VRE for regional Interdisciplinary communities in Southeast Europe and the Eastern Mediterranean

Parallel programming with OpenMP and MPI

Vladimir Lončar
Institute of Physics Belgrade

The VI-SEEM project initiative is co-funded by the European Commission under the H2020 Research Infrastructures contract no. **675121**

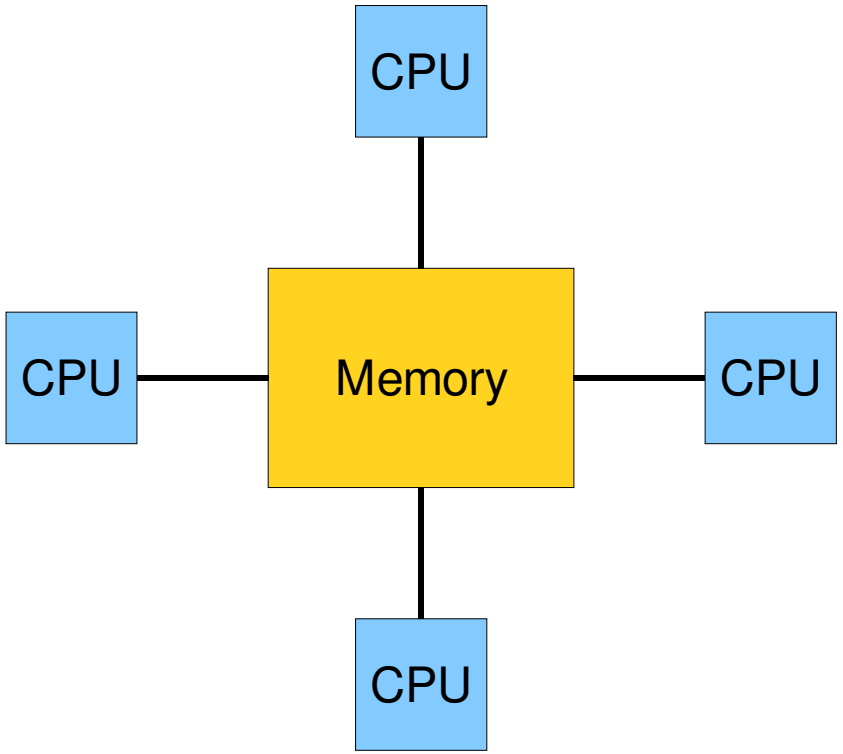
Agenda

- OpenMP
  - Introduction & “Hello World”
  - Parallel regions, loop parallelization
  - Work sharing constructs
  - Data scopes
  - Synchronization
- MPI
  - Introduction & “Hello World”
  - Point-to-point & collective communication
  - One-sided communication
  - I/O
- Hybrid OpenMP/MPI

Memory models

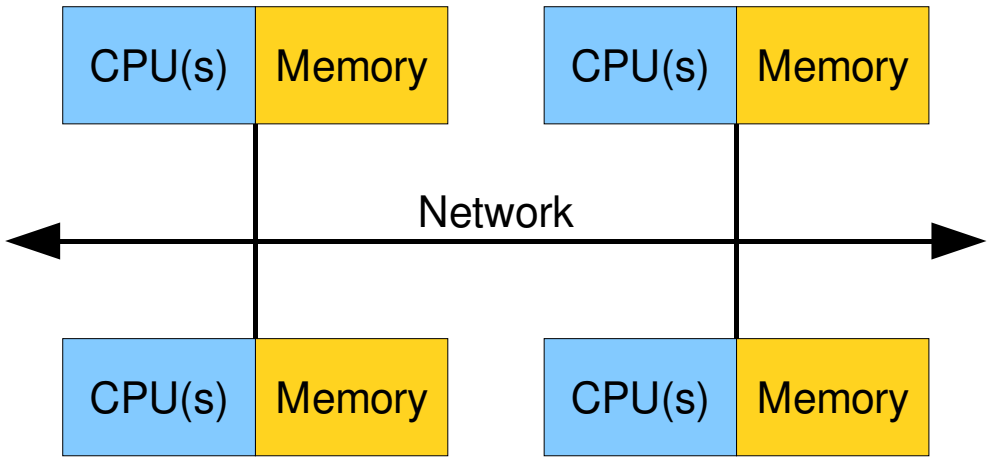
Shared memory

- Programmed with OpenMP (or MPI)



Distributed memory

- Programmed with MPI

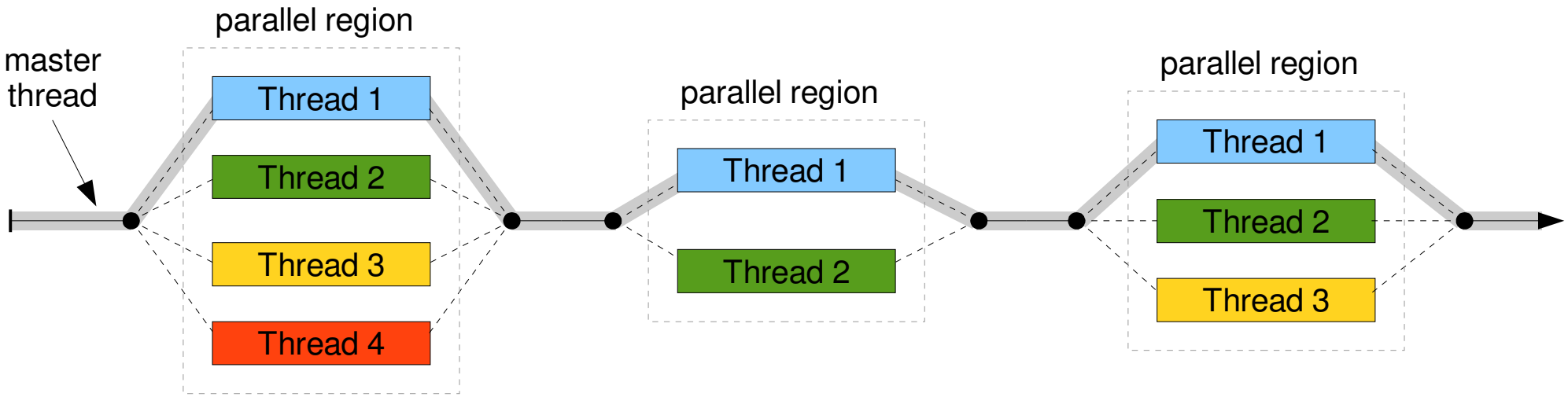


Introduction to OpenMP

- Application programming interface (API) for parallel programming on shared memory multiprocessors
  - Usually cores of a multicore CPU(s)
- Components of OpenMP
  - Compiler directives (pragmas)
  - Library functions
  - Environmental variables
- Supports multiple programming languages
  - Fortran, C, and C++
    - We will use C in examples
- Provides portable programming model
  - Significantly simplifies programming with threads

Fork–Join model

- OpenMP uses **fork–join** model
  - *Master* thread executes sequential code
  - Fork – Master thread creates/awakens additional threads to execute parallel code
  - Join – At end of parallel code created threads die or are suspended



Directives

- A way for the programmer to communicate with the compiler
  - Compiler free to ignore directives (they are hints)
- OpenMP directives
  - Case sensitive
  - End with newline
  - Applied to one succeeding statement (structured block)

```
#pragma omp directive-name [clause, ...]
{
    // code
}
```

Parallel regions

- Constructed using `parallel` pragma
- Block with `parallel` pragma is called *parallel region*
- All threads execute the same segment of code, in parallel
- Example:

```
#pragma omp parallel
{
  // this is executed by a team of threads in parallel
}
```

- How to identify each individual thread inside a parallel block?

```
#pragma omp parallel
{
    int t = omp_get_thread_num();
    printf("Hello world from %d!\n", t);
}
```

Hello OpenMP World!

```
#include <omp.h>
#include <stdio.h>

int main (int argc, char **argv) {
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int nth = omp_get_num_threads();
        printf("Hello World from thread = %d of %d\n",tid,nth);
    }

    return 0;
}
```

Compiling OpenMP programs

- Requires compiler support
  - Most modern compilers support OpenMP
    - GNU (`gcc`), LLVM (`clang`), Intel (`icc`), Portland (`pgcc`), IBM (`xlc`), Oracle (`suncc`), Microsoft (`cl.exe`) and many more
- GCC:

```
gcc -fopenmp hello_omp.c -o hello_omp
```

- Intel:

```
icc -qopenmp hello_omp.c -o hello_omp
```

- Intel (older versions):

```
icc -openmp hello_omp.c -o hello_omp
```

Loop parallelism

- Use `parallel for` pragma:

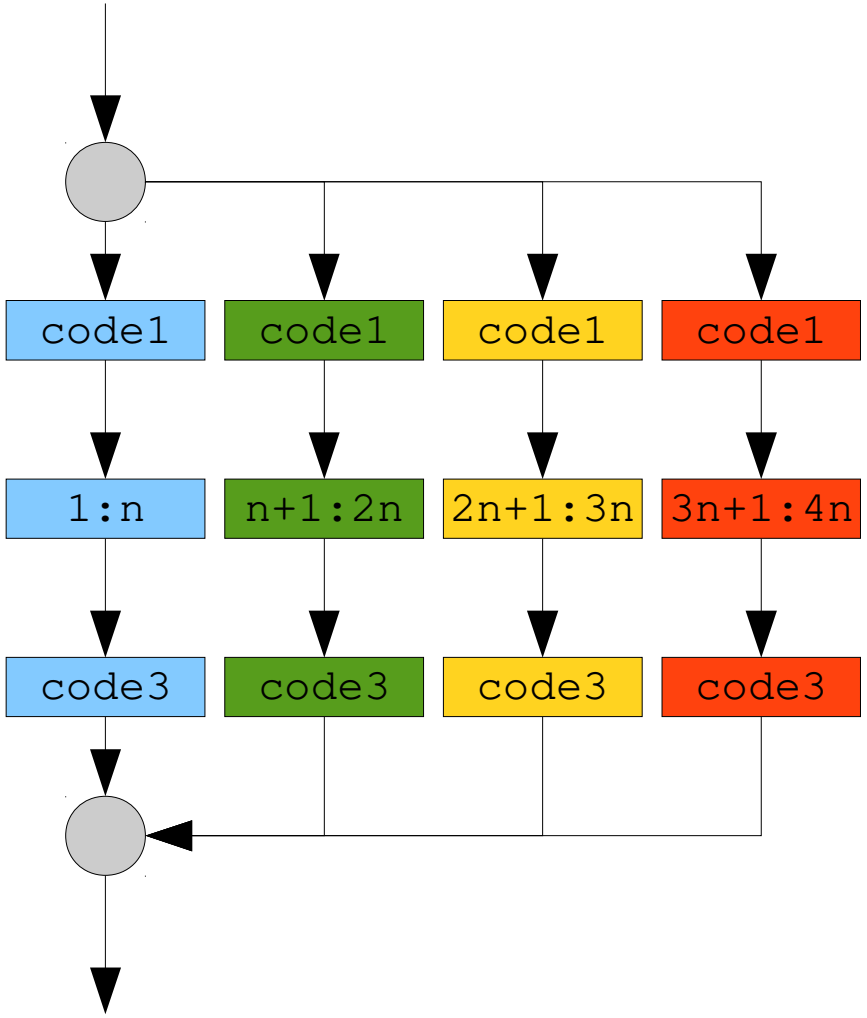
```
#pragma omp parallel
{
    #pragma omp for
    for (i=1; i<=4*N; i++) {
        // …
    }
}
```

```
#pragma omp parallel for
for (i=1; i<=4*N; i++) {
    // …
}
```

- OpenMP can only handle `for` loops, `while` loops can't be parallelized

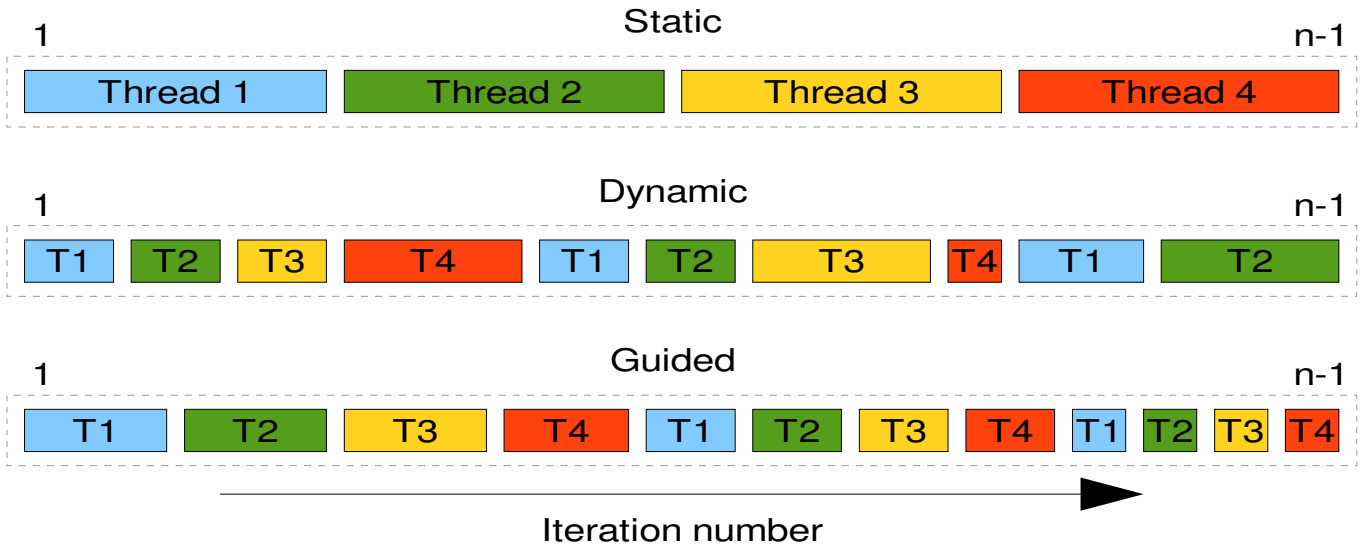
Execution of parallel region

```
#pragma omp parallel
{
    code1();
    #pragma omp for
    for (i=1; i<=4*N; i++) {
        code2();
    }
    code3();
}
```



Loop scheduling

- Specified with the `schedule(kind[,chunk])` clause, where `kind` is
  - `static` – Divide the loop into equal-sized chunks or as equal as possible
    - Good if all iterations take the same amount of time
  - `dynamic` – Use work queue to assign iterations to unoccupied threads
    - Better than `static` if iterations do not take the same amount of time
  - `guided` – Uses decreasing chunk size



More loop scheduling

- Other scheduling options:
  - `auto` – The schedule choice is left up to the compiler
  - `runtime` - Use the value of the `OMP_SCHEDULE` environment variable
- Optional chunk parameter controls the size of blocks
  - Increasing the chunk size makes the scheduling more static, and decreasing it makes it more dynamic

Sections

- Useful for independent, separate calculations
- Specified using `sections` and `section` directives
  - `section` directives are nested within a `sections` directive
  - Each `section` is executed once by a thread in the team

```
#pragma omp sections
{
    #pragma omp section
    // one calculation

    #pragma omp section
    // another calculation
}
```

Single/Master

- Only one thread executes code enclosed with the `single` directive
  - Implicit barrier at the end
- `master` directive is similar
  - Does not have a barrier at the end

```
#pragma omp parallel
{
    code1(); // Executed by every thread
    #pragma omp single
    {
        x = code2(); // A single thread executes this code
    }
    code3(x); // x has correct value here
}
```

Data scope

- By default, data declared outside a parallel region is **shared**, while data declared in the parallel region is **private**
- Scope can be explicitly defined using attribute clauses:
  - `private` – declares variables in its list to be private to each thread
  - `shared` – declares variables in its list to be shared among all threads in the team
  - `default` – allows the user to specify a default scope for all variables
  - `firstprivate` – initializes the variable to the value of their original objects
  - `lastprivate` – copies the value obtained from the sequentially last iteration (or section) back into the original variable object
  - `reduction` – performs a reduction operation on the variables in its list (+, *, min, max, bitwise, user-defined)
  - `threadprivate` – used for making thread data persistent
  - …

Data scope example

```
int i, n;
float a[100], b[100], result;
n = 100; result = 0.0;
for (i = 0; i < n; i++) {
    a[i] = i * 1.0; b[i] = i * 2.0;
}

#pragma omp parallel for default(none)  \
shared(n,a,b) private(i) reduction(+:result)
for (i = 0; i < n; i++) {
    result = result + (a[i] * b[i]);
}
printf("Final result = %f\n", result);
```

Synchronization

- OpenMP provides a variety of synchronization constructs that control the execution of each thread relative to other threads in the team:
  - Barriers
  - Locks
  - Critical sections
  - Atomic operations
  - Ordered execution
  - `flush` and `nowait` directives

Barriers and nowait

- Every work share construct has an implicit barrier
- Explicit barrier is defined with `barrier` construct

```
#pragma omp parallel
{
    x = code();
    #pragma omp barrier
    // Can safely use x after barrier
}
```

- Implicit barrier can be removed with `nowait` clause

```
#pragma omp for nowait
for (i = 0; i < 100; i++) {
    ...
}
```

Critical sections and atomic operations

- The `critical` directive specifies a region of code that must be executed by only one thread at a time

```
#pragma omp parallel
{
    x = code();
    #pragma omp critical
    do_something(x);
}
```

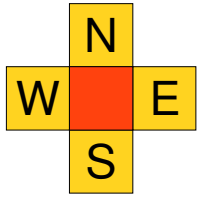
- Atomic operations are limited to single memory locations, but are possibly faster due to hardware support

```
a[i] += x; // Can be interrupted half-complete
#pragma omp atomic
b[i] += x; // Never interrupted because defined as atomic
```

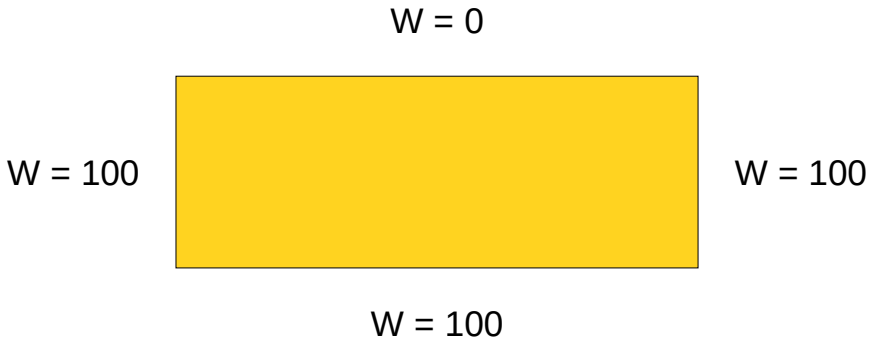
Example

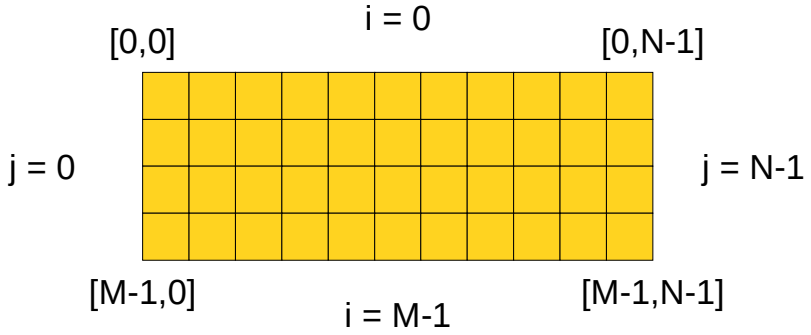
- Steady state heat equation (`heat_omp.c`)
  - Given boundary conditions
  - Interior point formula

$$w_C = \frac{w_N + w_E + w_S + w_W}{4}$$



  - Repeat until convergence of estimates
- Adapted from J. Burkardt's code
- OpenMP concepts used:
  - Parallel regions
  - Shared and private variables
  - Reduction
  - Single construct
- Compile: `make`





More OpenMP topics

- This was just a short introduction, OpenMP provides much more
  - More data scope attributes
  - More synchronization constructs
  - Nested parallelism & collapsing nested loops
  - Tasks
  - SIMD support
  - Offloading (since OpenMP 4.0)
  - Runtime tuning (affinity, binding…)

MPI

- Message Passing Model
  - Parallel programs consist of cooperating processes, each with its own memory
  - Processes send data to one another as messages
- Message Passing Interface (MPI)
  - Standardized message passing model
  - Just a standard, not an implementation
    - Multiple implementations exist, e.g., Open MPI, MPICH, vendor implementations
- Reasons for using MPI
  - Standardized & portable
  - Rich functionality
  - Many high-performance implementations

What MPI provides?

- A plethora of communications functions
  - Point-to-point communication routines
  - Collective operations
  - Remote-memory access
  - Blocking & non-blocking communication
- Process groups and hierarchies
- Datatypes
  - Basic & derived (user-defined) datatypes
- I/O operations
- 300+ functions in total

MPI basics

- MPI processes are collected into groups (communicators)
  - The group of all processes is initially given a predefined name called **MPI_COMM_WORLD**
- A process is identified by a unique number within each communicator, called **rank**
  - `MPI_Comm_rank(), MPI_Comm_size()`
- MPI environment has to be initialized at program start, and finalized before program ends
  - `MPI_Init(), MPI_Finalize()`
- MPI functions are defined in `mpi.h` header file

Hello MPI World!

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

int main(int argc, char **argv) {
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello MPI World from process %d of %d\n",rank,size);
    MPI_Finalize();

    return 0;
}
```

Compile & run

- Use `mpicc` compiler
  - Wrapper around host C/C++/Fortran compiler

```
mpicc hello_mpi.c -o hello_mpi
```

- Run with `mpiexec`
  - Specify number of processes and their placement
  - Pass additional arguments to MPI runtime

```
mpiexec -np 4 ./hello_mpi
```

- Output:

```
Hello MPI World from process 0 of 4
Hello MPI World from process 1 of 4
Hello MPI World from process 2 of 4
Hello MPI World from process 3 of 4
```

- Note that the order of `printf` statements may vary if processes share the output stream

Error handling

- MPI routines return an integer error code
  - In C, it is the function result
  - In Fortran, it is the parameter of the MPI function
- By default, an error causes all processes to abort
- User can associate an error handler with a communicator
  - Useful for libraries, not so much in scientific computation
  - Hard to recover from errors in parallel programs

Basic communication operations

- No messages have been exchanged in previous example
- Data is explicitly sent by one process and received by another
- Sender calls `MPI_Send()` specifying:
  - Whom to send (the rank of receiving process)
  - What to send (amount and type of data)
  - Optional user-defined tag (arbitrary integer)
- Receiver calls `MPI_Recv()` specifying:
  - Where the message will come from (rank of sending process)
  - What to receive (amount and type of data)
  - Optional user-defined tag (arbitrary integer)
  - Optional status object, populated with additional information about the receive operation after it completes

Send/Receive example

```
#include <mpi.h>
int main(int argc, char ** argv) {
    int rank, data[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

    MPI_Finalize(); return 0;
}
```

Blocking communication

- `MPI_Send/MPI_Recv` are blocking communication calls
  - Return of the routine implies completion
  - Blocking communication is simple to use but can be prone to deadlocks
  - Completion implies variable sent/received can be reused/read

```
if (rank == 0) {
    MPI_Send(...)
    MPI_Recv(...)
} else { // Can deadlock here you reverse Send/Recv
    MPI_Send(...)
    MPI_Recv(...)
}
```

Non-blocking communication

- `MPI_Isend/MPI_Irecv` are non-blocking variants
  - Returns immediately, we have to test for completion separately
  - Allows overlapping computation and communication
- Semantics:

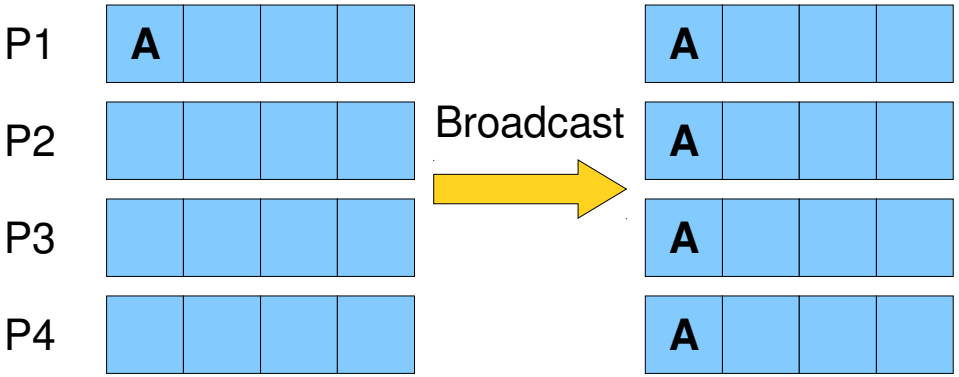
```
MPI_ISend(start, count, datatype, dest, tag, comm, request)
MPI_Irecv(start, count, datatype, src, tag, comm, request)
MPI_Wait(request, status)
```

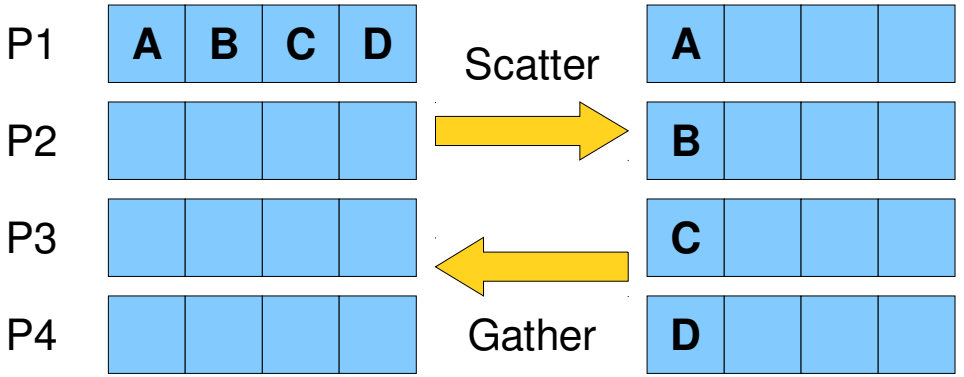
- All instances of `MPI_Send/MPI_recv` can be replaced with pars `MPI_Isend/MPI_Wait` and `MPI_Irecv/MPI_Wait`
- Blocking and non-blocking sends/receives can be combined
  - Use as a synchronization mechanism instead of barriers
- In case we need processes to exchange data, we can also use `MPI_Sendrecv()` instead of non-blocking operations

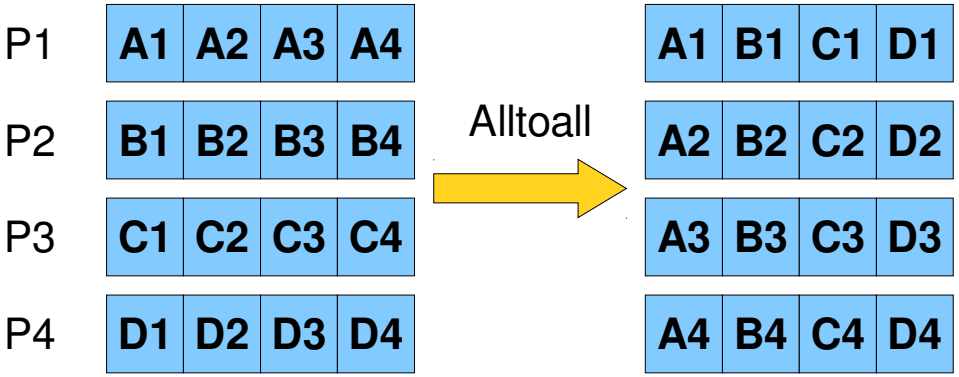
Collective operations

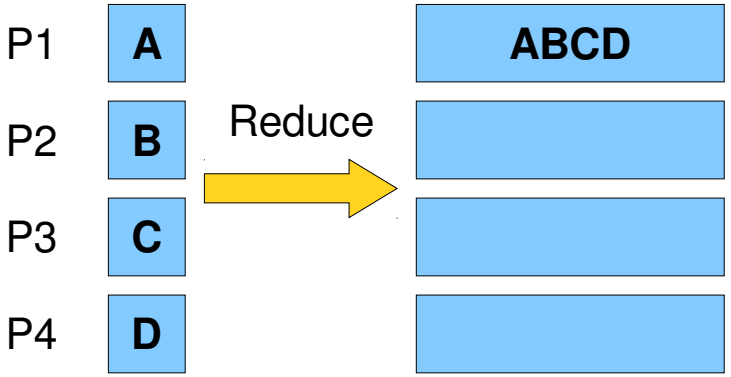
- Collective operations are called by all processes in a communicator
- Most common:
  - `MPI_Bcast()` – Broadcast (one to all)
  - `MPI_Reduce()` – Reduction (all to one)
  - `MPI_Scatter()` – Distribute data (one to all)
  - `MPI_Gather()` – Collect data (all to one)
  - `MPI_Alltoall()` – Distribute data (all to all)
- Many more
  - `MPI_Allgather, MPI_Allgatherv, MPI_Allreduce, MPI_Scan, MPI_Alltoallv, MPI_Scatterv, MPI_Gatherv, MPI_Reducescatter`
- Even more in MPI-3
  - Non-blocking collective operations
- Synchronization is also collective – `MPI_Barrier()`

Illustration of collective operations









Datatypes

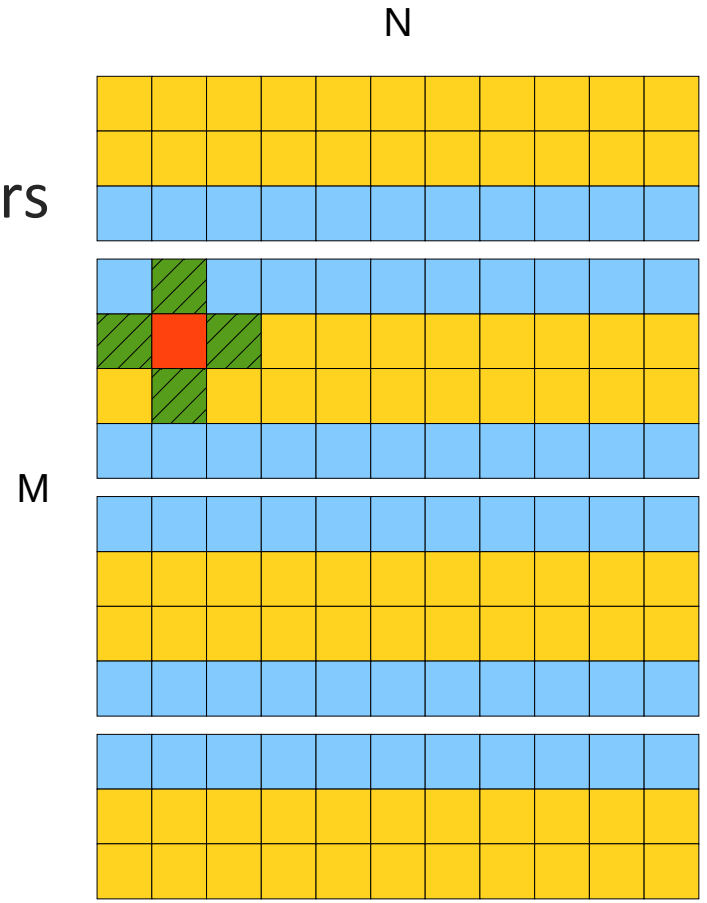
- MPI defines numerous basic datatypes, corresponding to built-in language datatypes
  - MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_BYTE, MPI_CHAR…
- Used as building blocks for *derived* datatypes
  - Contiguous array of MPI datatypes (`MPI_Type_contiguous`)
  - Strided block of datatypes (`MPI_Type_vector`)
  - Indexed array of blocks of datatypes (`MPI_Type_indexed`)
  - Arbitrary structure of datatypes (`MPI_Type_struct`)
- Derived types must be committed before use
  - `MPI_Type_commit()`

MPI input and output operations

- Multiple processes may write to separate files
  - Have to combine them manually later
- Difficult to coordinate reading/writing from/to a single file
- MPI I/O eases this
  - Single file pointer
  - Collective operations
  - Processes access relevant portion of data based on offset into the file
- Familiar semantics (open, read/write, close)
  - Open/Close: `MPI_File_open()`, `MPI_File_close()`
  - Read/Write: `MPI_File_read()`, `MPI_File_read_at()`, `MPI_File_write()`, `MPI_File_write_at()`
- Binary format is preferable
- Works great in combination with MPI derived datatypes

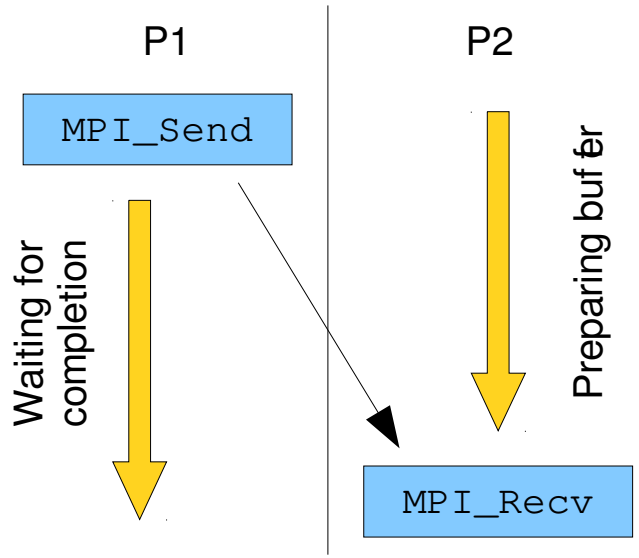
Example

- Steady state heat equation (`heat_mpi.c`)
- Slab decomposition (over M)
- Processes have to exchange data with neighbors
- MPI concepts used:
  - Initialization and finalization
  - Ghost nodes
  - Reduction (`MPI_Allreduce`)
  - Data exchanges (`MPI_Sendrecv`)
  - MPI I/O



One-sided communication

- Motivation:
  - In point-to-point communication, sender has to wait for the receiver to be ready to receive the data before it can send the data, causing delay in sending
  - Very expensive operation in blocking mode
- Idea:
  - Decouple data movement with process synchronization
  - Require only one process for data movement

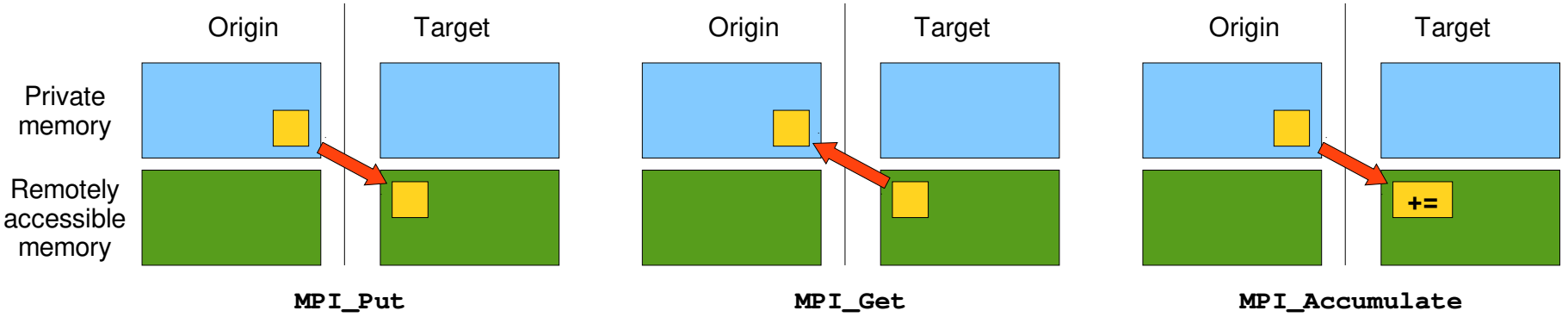


Remote Memory Access

- One-sided communication functions provide an interface to Remote Memory Access (RMA) communication methods
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory
- Many potential advantages:
  - Significantly faster than send/receive on systems with hardware support for RMA (think shared memory systems)
  - Irregular communication patterns can be more economically expressed
  - Dynamic communication pattern easier to code

One-sided communication concepts

- *Window*:
  - Each processor can make an area of memory available to one-sided transfers
  - `MPI_Win_create()` – Expose local memory to RMA operation
  - `MPI_Win_free()` – Deallocate window object
- Main functions:
  - `MPI_Put()` – Move data from local memory (*origin*) to remote memory (*target*)
  - `MPI_Get()` – Retrieve data from *target* memory into *origin's* memory
  - `MPI_Accumulate()` – Update *target* memory using local values

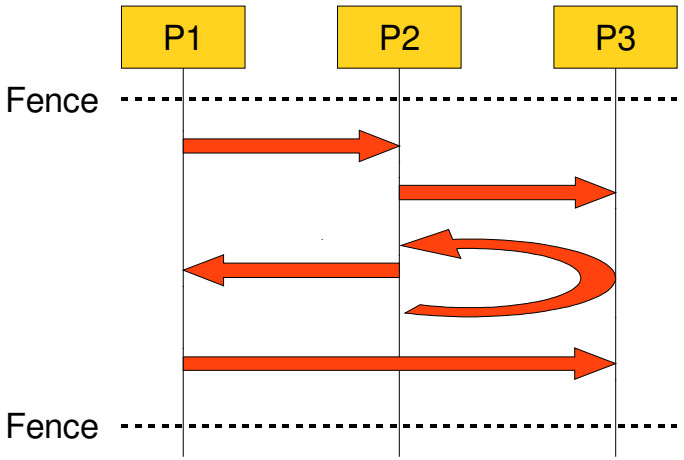


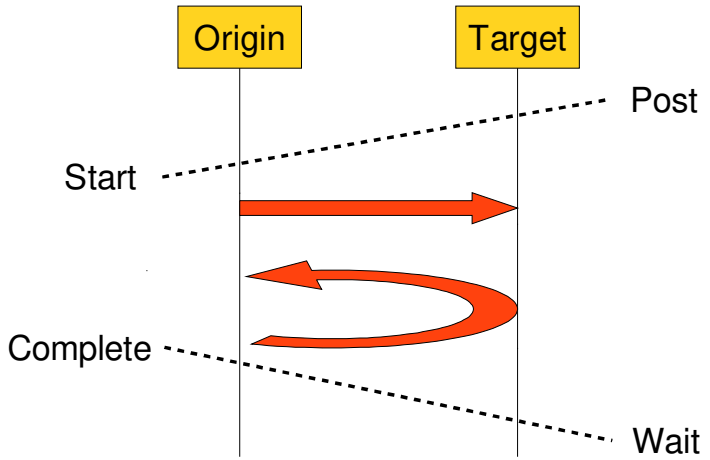
Synchronization in one-sided operations

- Data movement operations are non-blocking!
- Subsequent synchronization on window object needed to ensure operation is complete
- Data accesses occur within *epochs*
  - Epochs define ordering and completion semantics
  - Synchronization models provide mechanisms for establishing (i.e., starting and ending) epochs
- Active synchronization
  - Both origin and target participate in synchronization (declare an epoch)
- Passive synchronization
  - Only the origin is actively involved

Active synchronization

- Fence – `MPI_Win_fence()`
  - Collective synchronization model
  - Similar to `MPI_Wait()`, uses global synchronization
  - Starts and ends access and exposure epochs on all processes in the window
- Post-start-complete-wait – `MPI_Win_start()`, `MPI_Win_complete()`, `MPI_Win_post()`, `MPI_Win_wait()`
  - Finer-grained than fence, origin and target specify who they communicate with





Passive synchronization

- Only the origin process is involved in the communication
  - Communication paradigm closer to shared memory model
- Lock/Unlock
  - Origin process remotely locks/unlocks the window on the target
  - Shared and exclusive lock types (`MPI_LOCK_SHARED, MPI_LOCK_EXCLUSIVE`)

```
if (rank == 0) {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
    MPI_Put(outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);
    MPI_Win_unlock(1, win);
}
```

Example

- Steady state heat equation (`heat_rma.c`)
- Use only RMA functions
  - Window creation
  - Get/Put
  - Accumulate
  - Fences
- Better or worse than message passing?
  - Easier to access remote data
  - Accumulation is more complex than simple `MPI_Reduce()`

Hybrid programming

- Combining OpenMP and MPI within a single application
- Why hybrid?
  - Easier load balancing (with some algorithms)
  - Lower (memory) latency and data movement within node
- Why not?
  - May not always be better than pure OpenMP or MPI solution
- Modes of OpenMP/MPI operation
  - One MPI process per node
    - OpenMP threads share entire node memory, e.g., 16 threads/node on PARADOX IV
  - One MPI process per socket
    - OpenMP thread set shares socket memory, e.g., 8 threads/socket on PARADOX IV

Thread safety in (hybrid) MPI programs

- Thread safety in varies in MPI implementations
- Controlled with `MPI_Init_thread()`
  - `MPI_THREAD_SINGLE` – Only one thread will run (same as `MPI_Init`)
  - `MPI_THREAD_FUNNELED` – Processes may be multithreaded, but only the main thread can make MPI calls (MPI calls are delegated to main thread)
  - `MPI_THREAD_SERIALIZED` – Processes could be multithreaded and more than one thread can make MPI calls, but only one at a time
  - `MPI_THREAD_MULTIPLE` – Multiple threads can make MPI calls, with no restrictions

Example

- Steady state heat equation (`heat_hyb.c`)
- Combination of MPI and OpenMP
- Uses concepts presented in `heat_omp.c` and `heat_mpi.c`
- Run with single process per node
  - `mpiexec -np 4 -npernode 1 -bind-to-none ./heat_hyb ...`
- Not necessarily better performance than pure OpenMP or MPI versions

Summary

- Parallelism is the only way to achieve performance improvement with the modern hardware
- OpenMP provides for a simple, but powerful, programming model for shared memory programming
  - Fork/join model
  - Directive-based
  - Data parallelism
- MPI is the dominant model used in high-performance computing today
  - Based on message passing model…
  - …but also supports RMA-style programming
  - Industry standard with multiple high-quality implementations
- OpenMP and MPI can be combined into a hybrid programming model
- Basic concepts covered, much more left to explore

Additional resources

- OpenMP
  - LLNL OpenMP tutorial: https://computing.llnl.gov/tutorials/openMP/
  - B. Chapman et al., “Using OpenMP”, MIT Press, 2007.
  - Victor Eijkhout’s tutorial: http://pages.tacc.utexas.edu/~eijkhout/pcse/html/
- MPI
  - LLNL MPI tutorial: https://computing.llnl.gov/tutorials/mpi/
  - W. Gropp et al., “Using MPI”, MIT Press, 2014.
  - W. Gropp et al., “Using Advanced MPI”, MIT Press, 2014.
- Code examples
  - John Burkardt’s OpenMP and MPI examples
    - https://people.sc.fsu.edu/~jburkardt/c_src/openmp/openmp.html
    - https://people.sc.fsu.edu/~jburkardt/c_src/mpi/mpi.html
  - http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/