysize;
3     int buf[100];
4     MPI_Status status;
5     MPI_Init(&argc, &argv);
6     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
7     MPI_Comm_size(MPI_COMM_WORLD, &mysize);
8     if (myrank == 0) {
9        MPI_Send(buf, 100, MPI_INT, 1, 0, MPI_COMM_WORLD
             );
10    } else {
11       MPI_Recv(buf, 100, MPI_INT, 0, 0, MPI_COMM_WORLD
             ,&status);
12    }
13    MPI_Finalize();
14 }
```

watch out for deadlocks !

- ping.c runs ok with 2 processes :

Process 0		Process 1
send(1,0)	$\longrightarrow$	recv(0,0)
finalize()		finalize()

- Deadlock if more than two processes :

Process 0		Process 1	Process 2
send(1,0)	$\longrightarrow$	recv(0,0)	recv(0,0)
finalize()		finalize()	

ping_correct.c

```
1  int main(int argc, char *argv[]) {
2     int myrank, mysize;
3     int buf[100];
4     MPI_Status status;
5     MPI_Init(&argc, &argv);
6     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
7     MPI_Comm_size(MPI_COMM_WORLD, &mysize);
8     if (rank == 0) {
9        MPI_Send(buf, 100, MPI_INT, 1, 0, MPI_COMM_WORLD
             );
10    } else if (myrank==1) {
11       MPI_Recv(buf, 100, MPI_INT, 0, 0, MPI_COMM_WORLD
             ,&status);
12    }
13    MPI_Finalize();
14 }
```

Blocking point-to-point communications

- `MPI_Send` : Returns when buffer can be reused
- `MPI_Ssend` : Returns when other end called matching recv
- `MPI_Recv` : Returns when message has been received
- `MPI_Sendrecv` : Sends and receive within the same call to avoid deadlocks
- `MPI_Bsend` : like `MPI_Send` but with a user-provided buffer size
- `MPI_Rsend` : sends only if the matching receive is already posted
- `MPI_Sendrecv_replace` : sends, receive and replace buffer values using only one buffer

Non-Blocking point-to-point communications

- `MPI_Isend` and `MPI_Irecv` : Do not wait for message to be buffered send/recv. Fills an additional `MPI_Request` parameter that identifies the request
- Do not use / modify / delete buffers until request completed
- Wait calls block until request(s) completed :
  - `MPI_Wait(request, status)`
  - `MPI_Waitall(count, array_of_requests, array_of_statuses)`
- `MPI_Issend` : Non-blocking version of `MPI_Ssend`
- `MPI_Ibsend` : Non-blocking version of `MPI_Bsend`
- `MPI_Irsend` : Non-blocking version of `MPI_Rsend`

Non-Blocking point-to-point communications (cont'd)

Playing with `MPI_Request`

- `MPI_Waitsome` : Waits for an MPI request to complete
- `MPI_Waitany` : Waits for any specified MPI Request to complete
- `MPI_Test` : Tests for the completion of a request
- `MPI_Testall` : Tests for the completion of all previously initiated requests
- `MPI_Testany` : Tests for completion of any previously initiated requests
- `MPI_Testsome` : Tests for some given requests to complete

iping.c

```
1  int main(int argc, char *argv[]) {
2      int rank;
3      int buf[100];
4      MPI_Request request;
5      MPI_Status status;
6      MPI_Init(&argc, &argv);
7      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8      if (rank == 0)
9          MPI_Isend(buf, 100, MPI_INT, 1, 0,MPI_COMM_WORLD
               , &request);
10     else if (rank == 1)
11         MPI_Irecv(buf, 100, MPI_INT, 0, 0,MPI_COMM_WORLD
               , &request);
12     MPI_Wait(&request, &status);
13     MPI_Finalize();
14 }
```

exchange.c

Process 0 and 1 exchange the content of their buffer with non-blocking

```
1  if (rank == 0) {
2     MPI_Isend(buf1, 10, MPI_INT, 1, 0, MPI_COMM_WORLD,
          &request);
3     MPI_Recv(buf2, 10, MPI_INT, 1, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
4  }else if (rank == 1){
5     MPI_Isend(buf1, 10, MPI_INT, 0, 0, MPI_COMM_WORLD,
          &request);
6     MPI_Recv(buf2, 10, MPI_INT, 0, 0,MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
7  }
8
9  MPI_Wait(&request, &status);
10 memcpy(buf1, buf2, 10*sizeof(int));
```

exchange_sendrecv.c

Process 0 and 1 exchange the content of their buffer with sendrecv()

```
1 if (rank == 0) {
2     MPI_Sendrecv(buf1, 10, MPI_INT, 1, 0, buf2, 10,
          MPI_INT, 1, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
3 } else if (rank == 1) {
4     MPI_Sendrecv(buf1, 10, MPI_INT, 0, 0, buf2, 10,
          MPI_INT, 0, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
5 }
6
7 memcpy(buf1, buf2, 10*sizeof(int));
```

exchange_sendrecv_replace.c

Process 0 and 1 exchange the content of their buffer with sendrecv_replace()

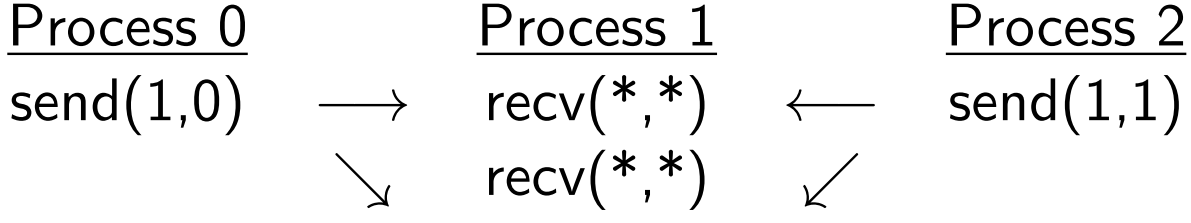
```
1 if (rank == 0){
2     MPI_Sendrecv_replace(buf1, 10, MPI_INT, 1, 0, 1,
          0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
3 } else if (rank == 1) {
4     MPI_Sendrecv_replace(buf1, 10, MPI_INT, 0, 0, 0,
          0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
5 }
```

Wildcard receives

MPI_ANY_SOURCE and MPI_ANY_TAG are wildcards :

```
1  switch(rank) {
2     case 0: MPI_Send(..., 1, 0, ...); break;
3     case 1:
4        MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ...);
5        MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ...);
6        break;
7     case 2: MPI_Send(..., 1, 1, ...); break;
8  }
```

leads to :



MPI_Barrier(communicator)

Returns when all processes in communicator have joined

```
1  switch(rank) {
2     case 0:
3        MPI_Send(..., dest = 1, tag = 0, ...);
4        MPI_Barrier(MPI_COMM_WORLD); break;
5     case 1:
6        MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ...);
7        MPI_Barrier(MPI_COMM_WORLD);
8        MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ...);
             break;
9     case 2:
10       MPI_Barrier(MPI_COMM_WORLD);
11       MPI_Send(..., src = 1, tag = 1, ...); break;
12 }
```

MPI_Barrier(communicator) (cont'd)

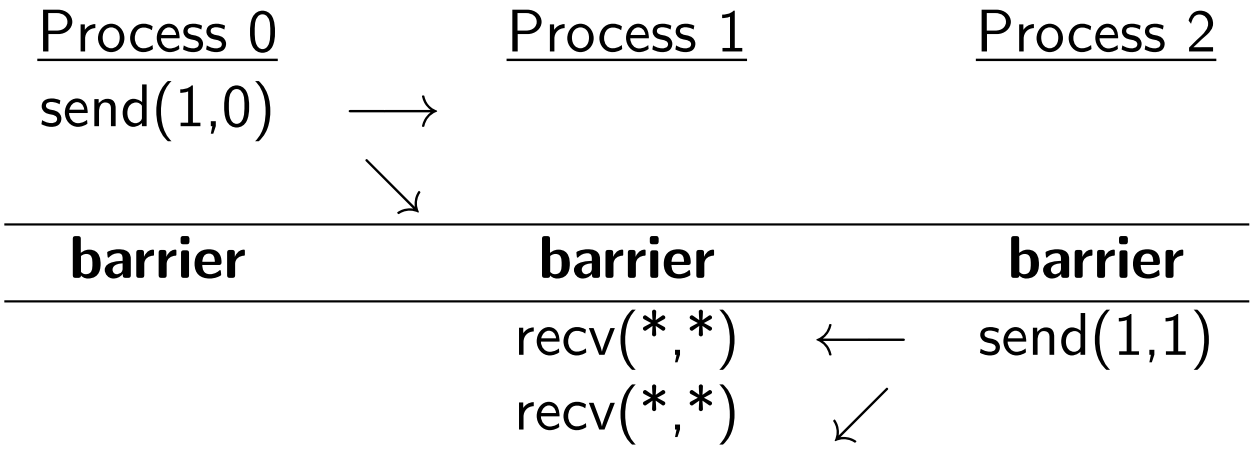
MPI_ANY_SOURCE and MPI_ANY_TAG are wildcards :

Process 0		Process 1		Process 2
send(1,0)	$\longrightarrow$	recv(*,*)		
barrier		**barrier**		**barrier**
		recv(*,*)	$\longleftarrow$	send(1,1)

Order of calls on process 1 is important

- recv – barrier – recv correct
- barrier – recv – recv message race or deadlock (depends on msg size)
- recv – recv – barrier deadlock

barrier – recv – recv



Order of calls on process 1 is important

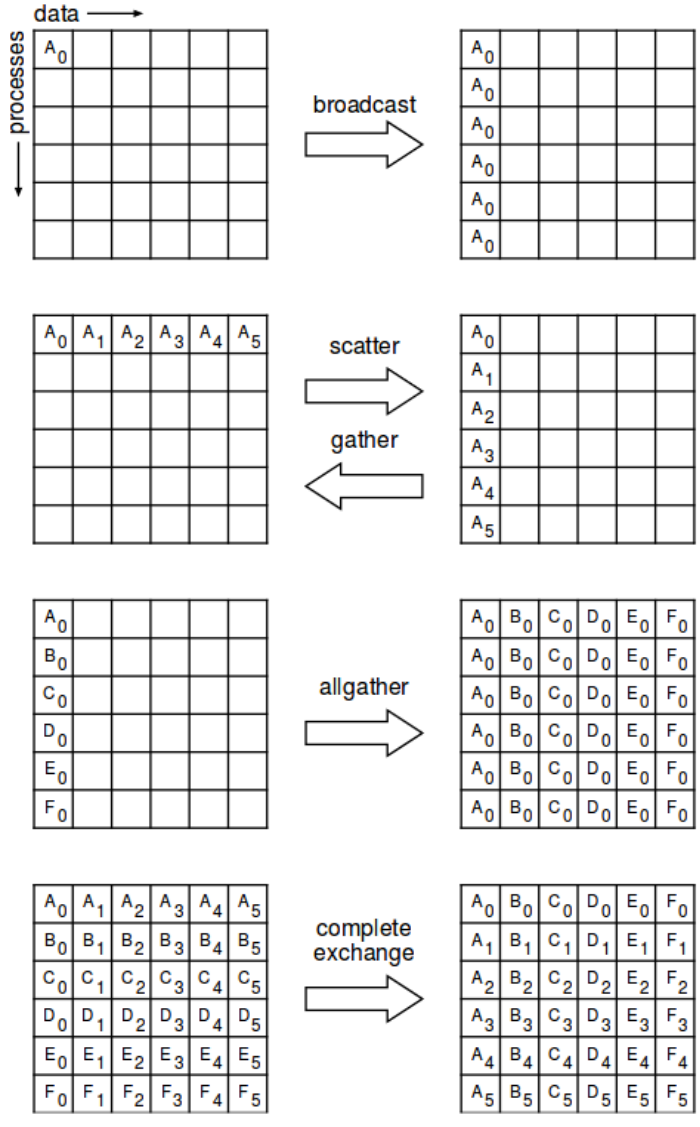
- recv – barrier – recv correct
- barrier – recv – recv message race or deadlock (depends on msg size)
- recv – recv – barrier deadlock

recv – recv – barrier

Process 0		Process 1		Process 2
send(1,0)	$\longrightarrow$			
		recv(*,0)		
		recv(2,*)		
barrier		**barrier**		**barrier**
			$\longleftarrow$	send(1,1)

Process 1 never enters the barrier since its second recv is never matched

Collective communications



Collective communications

- `MPI_Bcast` : Sends the same data to every process
- `MPI_Scatter` : Sends pieces of a buffer to every process of the communicator
- `MPI_Gather` : Retrieves pieces of data from every process
- `MPI_Allgather` : All pieces retrieved by all processes
- `MPI_Reduce` : Performs a reduction operation (`MPI_SUM`, ...) across all nodes. E.g. dot product on distributed vectors
- `MPI_Allreduce` : Result distributed to all processes
- `MPI_Alltoall` : Sends all data to all processes
- **Every process of the communicator must participate**. Parameters must match. Mismatches cause race conditions or deadlocks

Receiving image parts in order

```
1  /* Generate image parts */
2  ...
3  /* Each process sends */
4     MPI_Isend(imgPart, partSize, MPI_BYTE, 0, 0,
          MPI_COMM_WORLD, &request);
5
6  // Process 0 receives all parts into buf
7  if (rank == 0){
8     char *buf = malloc(nProcs*partSize);
9     for (int i=0; i<nProcs; i++){
10       MPI_Recv(buf + i*partSize, partSize, MPI_BYTE, i
             , 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
11    }
12
13 MPI_Wait(&request, MPI_STATUS_IGNORE);
```

Receiving image parts out-of-order

```
1  /* Generate image parts */
2  ...
3  /* Each process sends */
4     MPI_Isend(imgPart, partSize, MPI_BYTE, 0, 0,
          MPI_COMM_WORLD, &request);
5  // Process 0 receives all parts into buf
6  if (rank == 0) {
7     char *buf = malloc(nProcs*partSize);
8     MPI_Status s; int count;
9     for (int i=0; i<nProcs; i++) {
10       MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &s)
             ;
11       MPI_Get_count(&s, MPI_BYTE, &count);
12       MPI_Recv(buf + s.MPI_SOURCE*count, count,
             MPI_BYTE, s.MPI_SOURCE, s.MPI_TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
13 } } MPI_Wait(&request, MPI_STATUS_IGNORE);
```

Receiving image parts with a collective

```
1  /* Generate image parts */
2  ...
3  /* Process 0 is the root of the collective, i.e. the
       receiver of all parts */
4  int root = 0;
5  char *buf = NULL;
6  if (rank == root) /* Only the root must allocate buf
       */
7     buf = malloc(nProcs*partSize)
8
9     MPI_Gather(part, partSize, MPI_BYTE, buf, partSize
          , MPI_BYTE, root, MPI_COMM_WORLD);
```

Collectives within conditions

Avoid collective calls within conditional clauses. What happens if :

```
1  int root = 0;
2  char *buf = NULL;
3  if (rank == root) { /* Only the root must allocate
       buf */
4     buf = malloc(nProcs*partSize)
5
6     MPI_Gather(part, partSize, MPI_BYTE, buf, partSize
          , MPI_BYTE, root, MPI_COMM_WORLD);
7  }else{
8     MPI_Send(part, ... , ... , myrank, MPI_TAG);
9  }
```

Customized communicators and datatypes

- You can define your own communicators :
  - `MPI_Comm_dup` duplicates a communicator (e.g. to enable private communications within library functions)
  - `MPI_Comm_split` splits a communicator into multiple smaller communicators (useful when using 2D and 3D domain decompositions)
- You can define custom datatypes :
  - Simple structs (`MPI_Type_struct`)
  - Vectors (`MPI_Type_vector`)
  - **NO POINTERS**

Does this program terminate? (assume 2 processes)

```
1  int rank;
2  MPI_Init(&argc, &argv);
3  MPI_Comm_rank(&rank, MPI_COMM_WORLD);
4  if (rank)
5    MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
6  else
7    MPI_Recv(&rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
8
9  if (rank)
10   MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
11 else
12   MPI_Recv(&rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
13
14 MPI_Finalize();
```

Timing MPI programs

- `MPI_Wtime()` : returns a double precision floating point number, the time in seconds since some abritrary point of time in the past
- `MPI_Wtick()` : returns a double precision floating point number, the time in seconds between successive ticks of the clock
- both functions are not synchronized across the running processors.

Next time

That's all for MPI 1.0. What's new in MPI2 :

- Parallel I/O
- Onse-sided communications
- Dynamic process management

Implementation of MPI

You don't need to have always access to a cluster or a supercomputer to develop parallel applications. Here follows some implementations of MPI :

- **Linux** OpenMPI, MPICH2, IntelMPI (licensed)
- **MacOS X** OpenMPI (without Fortran for the official port, with if installing by hand), MPICH2 (via homebrew)
- **Windows** OpenMPI, MPICH1 (older version)

What will we learn today ?

- Advanced `MPI_Types`, MPI communicators and groups (MPI 1.0)
- Persistent communications (MPI 2.0)
- One-sided communications (RMA) (MPI 2.0 and MPI 3.0)
- Dynamic process management (MPI 2.0)
- Parallel I/O (MPI 2.0)
- Non-blocking collectives (MPI 3.0)

Basic MPI datatypes (C)

C datatype	**MPI datatype**
signed char	MPI_CHAR
signed short int	MPI_SHORT
signed int	MPI_INT
signed long int	MPI_LONG
unsigned char	MPI_UNSIGNED_CHAR
unsigned short int	MPI_UNSIGNED_SHORT
unsigned long int	MPI_UNSIGNED_LONG
unsigned int	MPI_UNSIGNED
float	MPI_FLOAT
double	MPI_DOUBLE
long double	MPI_LONG_DOUBLE

Basic MPI datatypes (FORTRAN)

C datatype	**MPI datatype**
INTEGER	MPI_INTEGER
REAL	MPI_REAL
REAL*8	MPI_REAL8
DOUBLE PRECISION	MPI_DOUBLE_PRECISION
COMPLEX	MPI_COMPLEX
LOGICAL	MPI_LOGICAL
CHARACTER	MPI_CHARACTER

Derived MPI datatypes

OK. That is perfect when all the data are of the same type (integers, floats, characters, etc..). But how to send a structure using MPI ?

```
1 struct {
2    int x; int y;
3    double vx; double vy;
4    float mass;
5 } particle;
6 particle p = {1,2,0.3,0.4,1.0};
7 MPI_Send(p, ...);
```

Derived MPI datatypes

- Definition of **new** datatypes by grouping basic MPI datatypes
- It is possible to group
  - data from different types
  - group non-contiguous data
- A derived datatype is defined in three steps :
  - construct the type
  - commit it to the system
  - free it

Derived MPI datatypes

- `MPI_Type_contiguous` Produces a new data type by making count copies of an existing data type.
- `MPI_Type_vector`, `MPI_Type_create_hvector` Similar to contiguous, but allows for regular gaps (stride) in the displacements
- `MPI_Type_indexed`, `MPI_Type_create_hindexed` An array of displacements of the input data type is provided as the map for the new data type.
- `MPI_Type_struct` The new data type is formed according to completely defined map of the component data types.
- `MPI_Type_extent` Returns the size in bytes of the specified data type.
- `MPI_Type_commit` Commits new datatype to the system.
- `MPI_Type_free` Deallocates the specified datatype object.

MPI_Type_struct example

```
1 struct { int a; char b; } foo;
2 MPI_Datatype newtype;
3 blen[0] = 1; indices[0] = 0; oldtypes[0] = MPI_INT;
4 blen[1] = 1; indices[1] = &foo.b - &foo; oldtypes[1]
      = MPI_CHAR;
5 blen[2] = 1; indices[2] = sizeof(foo); oldtypes[2] =
      MPI_UB;
6 MPI_Type_struct( 3, blen, indices, oldtypes, &newtype
       );
7 foo f = {1,'z'};
8 MPI_Send(&f, 1, newtype, 0, 100, MPI_COMM_WORLD );)
9 MPI_Type_free( &newtype );
```

The MPI datatype `newtype` is a structure that contains {`foo`, `sizeof(foo)`}

Pack/Unpack data

- Instead of creating a new datatype, it is possible to pack and unpack data of different types
- Less good than MPI derived datatypes in terms of memory usage and performance
- "just like a streaming"
- `int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)`
- `int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)`

Pack/Unpack data

```
1  int x; float a, int position=0;
2  char buffer[100];
3  if (myrank==0)
4     MPI_Pack(&a, 1, MPI_FLOAT, buffer, 100, &position,
           MPI_COMM_WORLD)
5     MPI_Pack(&x, 1, MPI_INT, buffer, 100, &position,
          MPI_COMM_WORLD)
6     MPI_Send(buffer, 100, MPI_PACKED, 1, 999,
          MPI_COMM_WORLD);
7  }else if (myrank==1) {
8     MPI_Recv(buffer, 1000, MPI_PACKED, 0, 999,
          MPI_COMM_WORLD, status)
9     MPI_Unpack(buffer, 100, &position, &a, 1,
          MPI_FLOAT, MPI_COMM_WORLD);
10    MPI_Unpack(buffer, 100, &position, &x, 1, MPI_INT,
           MPI_COMM_WORLD);
11 }
```

MPI Groups and Communicators

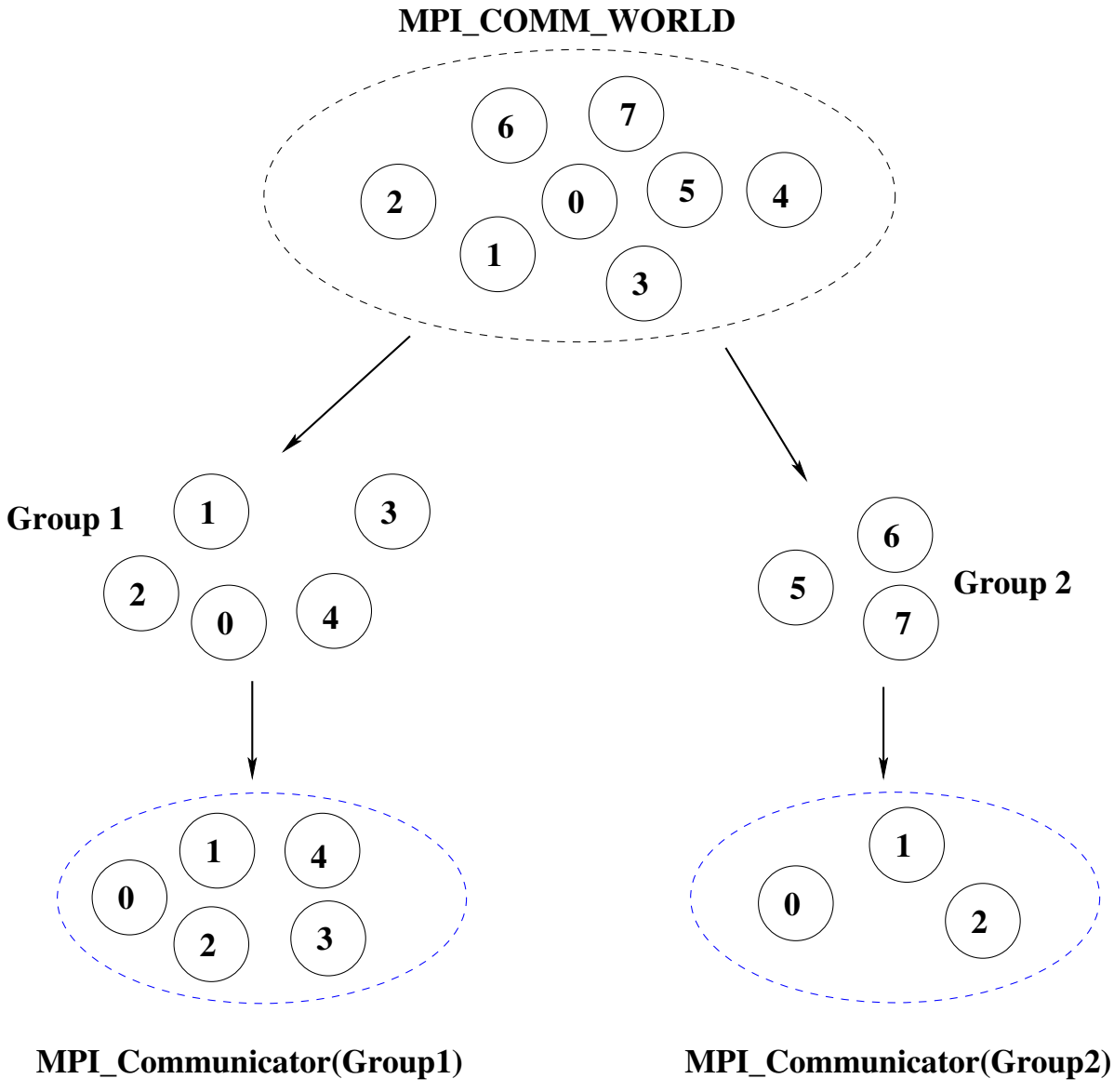
MPI Groups :

- ordered set of processes
- each process has an unique ID (rank within the group) and can belong to several different groups

MPI Communicators :

- A group of processes
- encapsulate the communications between the belonging processes
- An MPI communication can take place only with a communicator (not a group)

MPI Groups and Communicators



Create a new communicator

```
1  MPI_Init(&argc,&argv);
2  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
3  MPI_Comm_size(MPI_COMM_WORLD, &size);
4  MPI_Comm_group(MPI_COMM_WORLD, &old_g);
5  int nbr_g1 = 5;
6  ranks1 = (int*) malloc(nbr_g1*sizeof(int));
7  ranks2 = (int*) malloc((size-nbr_g1)*sizeof(int));
8  for (i=0;i<nbr_grp1;i++) ranks1[i]=i;
9  for (i=0;i<(size-nbr_g1);i++) ranks2[i]=size-i-1;
10 if (rank < nbr_g1) {
11  MPI_Group_incl(old_g,nbr_g1,ranks1,&new_g);
12 } else {
13  MPI_Group_incl(old_g,(size-nbr_g1),ranks2,&new_g);
14 }
15 MPI_Comm_create(MPI_COMM_WORLD,new_g,&new_comm);
16 MPI_Group_rank (new_g, &new_rank);
17 printf("rank %d grprank is %d \n",rank,new_rank);
18 MPI_Finalize();
```

Persistent communications

- when a same communication is repeated within a loop (e.g. exchanging neighbors in 2D Poisson)
- Improvement can be done using a persistent communication
- Using a `MPI_Request` to initiate and complete a communication
- `MPI_Send_Init()` : creates a persistent communication request for a standard mode send operation
- `MPI_Bsend_Init()` : creates a persistent communication request for a bufferd mode send operation
- `MPI_Ssend_Init()` : creates a persistent communication object for a synchronous mode send operation
- `MPI_Rsend_Init()` : creates a persistent communication object for a ready mode send operation
- `MPI_Recv_Init()` : creates a persistent communication request for a receive operation.

Persistent communications example

```
1     MPI_Request recvreq;
2     MPI_Request sendreq;
3
4     MPI_Recv_init (buffer, N, MPI_FLOAT, rank-1,
          tag_check_infos, MPI_COMM_WORLD, &recvreq);
5     MPI_Send_init (buffer, N, MPI_FLOAT, rank+1,
          tag_check_infos, MPI_COMM_WORLD, &sendreq);
6
7  /* ... copy stuff into buffer ... */
8
9     MPI_Start(&recvreq);
10    MPI_Wait(&recvreq, &status);
11
12    MPI_Request_free( &recvreq );
13    MPI_Request_free( &sendreq );
```

One-sided communication

- A MPI process can access another MPI process's memory space directly (RMA)
- No explicit coordination between both processes
- explicit transfer, explicit synchronization
- Better performance

One-sided communication

Initialization/Free (of the *window*)

- `MPI_Alloc_Mem()`, `MPI_Free_Mem()`
- `MPI_Win_Create()`, `MPI_Win_Free()`

Remote memory access

- `MPI_Put()`
- `MPI_Get()`
- `MPI_Accumulate()`

Synchronization

- `MPI_Win_Fence()`
- `MPI_Win_Post()`, `MPI_Win_Start()`, `MPI_Win_Complete()`, `MPI_Win_Wait()`
- `MPI_Win_Lock()`, `MPI_Win_Unlock()`

Memory allocation

- allocate `size` of memory segments in bytes
- `info` can be used to provide directives that control the desired location of the allocated memory
- `*baseptr` is the pointer to the beginning of the memory segment

```
1  int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void
       *baseptr)
```

Memory `window` creation

- A `MPI_Win` is an opaque object which can be reused to perform one-sided communication
- A `window` is a specified region in memory that can be accessed by another process

```
1 int MPI_Win_create(void *base, MPI_Aint size, int
    disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win
    *win)
```

where `base` is the initial address of the region, of `size` length of size `disp_unit` in bytes.

Put/Get within the window

- close to an MPI_Send call with
  - *what to send* : origin_addr start of the buffer of size origin_count of type origin_datatype
  - *to which process* : target_rank at the place target_count of type target_datatype
  - *in which context* : within the window win

```
1 int MPI_Put(const void *origin_addr, int origin_count
    , MPI_Datatype origin_datatype, int target_rank,
    MPI_Aint target_disp, int target_count,
    MPI_Datatype target_datatype, MPI_Win win)
```

```
1 int MPI_Get(void *origin_addr, int origin_count,
    MPI_Datatype origin_datatype, int target_rank,
    MPI_Aint target_disp, int target_count,
    MPI_Datatype target_datatype, MPI_Win win)
```

One-sided communications example

```
1  MPI_Win win;
2  int *mem;
3  float x = 1.0;
4  MPI_Alloc_mem(size * sizeof(int), MPI_INFO_NULL, &mem
       );
5  MPI_Win_create(mem, size * sizeof(int), sizeof(int),
       MPI_INFO_NULL, MPI_COMM_WORLD, &win);
6
7  // Write x at position 1 within process 0 's memory
8  MPI_Put(&x, 1, MPI_FLOAT, 0, rank, 1, MPI_INT, win);
9
10 MPI_Win_free(win);
11 MPI_Free_mem(mem);
```

One-sided communications remarks

- Pay attention to the memory coherence
- Can be dangerous : how a process knows if its data are in use/modified ?
- MPI-3 provides new features :
  - cache-coherent windows,
  - new primitives `MPI_Get_accumulate()`, `MPI_Fetch_and_op()`, `MPI_Compare_and_swap`,
  - requested-based primitives like `MPI_R{put,get,accumulate,get_accumulate}`,
  - "all"-versions of the synchronization routines : `MPI_Win_{un}lock_all`, `MPI_Win_flush{_all}`, `MPI_Win_flush_local{_all}`

Static model vs. Dynamic model

MPI 1:

- Fixed number of MPI processes
- What happens if a process fail ? The entire job fails.
- impossible to "inter-connect" two independent MPI running jobs

MPI 2 / MPI 3:

- support the creation of processes **on the fly** that can connect to existing running processes
- Concept of **intercommunicator** to handle the communications between parents and children
- Tricky !!

Master-Slave example (www.mpi-forum.org) – Master

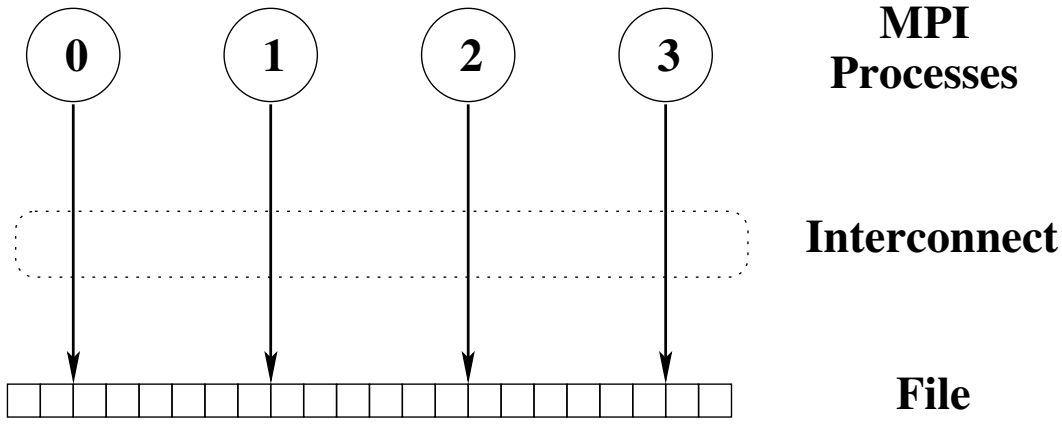
```
1  int main(int argc, char *argv[]) {
2      int world_size, universe_size, *universe_sizep,
           flag;
3      MPI_Comm everyone;           /* intercommunicator */
4      char worker_program[100];
5      MPI_Init(&argc, &argv);
6      MPI_Comm_size(MPI_COMM_WORLD, &world_size);
7      MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, &
           universe_sizep, &flag);
8      universe_size = *universe_sizep;
9      choose_worker_program(worker_program);
10     MPI_Comm_spawn(worker_program, MPI_ARGV_NULL,
           universe_size-1, MPI_INFO_NULL, 0,
           MPI_COMM_SELF, &everyone, MPI_ERRCODES_IGNORE);
11 / * Parallel code here. */
12     MPI_Finalize();
13     return 0; }
```

Master-Slave example (www.mpi-forum.org) – Slave

```
1  int main(int argc, char *argv[]) {
2      int size;
3      MPI_Comm parent;
4      MPI_Init(&argc, &argv);
5      MPI_Comm_get_parent(&parent);
6      if (parent == MPI_COMM_NULL) error("No parent!");
7      MPI_Comm_remote_size(parent, &size);
8      if (size != 1) error("Something's wrong with the
           parent");
9
10 / * Parallel code here. */
11
12     MPI_Finalize();
13     return 0;
14 }
```

Introducing remarks

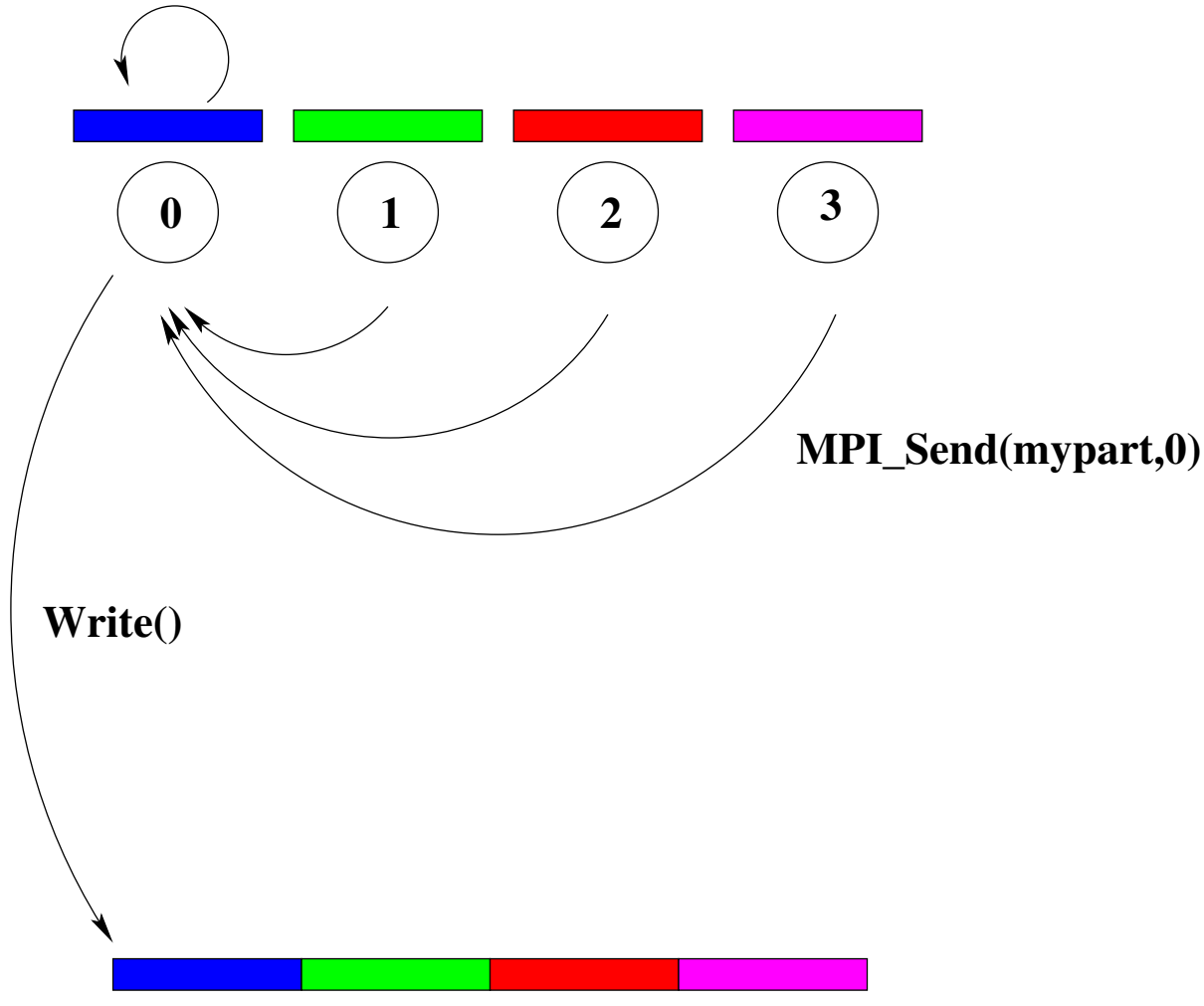
- I/O is often (if not always) the main bottleneck in a parallel application
- MPI provides a mechanism to read/write in parallel



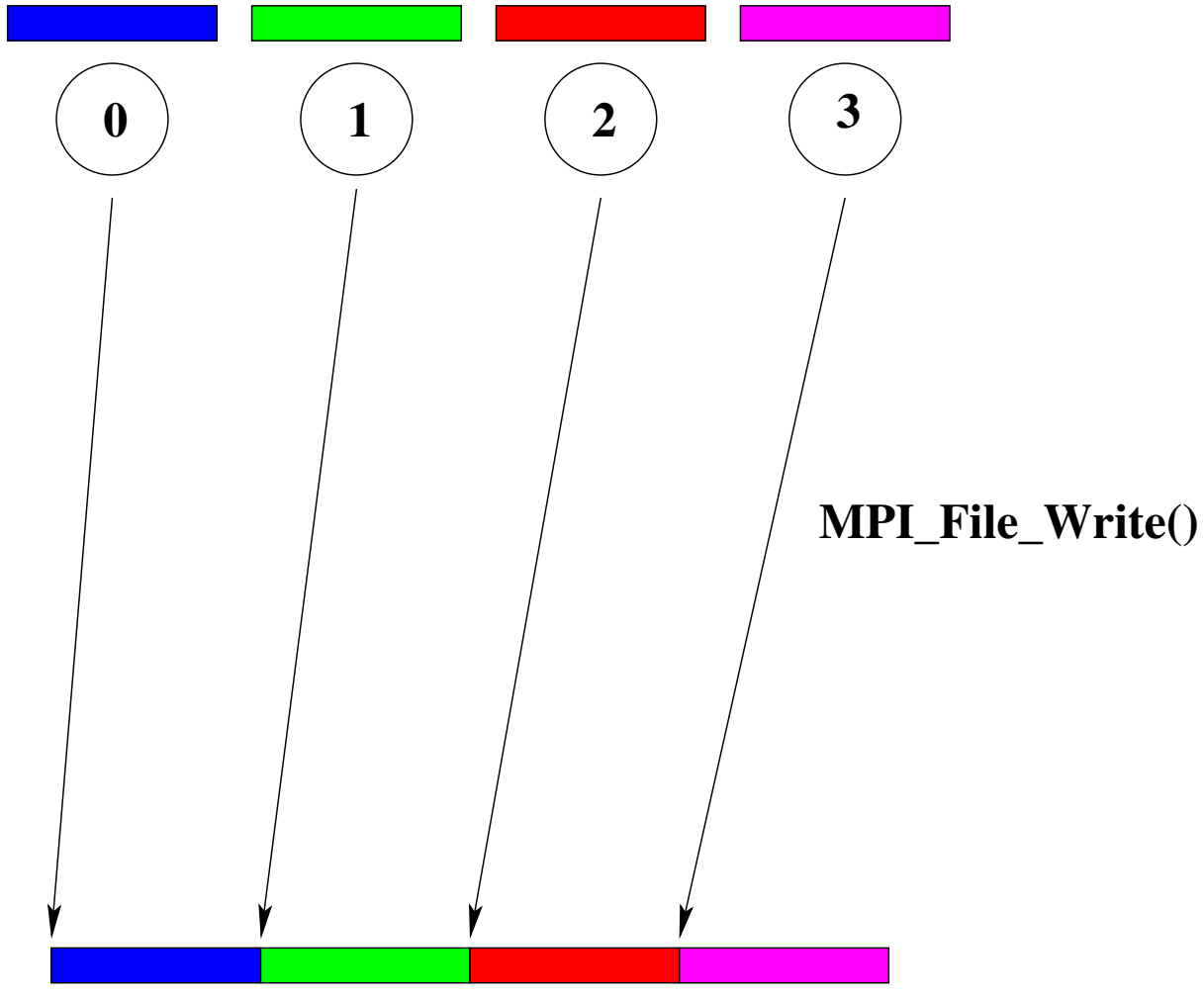
Introducing remarks

- MPI IO API works on your desktop/laptop
- Most of the large HPC systems have a **parallel file system** (like GPFS, Lustre, etc..)
- If the file is distributed smartly on a parallel file system : performance increases
- MPI IO offers a high-level API to access a distributed file (no needs to implement complexe POSIX calls)
- **does not work with ASCII files**
- Most of the standard file format support MPI IO (e.g. HDF5, NetCDF, etc..)

Poisson so far



Poisson ideal



Open/Close a file in parallel

- `comm` : the communicator that contains the writing/reading MPI processes
- `*filename` : a file name
- `amode` : file access mode (Read only `MPI_MODE_RDONLY`, read/write `MPI_MODE_RDWR`, create `MPI_MODE_CREATE`, etc..)
- `info` : file info object
- `*fh` : file handle

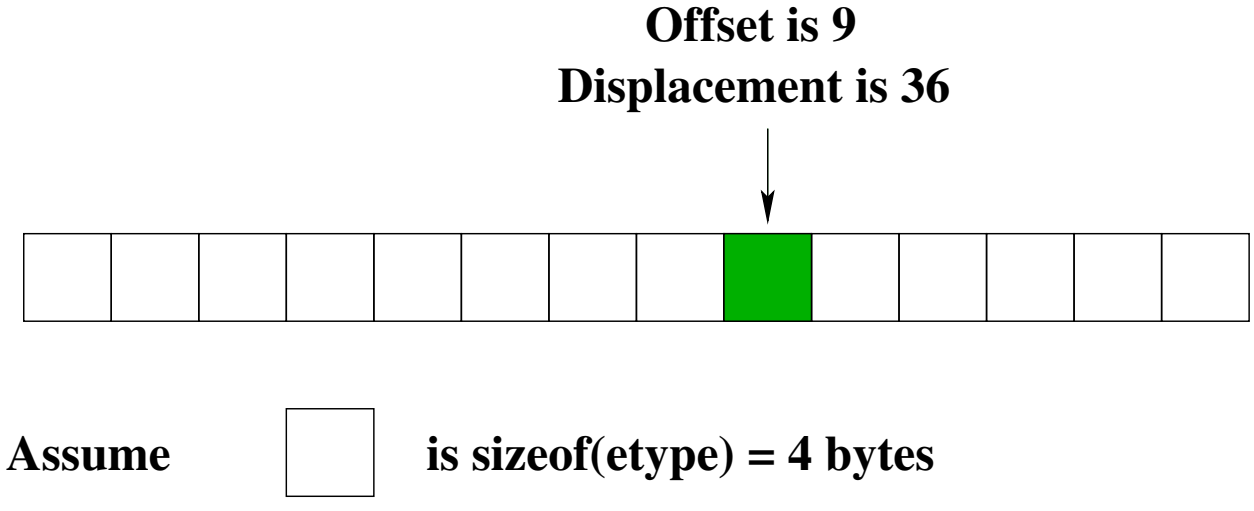
```
1 int MPI_File_open(MPI_Comm comm, const char *filename
      , int amode, MPI_Info info, MPI_File *fh)
```

```
1 int MPI_File_close(MPI_File *fh)
```

Collective calls !!

etype, offset and displacement

- **etype** is the elementary type of the data of the parallel accessed file
- **offset** is a position in the file in term of multiple of etypes
- **displacement** of a position within the file is the number of bytes from the beginning of the file



Simple independent read/write

- Can be used from a single (or group) of processes
- The `offset` must be specified in the `*buf` buffer
- `count` elements of type `datatype` are written

```
1  int MPI_File_write_at(MPI_File fh, MPI_Offset offset,
       ROMIO_CONST void *buf, int count, MPI_Datatype
     datatype, MPI_Status *status)
```

```
1  int MPI_File_read_at(MPI_File fh, MPI_Offset offset,
     void *buf,int count, MPI_Datatype datatype,
     MPI_Status *status)
```

`view` by each process

- Initialy, each process view the file as a linear byte stream and each process views data in its own native representation
- this is changed using `MPI_File_set_view`
- `disp` is the displacement (defines the beginning of the data of the file that belongs to the process) in bytes
- `etype` is the elementary type

```
1 int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
    MPI_Datatype etype, MPI_Datatype filetype,
    ROMIO_CONST char *datarep, MPI_Info info)
```

```
1 int MPI_File_get_view(MPI_File fh, MPI_Offset *disp,
    MPI_Datatype *etype, MPI_Datatype *filetype, char
     *datarep)
```

Setting up a `view`

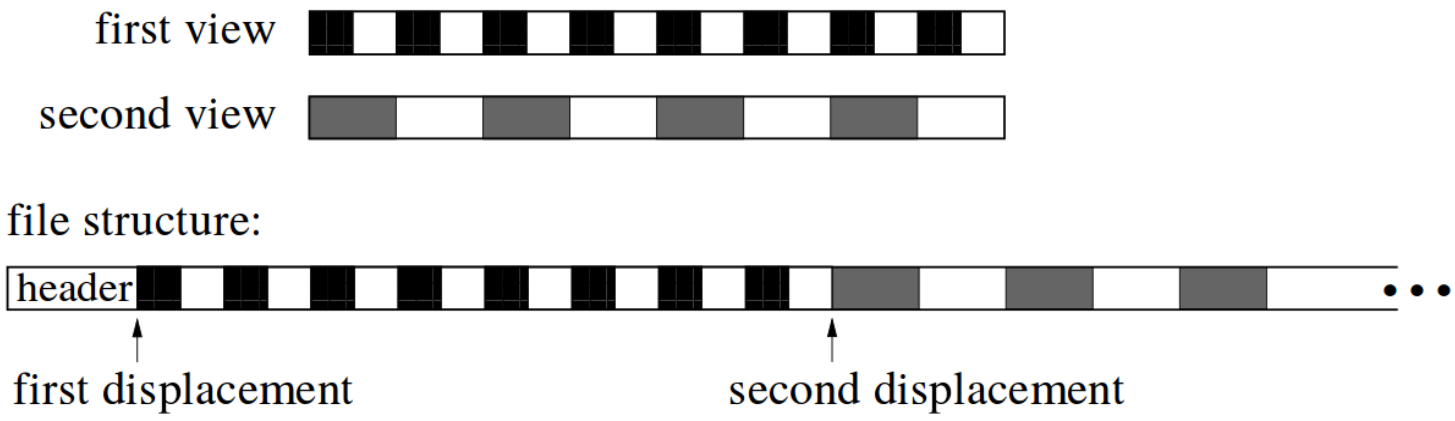


Figure 13.3: Displacements

(source : MPI 2.2 specifications)

Simple independent read/write without offset

- the `view` is specified prior to the call

```
1 int MPI_File_write(MPI_File fh, ROMIO_CONST void *buf
    , int count, MPI_Datatype datatype, MPI_Status *
    status)
```

```
1 int MPI_File_read(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)
```

Collective read/write with/without offset

- Same structure than Independent routines but with `_all` at the end
- for instance :

```
1  int MPI_File_write_all(MPI_File fh, ROMIO_CONST void
       *buf, int count, MPI_Datatype datatype,
       MPI_Status *status)
```

Using subarrays

- subarray from a structured data grid
- definition of subarrays leads to a collective call
- definition of a pattern
- hallos (or ghostcells) are allowed
- like defining a new datatype
- subarrays **must** have the same size on each process

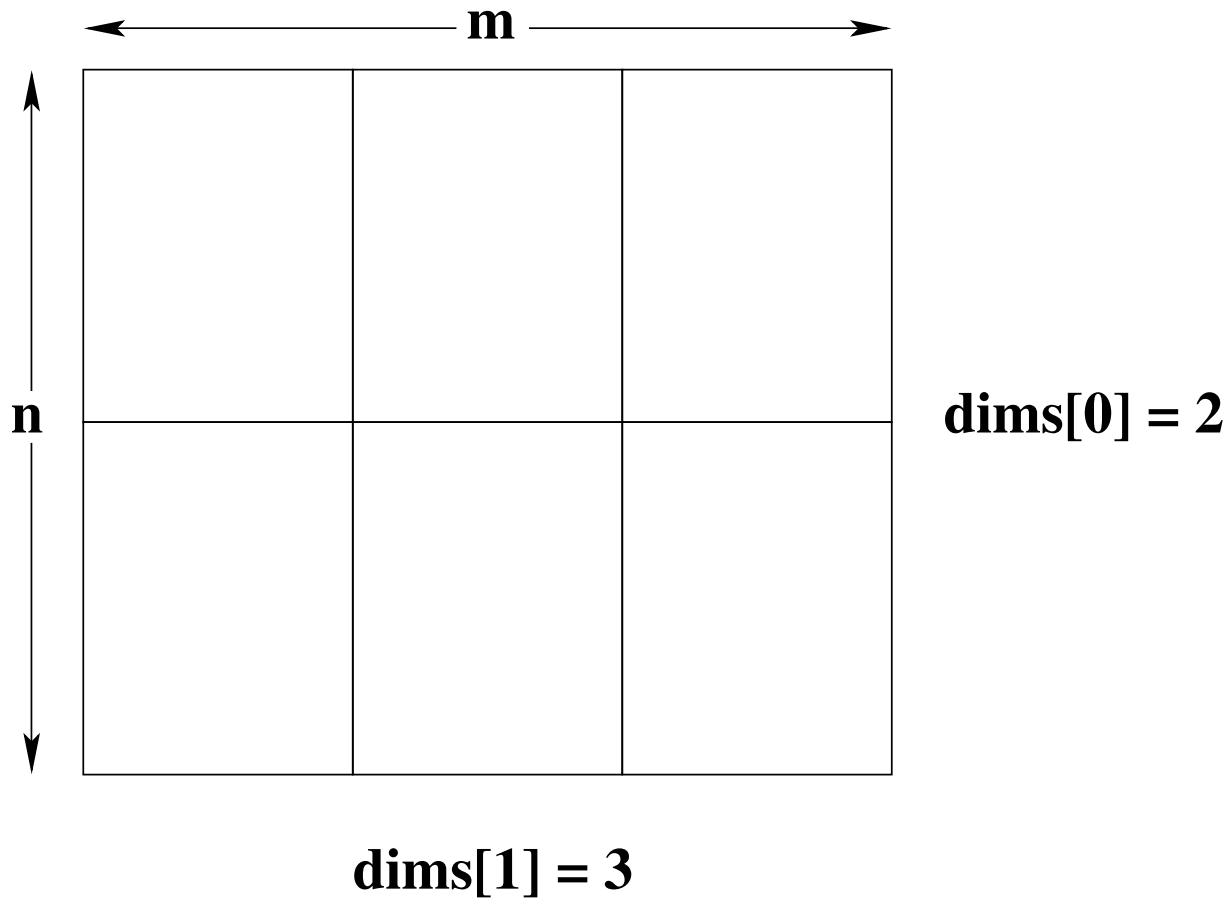
```
1  int MPI_Type_create_subarray(int ndims, const int
       array_of_sizes[], const int array_of_subsizes[],
       const int array_of_starts[], int order,
       MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Parenthesis : Virtual Topology

- A subarray must be mapped onto a topology
- It is done through the creation of a new communicator
- Cartesian topology fits with subarrays from a regular cartesian grid

```
1 int MPI_Cart_create(MPI_Comm comm_old, int ndims,
    const int dims[], const int periods[], int
    reorder, MPI_Comm *comm_cart)
```

Virtual topology example



(source : Andrew Siegel, University of Chicago)

Virtual topology example

```
1  gsizes[0] = m; // no. of rows in global array
2  gsizes[1] = n; // no. of columns in global array
3  psizes[0] = 2; // no. of procs. in vert. dimension
4  psizes[1] = 3; // no. of procs. in hori. dimension
5  lsizes[0] = m/psizes[0]; // no. of rows in local
       array
6  lsizes[1] = n/psizes[1]; // no. of columns in local
       array
7  dims[0] = 2; dims[1] = 3;
8  periods[0] = periods[1] = 1;
9  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0,
       &comm);
10 MPI_Comm_rank(comm, &rank);
11 MPI_Cart_coords(comm, rank, 2, coords);
```

(source : Andrew Siegel, University of Chicago)

Back to the subarrays

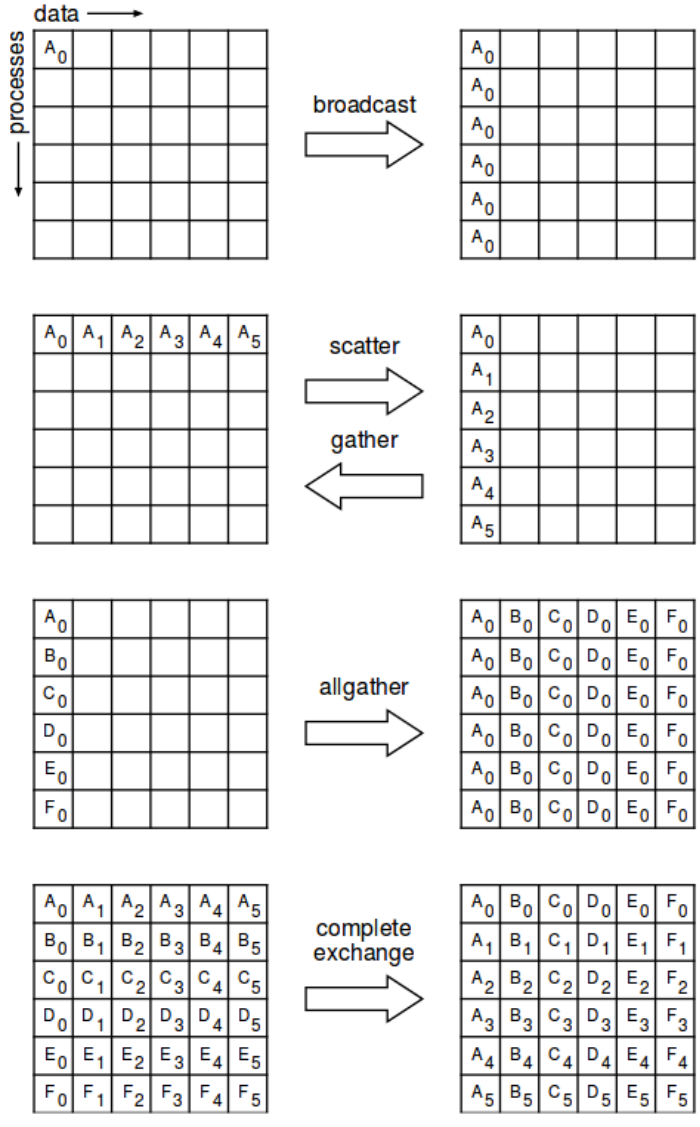
```
1 start_indices[0] = coords[0] * lsizes[0];
2 start_indices[1] = coords[1] * lsizes[1];
3 MPI_Type_create_subarray(2, gsizes, lsizes,
      start_indices, MPI_ORDER_C, MPI_FLOAT, &filetype)
      ;
4 MPI_Type_commit(&filetype);
5 MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
      MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL,
       &fh);
6 MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native
      ", MPI_INFO_NULL);
7 local_array_size = lsizes[0] * lsizes[1];
8 MPI_File_write_all(fh, local_array, local_array_size,
       MPI_FLOAT, &status);
```

(source : Andrew Siegel, University of Chicago)

MPI IO remarks

- The `I` versions (non-blocking) exist
- use collectives as often as possible
- contigous / non-contiguous data can lead to tricky implementations
- MPI IO is efficient on parallel file systems (such as all the clusters at EPFL). It can slow down a code on a simple Desktop/laptop

Reminder : Collective communications



"v-version" of `MPI_Gather` : `MPI_Gatherv`

- adds a stride at receiving ends
- (example does not work if `stride > 100`)

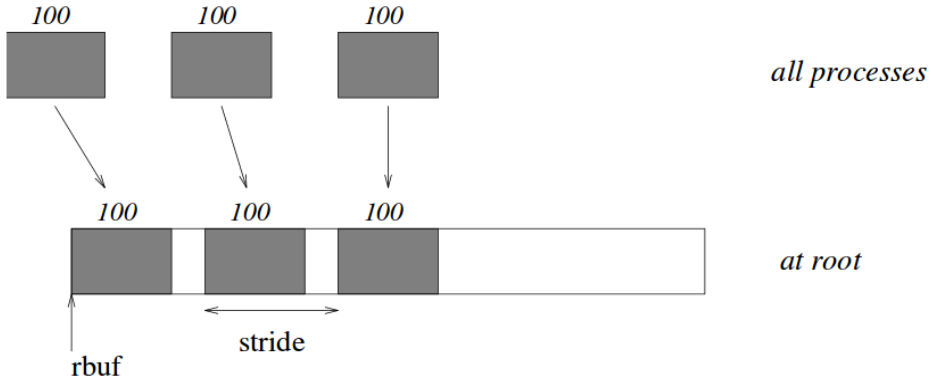


Figure 5.5: The root process gathers 100 `ints` from each process in the group, each set is placed `stride ints` apart.

“v-version” of MPI_Gather : MPI_Gatherv

```
1  MPI_Comm comm;
2  int gsize,sendarray[100];
3  int root, *rbuf, stride;
4  int *displs,i,*rcounts;
5  ...
6  MPI_Comm_size(comm, &gsize);
7  rbuf = (int *)malloc(gsize*stride*sizeof(int));
8  displs = (int *)malloc(gsize*sizeof(int));
9  rcounts = (int *)malloc(gsize*sizeof(int));
10 for (i=0; i<gsize; ++i) {
11    displs[i] = i*stride;
12    rcounts[i] = 100;
13 }
14 MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts,
       displs, MPI_INT,root, comm);
```

"v-version" of `MPI_Scatter` : `MPI_Scatterv`

- adds a stride at sending ends
- (example does not work if `stride > 100`)

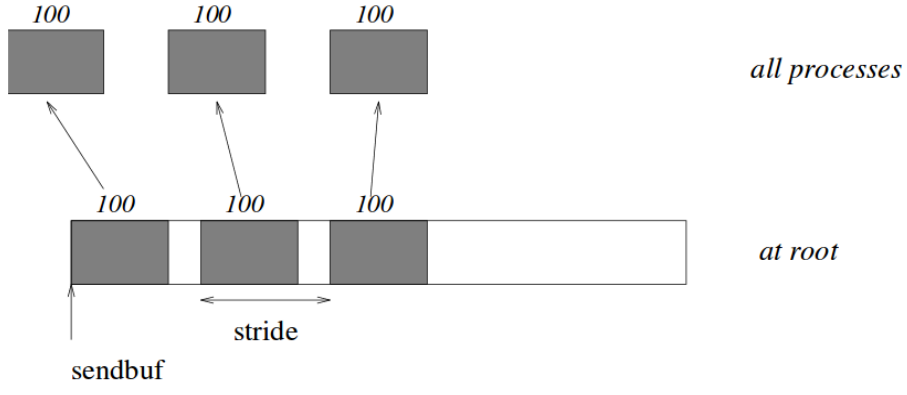


Figure 5.10: The root process scatters sets of 100 `ints`, moving by `stride ints` from send to send in the scatter.

"v-version" of MPI_Scatter : MPI_Scatterv

```
1  MPI_Comm comm;
2  int gsize,*sendbuf;
3  int root, rbuf[100], i, *displs, *scounts;
4  ...
5  MPI_Comm_size(comm, &gsize);
6  sendbuf = (int *)malloc(gsize*stride*sizeof(int));
7  ...
8  displs = (int *)malloc(gsize*sizeof(int));
9  scounts = (int *)malloc(gsize*sizeof(int));
10 for (i=0; i<gsize; ++i) {
11    displs[i] = i*stride;
12    scounts[i] = 100;
13 }
14 MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf,
       100, MPI_INT, root, comm);
```

Non-blocking collectives (NBC) for what ?

- Same situations as for non-blocking point-to-point communications :
  - small message sizes
  - enough computation to perform between start and end of the communication
  - algorithm that authorizes to compute with a partial knowledge of the data

Non-blocking collectives (NBC)

- `int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)` : NB version of `MPI_Barrier()`
- `int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root,MPI_Comm comm, MPI_Req` : NB version of `MPI_Bcast()`. Example :

```
1 MPI_Comm comm;
2 int array1[100], array2[100];
3 int root=0;
4 MPI_Request req;
5 ...
6 MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req
      );
7 compute(array2, 100);
8 MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Non-blocking collectives (NBC)

Scatter/Gather versions

- `int MPI_Igather()` : NB version of `int MPI_Gather()`
- `int MPI_Igatherv()` : NB version of `int MPI_Gatherv()`
- `int MPI_Iscatter()` : NB version of `int MPI_Scatter()`
- `int MPI_Iscatterv()` : NB version of `int MPI_Scatterv()`

All Scatter/Gather versions

- `int MPI_Iallgather()` : NB version of `int MPI_Allgather()`
- `int MPI_Iallgatherv()` : NB version of `int MPI_Allgatherv()`

Non-blocking collectives (NBC)

All to all

- `int MPI_Ialltoall()` : NB version of `int MPI_Alltoall()`
- `int MPI_Ialltoallv()` : NB version of `int MPI_Alltoallv()`
- `int MPI_Ialltoallw()` : NB version of `int MPI_Alltoallw()`

Non-blocking collectives (NBC)

Reduce

- `int MPI_Ireduce()` : NB version of `int MPI_Reduce()`
- `int MPI_Iallreduce()` : NB version of `int MPI_Allreduce()`
- `int MPI_Ireduce_scatter_block()` : NB version of `int MPI_Reduce_scatter_block()`
- `int MPI_Ireduce_scatter()` : NB version of `int MPI_Reduce_scatter()`

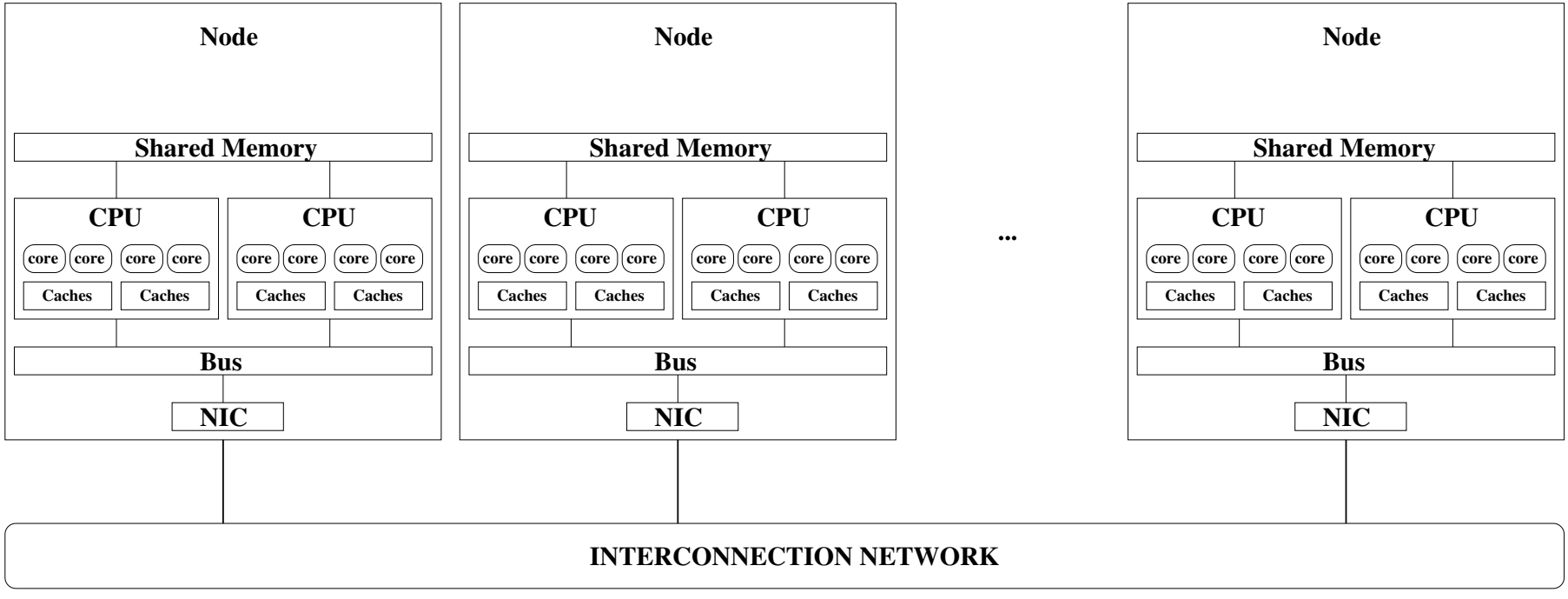
Scan

- `int MPI_Iscan()` : NB version of `int MPI_Scan()`
- `int MPI_Iexscan` : NB version of `int MPI_Exscan`

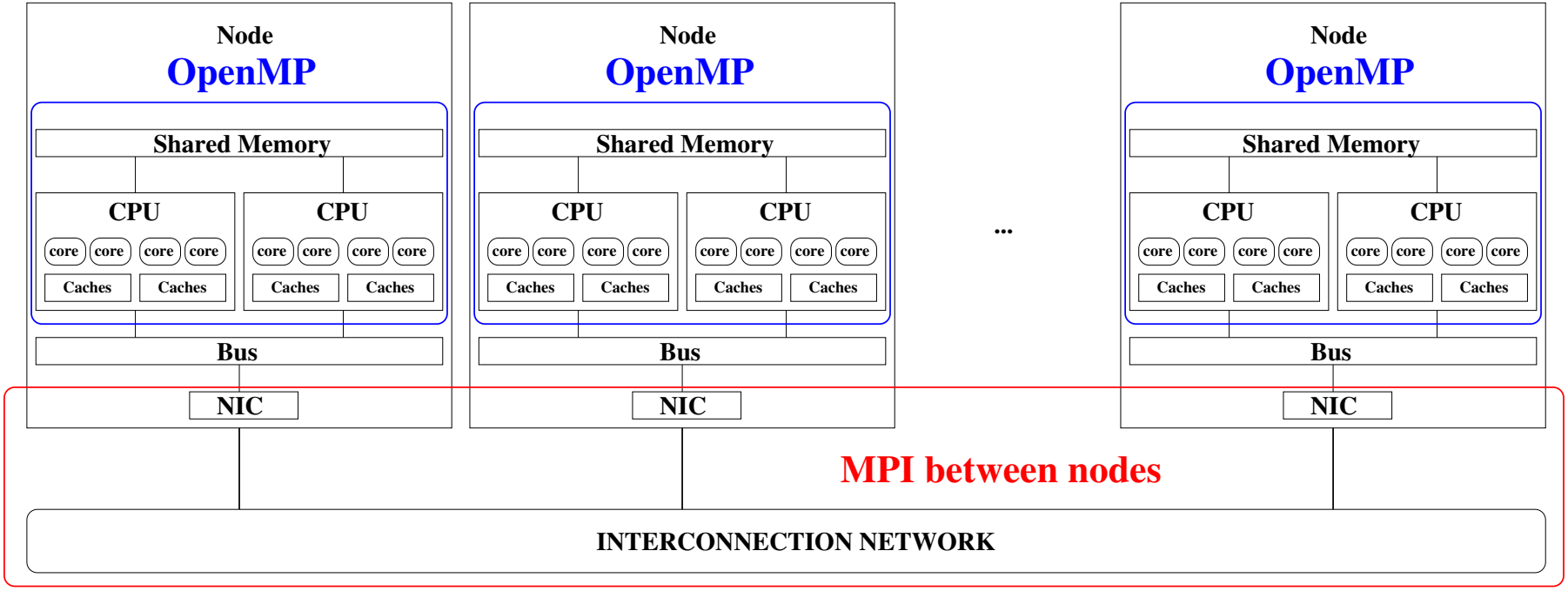
What will we learn today ?

- Hybrid programming models comparison
  - Pure MPI
  - MPI+OpenMP
  - (MPI + MPI one-sided (MPI-3))
- How to write a (production) project proposal

Situation


Node
Shared Memory
CPU
core
core
core
core
Caches
Caches
CPU
core
core
core
core
Caches
Caches
Bus
NIC
Node
Shared Memory
CPU
core
core
core
core
Caches
Caches
CPU
core
core
core
core
Caches
Caches
Bus
NIC
...
Node
Shared Memory
CPU
core
core
core
core
Caches
Caches
CPU
core
core
core
core
Caches
Caches
Bus
NIC
INTERCONNECTION NETWORK

Situation


Node
OpenMP
Shared Memory
CPU
CPU
core
Caches
Bus
NIC
...
MPI between nodes
INTERCONNECTION NETWORK

Situation : problems

- Thread safety ?
- Which thread/process can/will call the MPI library ?
- MPI process placement in the case of multi-CPU processors ?
- Data visibility ? OpenMP private ?
- Does my problem fits with the targeted machine ?
- Levels of parallelism within my problem ?

Hybrid vs. Pure MPI

Pure MPI

- + no code modification
- + most of the libraries support multi-thread
- - does application topology fits system topology ?
- - useless communications

Hybrid

- + no message within an SMP node
- + less (no) topology problems
- - all threads sleep when master communicates
- - MPI-libs must support (at least) thread safety

Hybrid MPI/OpenMP hello world

```
1  int main(int argc, char *argv[]) {
2    int numprocs, rank, namelen,provided;
3    char processor_name[MPI_MAX_PROCESSOR_NAME];
4    int iam = 0, np = 1;
5    MPI_Init_thread(&argc, &argv,MPI_THREAD_SINGLE,
         provided);
6    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
7    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8    #pragma omp parallel default(shared) private(iam,
         np)
9    {
10     np = omp_get_num_threads();
11     iam = omp_get_thread_num();
12     printf("Hello from thread %d out of %d from
           process %d out of %d on\n",iam, np, rank,
           numprocs);
13   }
14   MPI_Finalize();
```

Hybrid MPI/OpenMP hello world

Compilation using the GNU gcc compiler:

```
mpicc -fopenmp hello.c -o hello
```

Compilation using the Intel C compiler:

```
mpiicc -openmp hello.c -o hello
```

Warning : when using Intel MPI : it is mandatory to link against the thread-safe library (-mt_mpi) or at least to check if the executable has been linked against this lib (`ldd hello` should print `libmpi_mt.so.12`) if the `mpiicc` or `mpiifort` has been used.

Submission script on Bellatrix

```
#!/bin/bash
#SBATCH --nodes 2
#SBATCH --ntasks 2
#SBATCH --cpus-per-task 1
#SBATCH --ntasks-per-node=1
#SBATCH --time 00:00:30
#SBATCH --workdir /scratch/vkeller

module purge
module load intel/15.0.0 intelmpi/5.0.1

export OMP_NUM_THREADS=16
srun ./hello
```

It will start 2 MPI processes each will spawn 16 threads

Changes to your code

- change your MPI initialisation routine
  - `MPI_Init` is replaced by `MPI_Init_thread`
  - `MPI_Init_thread` has two additional parameters for the level of thread support required, and for the level of thread support provided by the library implementation

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

required specifies the requested level of thread support, and the actual level of support is then returned into provided

- add OpenMP directives or Pthread calls as long as you stick to the level of thread safety you specified in the call to `MPI_Init_thread`

The 4 Options for Thread Support

- `MPI_THREAD_SINGLE`
  - Only one thread will execute
  - Standard MPI-only application
- `MPI_THREAD_FUNNELED`
  - Only the Master Thread will make calls to the MPI library
  - A thread can determine whether it is the master thread by a call to `MPI_Is_thread_main`
- `MPI_THREAD_SERIALIZED`
  - Only one thread at a time will make calls to the MPI library, but all threads are eligible to make such calls

The 4 Options for Thread Support

- `MPI_THREAD_MULTIPLE`
  - Any thread may call the MPI library at any time
  - The MPI library is responsible for thread safety within that library, and for any libraries that it in turn uses
  - Codes that rely on the level of `MPI_THREAD_MULTIPLE` may run significantly slower than the case where one of the other options has been chosen
  - You might need to link in a separate library in order to get this level of support

In most cases `MPI_THREAD_FUNNELED` provides the best choice for hybrid programs

```
int MPI_Query_thread( int * thread_level_provided);
```

Returns the level of thread support provided by the MPI library

Topology problems

How to deal with :

- topology / mapping ? (Which physical core is assigned to which process/thread)
- sub-domain decomposition ?
- halos size ? halos shapes ?
- unnecessary communications ?
- **computation to communication ratio** ?

Pure MPI ? Hybrid ?

A good solution is : one MPI process per "SMP" node

Halo regions

- Halo regions are local copies of remote data that are needed for computations
- Halo regions need to be copied fequently
- Using threads reduces the size of halo region copies that need to be stored
- Reducing halo region sizes also reduces communication requirements

Take-home messages

- Always take into account the problems related to the physical topology
- A real application is not as easy as a hello world.
- Some clusters have different connectivity topologies: match them to your problem. Examples of hardware topologies :
  - all-to-all
  - 2D/3D torus
  - tree
  - ...
- One MPI process per physical node

Main messages

- Do not use hybrid if the pure MPI code scales ok
- Be aware of intranode MPI behavior
- Always observe the topology dependence of
  - Intranode MPI
  - Threads overheads
- Finally: Always compare the best pure MPI code with the best hybrid code!

Examples that *can* benefit of an hybrid approach

- MPI codes with a lot of all-to-all communications
- MPI codes with a very poor load balancing at the algorithmic level (less communications)
- MPI codes with memory limitations
- MPI codes that can be easily *fine-grained parallelized* (at loop level)

How to write a project proposal ?

Structure of a (good) project proposal

- (Administration) Who submits it ?
- (Scientific : Background) From where do we come ?
- (Scientific : Outcome) Where do we go ?
- (Technical : Application) What do we have, how it is implemented ?
- (Technical : Performance) Why do we need a high-end machine ?
- (Technical : Resource budget) How much do we need ?

Project title

- Keep it as explicit as possible
- Propose an accronym (8 letters) : often used as group name
- Follow the Computing Center's rules
- *A High Performance Implementation of a 2D Elliptic Equation Solver: the Poisson's Equation Case at Scale*. Acronym : POISSON
- *Implementation of a Generic Poisson Solver*. Acronym : POISSON

Intro cartouche

The goal is to have the basic administrative stuff and the main requests at a first glance

Principal investigator	Vincent Keller, PhD
Institution	École Polytechnique Fédérale de Lausanne
Laboratory	Scientific IT and Application Support
Adress	Station 1, CH-1015 LAUSANNE
Involved researchers	Nicolas Richart, PhD ; Christian Clémençon
Date of submission	April 13, 2015
Expected end of project	October 13, 2015
Target machine	CADMOS Blue Gene Q
Proposed acronym	ACRO

Intro cartouche : Good / Bad practices

- Keep it as short as possible
- PI + Key personnel, institution(s), start/end of project, summary of the technical request
- Not too many details
- CVs of the project members (put them in an approriate appendix)
- Potential Collaborations

Scientific background

- **Most important section of the proposal**
- You and your fellow research scientists or engineers are aware if your research deserves to get computing power
- To help you :
  - Good abstract
  - Description of the project
  - References

Abstract

- Explicitly express the link between the scientific background and the technical request
- 500 words max.
- Assume the reader has little/no knowledge about the technical part and/or the scientific matter
- No copy/paste from previous publications

Description of the project

- Description of the research/production project
- Motivation
  - My project uses an existing, known and well-tested code (Production project)
  - My project will explore a new paradigm at scale (exploratory project)
  - My project has never been tested at scale, needs improvements from an existing code (research project)
- Computational objectives
- Expected innovations both at scientific and computational levels

Scientific background and outcome

Most important point : this part is usually peer-reviewed by a pool of experts/reviewers from the domain.

- State-of-the-art review
- Extensive references
- Current status on the subject
- "We are the only ones in the world doing that and you are too stupid to understand it"

Technical part

Technical part of the project proposal. What is the **application** you are planning to run on the target architecture ?

- What code (application) are we planning to use
- Numerical methods
  - FEM, SEM, FDM, LBM, SPH, MD, etc..
- Algorithms
  - Describe the main algorithms
  - Main bottlenecks and how they have been tackled
- Implementation
  - Pure MPI, GPU, OpenMP, hybrid ? ...
  - Does it use special libraries ?
- If third-party code : name, version and license. Your contact within the third-party company/institution

Performance expectations

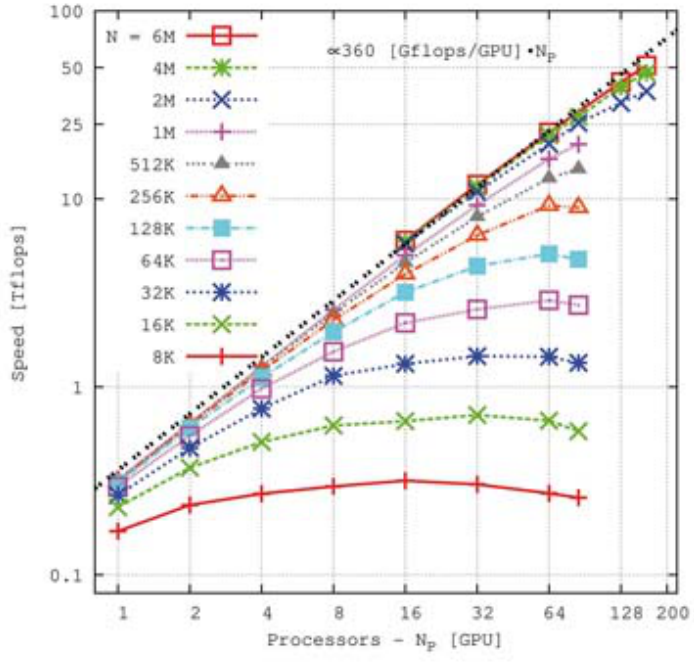
This section should explain how your code behaves and the accuracy with what you request.

- theoretical complexity: communication and computation complexities
- weak scaling : behavior of the application by fixing the size of the problem per processor and increasing the number of processors: **parallel efficiency** ($E_p = S_p/p$)
- strong scaling : behavior of the application by fixing the total size of the problem and increasing the number of processors: **speedup** ($S_p = t_1/t_p$)

Benchmarking

- You get access to a **test partition** which allows you to run test cases and measure the performance
- You also have previous performance measurements from the development
- Benchmarking is done on larger enough problems, closer to what will be done in production
- Use the theoretical complexities (computation and communication) to predict how your application will behave at production scale

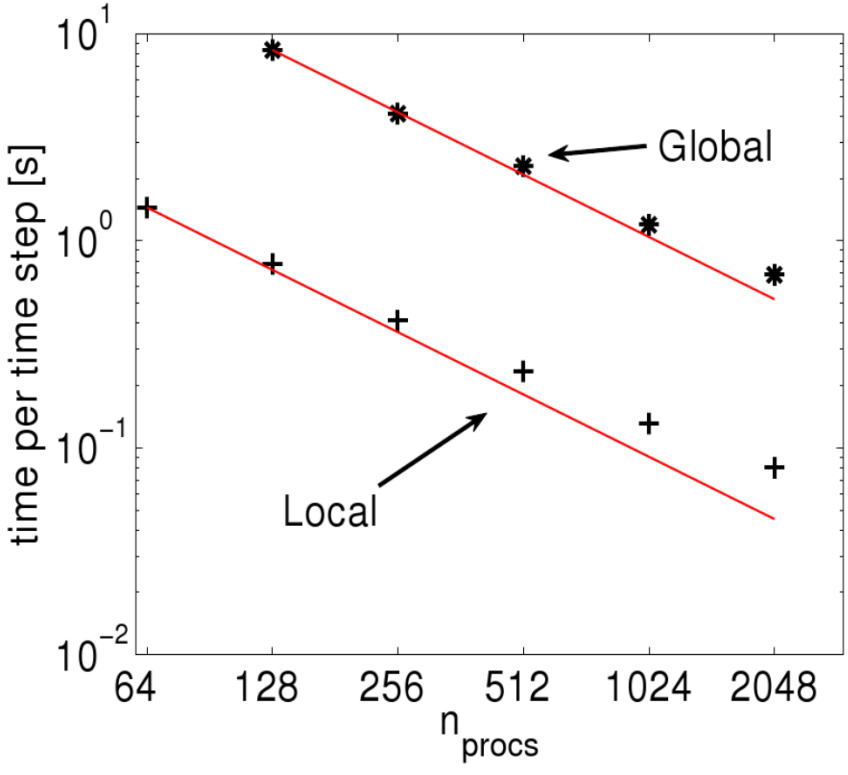
Strong scaling : Good example (with speedup)



- log-log graph
- Different problem sizes
- linear regression curves

Source: HLRS `http://inside.hlrs.de/_old/htm/Edition_01_12/article_20.html`

Strong scaling : Good example (with timing)

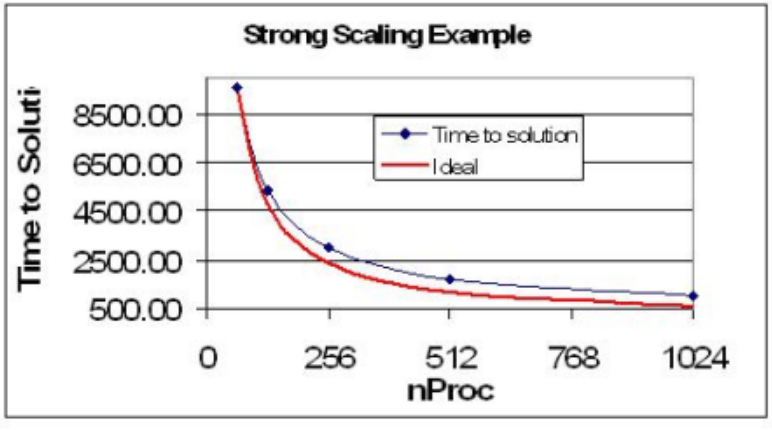


- log-log graph
- time **per step**
- explicit caption
- linear regression curves

Fig. 2. Strong scaling on HPC-FF for both the local (flux-tube) and global versions of the GENE code.

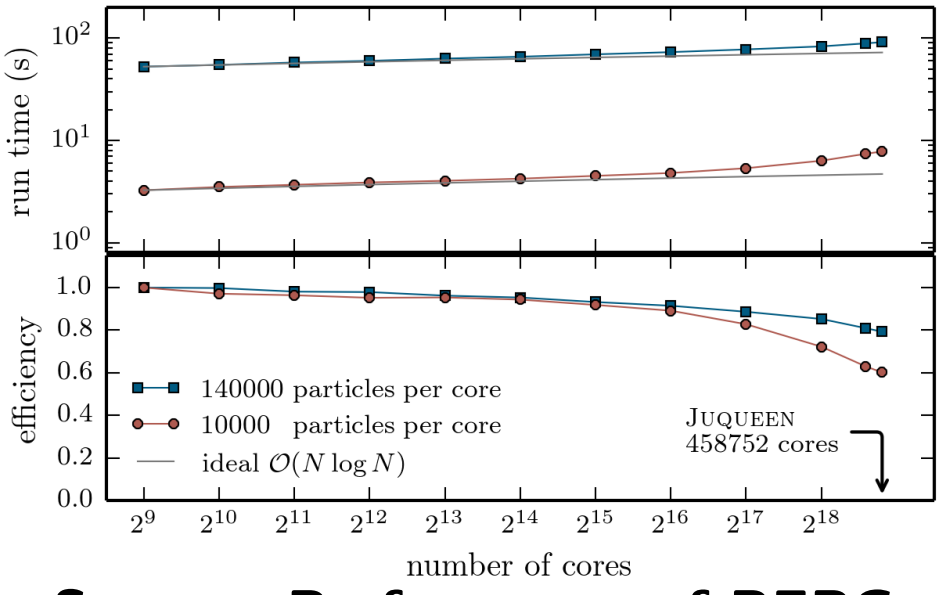
Source: CRPP, EPFL, Project proposal for BG/P, 2011

Strong scaling : bad example



- linear scale
- no caption
- "Excel" look
- useless for any conclusion

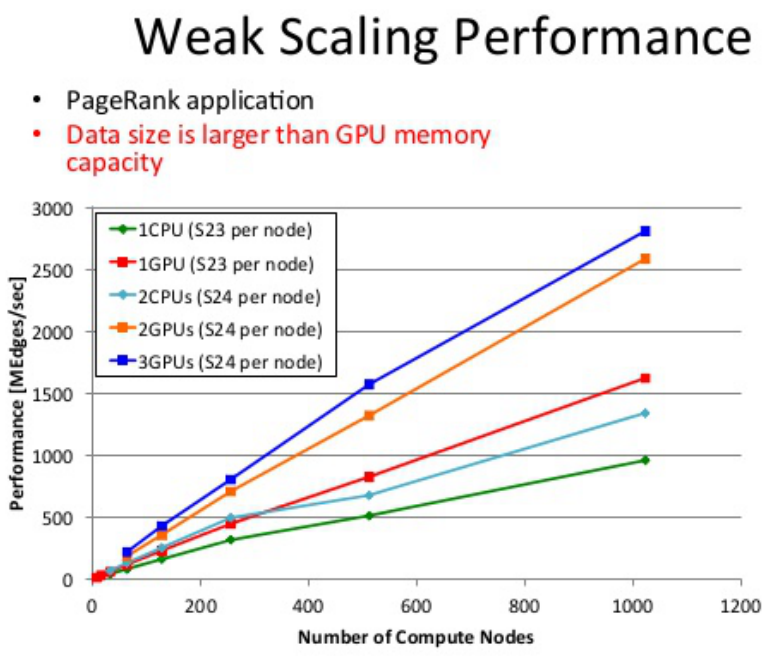
Weak scaling : Good example (FZJ Jülich, Germany)



- log-log graph
- runtimes and efficiency on the same graph
- two different problem sizes
- complexity order clearly stated
- Prediction on the largest partition of JUQUEEN

Source: Performance of PEPC `http://www.fz-juelich.de/ias/jsc/EN/AboutUs/Organisation/ComputationalScience/Simlabs/slpp/SoftwarePEPC/_node.html`

Weak scaling : bad example



- linear scales
- no caption
- "Excel" look
- comparison between CPUs and GPUs on the same graph? What is the "best"? Data does not fit in GPU memory, then?

Resource budget

Based on the performance expectations and benchmarks already done, you are able to provide a resource budget for your proposal

- Typical size of problem to solve. Total number of problems to solve
- Minimum and maximum memory size for the problems
- Disk space
- Communications needs (paradigm choice)
- Architecture

Summary of the Resource Budget

For the targeted problems

Total number of requested cores	128 - 512 [cores]
Minimum total memory	10 [GB]
Maximum total memory	256 [GB]
Temporary disk space for a single run	100 [GB]
Permanent disk space for the entire project	1 [TB]
Communications	Pure MPI
License	own code (BSD)
Code publicly available ?	Yes
Library requirements	LAPACK
Architectures where code ran	Intel 64, BG/Q

Facultative add-ons

- Technical support from the Computing Center (CC)
- Application-level support from the CC or other institution/groups