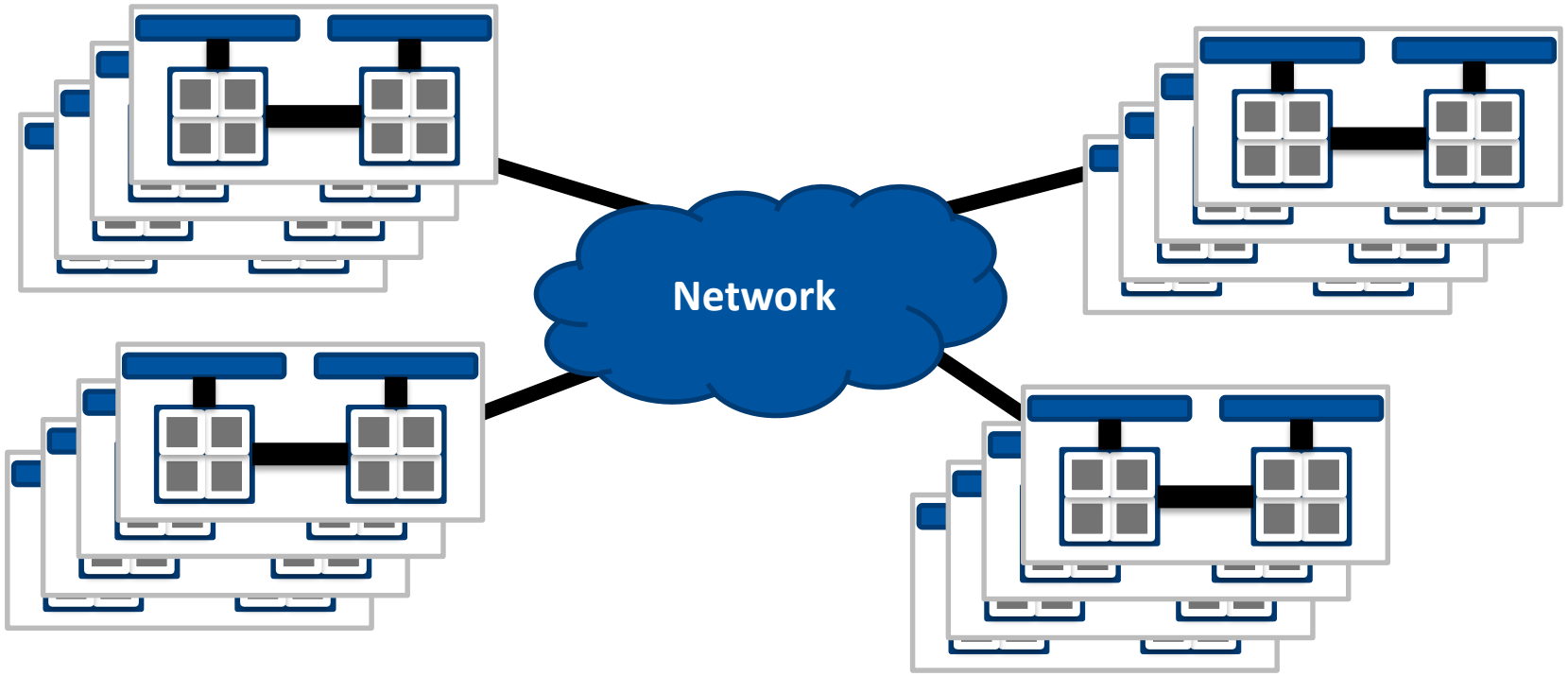
Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. Distributed-memory programming with MPI
9. **Hybrid programming (MPI + OpenMP)**
   - → **Hybrid programming basics**
   - → MPI + OpenMP
   - → MPI and threads
   - → Addressing remote threads
   - → Hybrid programs and LSF
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Motivation for hybrid programming

- **Clusters of small supercomputers**
  - → Increasingly complex nodes – many cores, GPUs, Intel® Xeon Phi™, etc.

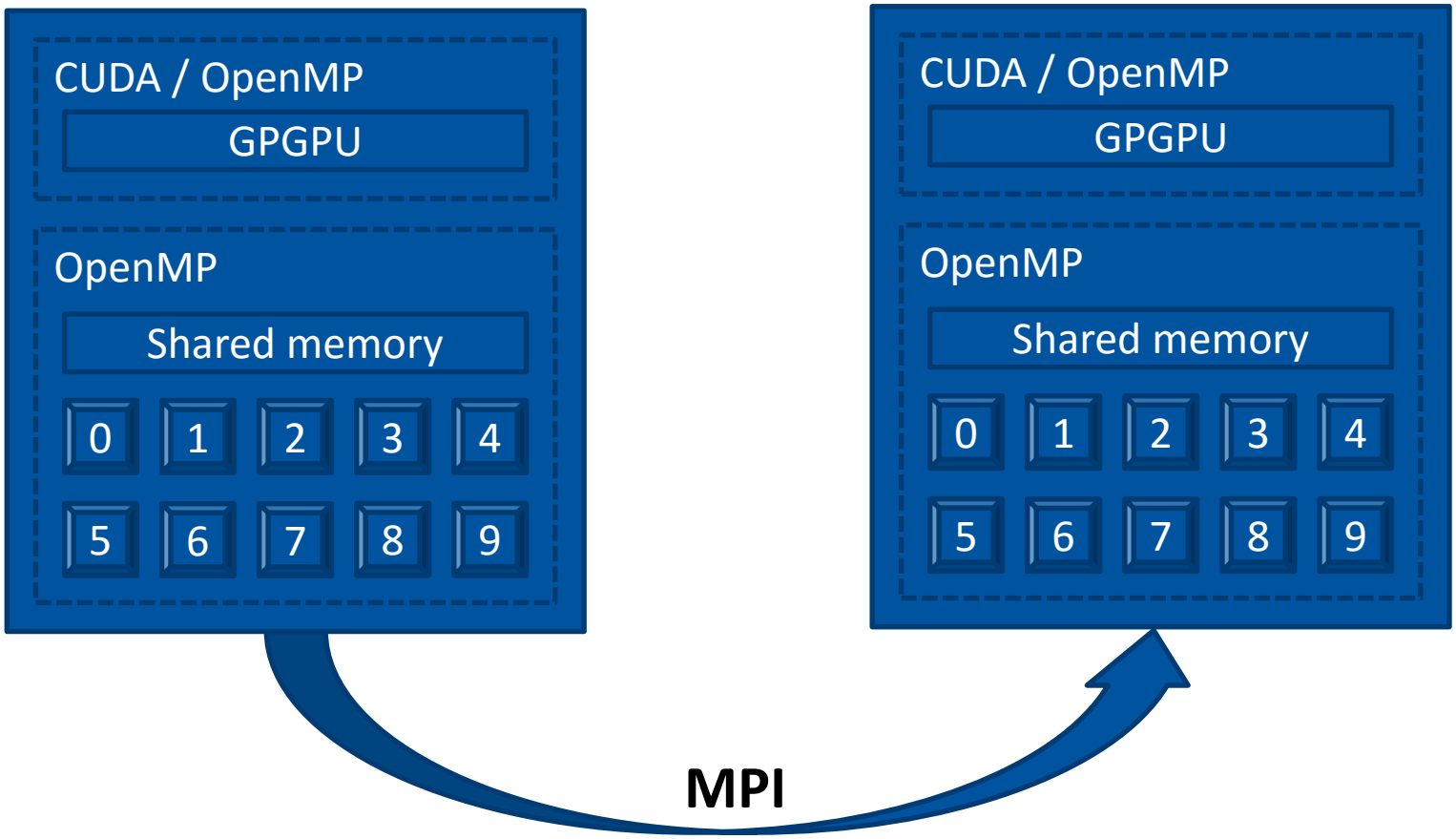


MPI on shared-memory nodes

- **MPI is sufficiently abstract to run on a single node**
  - → It doesn't matter where the processes are located
  - → Message passing implemented using shared memory and IPC
    - → The MPI library takes care of data transfer
    - → Usually much faster than sending messages over the network
  - → but…
- **… it is far from optimal**
  - → MPI processes are implemented as separate OS processes
  - → Portable data sharing is hard to achieve
  - → Lots of control / domain data has to be duplicated
  - → Reduced cache utilization

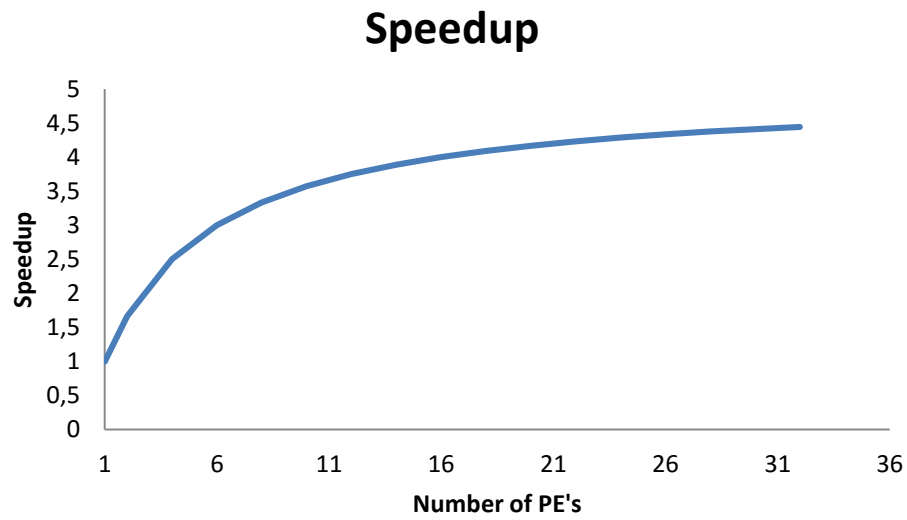
Hybrid programming

- **(Hierarchical) mixing of different programming paradigms**

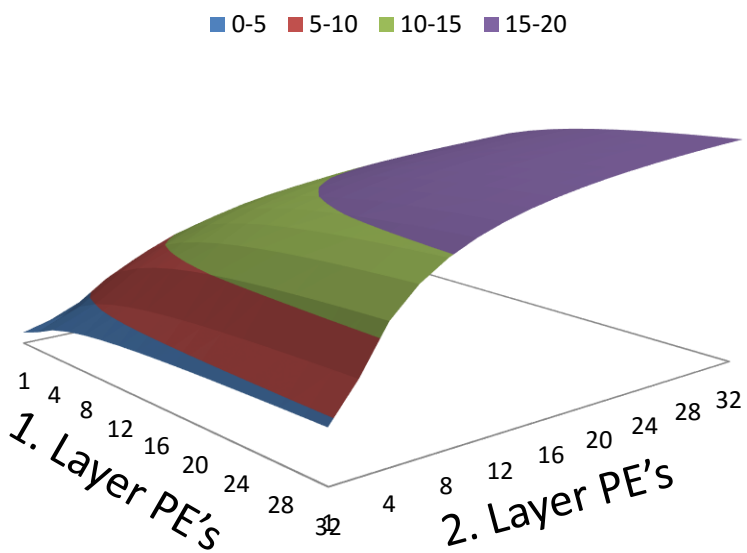


Motivation for hybrid programming

- **Speedup with Amdahl's Law:**

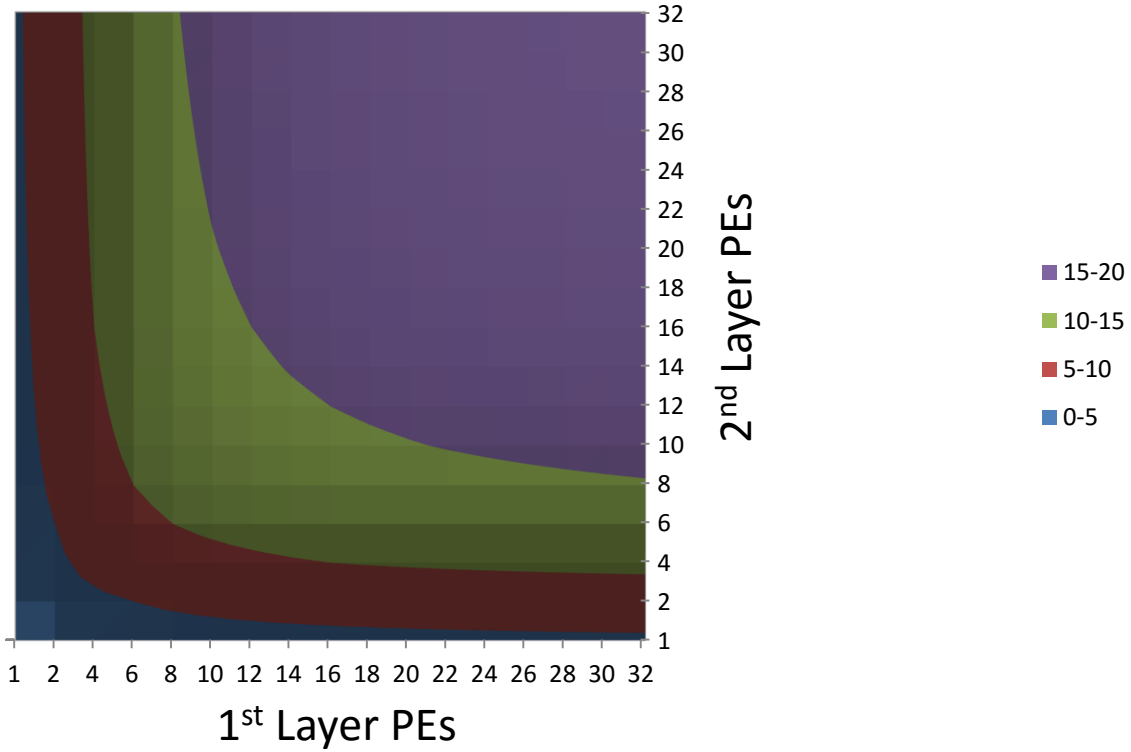


- **Hybrid parallelization adds an additional layer of possible speedup:**



Motivation for hybrid programming

- **Orthogonal projection of the speedup surface:**



- **Depending on the granularity of the parallelized code the speedup increases further due to the use of multiple parallelization layers**

Best of all worlds

- **OpenMP / pthreads**
  - → Last-level cache reuse, data sharing
  - → Simple programming model (not so simple with POSIX threads)
  - → Threaded libraries (e.g., MKL)
- **OpenACC / OpenCL / CUDA / OpenMP for accelerators**
  - → Massively parallel GPGPU accelerators and co-processors
- **MPI**
  - → Fast transparent networking
  - → Scalability
- **Downsides**
  - → Code can become hard to maintain
  - → Harder to debug and tune

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. Distributed-memory programming with MPI
9. **Hybrid programming (MPI + OpenMP)**
   - → Hybrid programming basics
   - → **MPI + OpenMP**
   - → MPI and threads
   - → Addressing remote threads
   - → Hybrid programs and LSF
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

MPI and OpenMP

- **The most widely adopted hybrid programming model in engineering and scientific codes**

	no MPI	MPI
no OpenMP	serial code	MPI code
OpenMP	OpenMP code	hybrid code

MPI+OpenMP programming models

- **Basically the idea of a hybrid MPI+OpenMP programming model is to spawn a team of OpenMP threads inside each MPI process**

- **Following the general guidelines for OpenMP parallelization, the pragmas are applied to computationally expensive parts of the MPI process first (e.g. loops, etc.)**

- **Synchronization is still restricted to the use of appropriate MPI calls (MPI_Barrier, MPI_Reduce, …), as there is no OpenMP functionality to achieve synchronization between different MPI processes**

- **We will consider two different hybrid programming approaches:**
  - → Vector mode
  - → Task mode

MPI+OpenMP programming models: Vector mode (1/3)

- **All MPI calls are made outside concurrent execution regions**
  - → Outside parallel regions, i.e. in the “serial” part of the OpenMP code, or
  - → Inside constructs which are executed by one thread only (**section**, **master**, appropriate **if** operators)

- **Existing MPI code can be “made” hybrid by applying OpenMP parallelization worksharing directives to time-consuming loops and considering adequate NUMA placement of the data**

- **No interference between OpenMP and MPI may occur. Both paradigms are programmed independently**

MPI+OpenMP programming models: Vector mode (2/3)

- **Vector mode implementations are similar to programming parallel vector computers with MPI where the inner layer of parallelization (vector processing) is independent of the MPI parallelization**

- **Especially applications for which the number of MPI processes is somehow limited by the problem constraints may benefit from vector mode models**
  - → E.g. the all-to-all collective scales like $O(N^2)$ (N – number of MPI processes)
  - → Increasing parallelization by parallelizing lower levels through the use of OpenMP is the only way to go beyond the MPI limit

MPI+OpenMP programming models: Vector mode (3/3)

- **Pseudocode for a vector mode implementation:**

```
[…]
#pragma omp parallel for
for(k = 0; k < N; k++)
{
   // Parallel work done here by OpenMP
   // (e.g. updates over a local stencil)
}

// Halo exchange done by MPI
MPI_Irecv(halo data from –dir neighbour)
MPI_Isend(data to +dir neighbour)

MPI_Irecv(halo data from +dir neighbour)
MPI_Isend(data to -dir neighbour)
MPI_Waitall();
```

MPI+OpenMP programming models: Task mode (1/3)

- **The task mode is a more general model of hybrid programming**
  - → Allows any kind of MPI communication inside OpenMP parallel regions
  - → Based on the thread safety requirements, the MPI standard defines three different levels of interaction between OpenMP and MPI
  - → Code must check first if any/which of these levels are supported by the MPI implementation

- **Task mode is especially useful for functional task decomposition and decoupling of computation and communication**

MPI+OpenMP programming models: Task mode (2/3)

- **The task mode provides a high level of flexibility, but**
  - → blows up code size
  - → increases code complexity

- **As neither OpenMP nor MPI implement mechanisms for the task mode approach usually OpenMP is programmed in a way similar to MPI**
  - → Explicit OpenMP tasks could run in any worker thread (untied tasks can even be migrated while executing), therefore hard to address the messages
  - → Different functional tasks need to be mapped to individual OpenMP thread ID's using case switch constructs
  - → Often the convenient OpenMP worksharing directives can not be used

MPI+OpenMP programming models: Task mode (3/3)

- **Sample pseudocode for the task mode model:**

```
#pragma omp parallel
{
  threadID = omp_get_thread_num();
  nthreads = omp_get_num_threads();
  for (iteration=0; iteration<MAXITER; ++iteration)
  {
    if (threadID == 0)
    {
      // Standard Jacobi iteration
      // Update boundary cells
      …
      // Exchange halo cells
      MPI_Irecv(halo data from –dir neighbour)
      MPI_Isend(data to +dir neighbour)
      MPI_Irecv(halo data from +dir neighbour)
      MPI_Isend(data to -dir neighbour)
      MPI_Waitall();
    }
```

```
    else
    {
      // Remaining threads perform update of
      // INNER cells 1,…,N-1
      // Distribute outer loop iterations manually:
      chunksize = (N-1)/(nthreads-1) + 1;
      my_k_start = 1+(threadID-1)*chunksize;
      my_k_end = min(my_k_end, (N-1));
      for (k=my_k_start; k<my_k_end; ++k)
        for (j=1; j<(N-1); ++j)
          for (i=1; i<(N-1); ++i)
          {
             …
          }
    }
    #pragma omp barrier
  }
}
```

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. Distributed-memory programming with MPI
9. **Hybrid programming (MPI + OpenMP)**
   - → Hybrid programming basics
   - → MPI + OpenMP
   - → **MPI and threads**
   - → Addressing remote threads
   - → Hybrid programs and LSF
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

MPI – threads interaction

- **Most MPI implementation are threaded (e.g., to implement non-blocking requests), but *not thread-safe***

- **Four levels of threading support**

Higher levels

Level identifier	Description
`MPI_THREAD_SINGLE`	Only one thread may execute
`MPI_THREAD_FUNNELED`	Only the main thread may make MPI calls
`MPI_THREAD_SERIALIZED`	Only one thread may make MPI calls at a time
`MPI_THREAD_MULTIPLE`	Multiple threads may call MPI at once with no restrictions

- **All implementations support `MPI_THREAD_SINGLE`, but not all fully support `MPI_THREAD_MULTIPLE`**

MPI and threads – initialization (1/2)

- **Initialize MPI with thread support**

```
C:       int MPI_Init_thread (int *argc, char ***argv,
                              int required, int *provided)
Fortran: SUBROUTINE MPI_INIT_THREAD (required, provided,
                                     ierr)
```

→ **`required`** tells MPI what level of thread support is required

→ **`provided`** tells us what level of threading MPI actually provides

→ could be lower or higher than the required level

→ **`MPI_INIT`** – same as a call with **`required = MPI_THREAD_SINGLE`**

→ The thread that called **`MPI_INIT_THREAD`** becomes the *main thread*

→ The level of thread support cannot be changed later

MPI and threads – initialization (2/2)

- **Obtain the current level of thread support**

```
int MPI_Query_thread (int *provided)
```

→ **`provided`** is filled with the current level of thread support

→ If **`MPI_INIT_THREAD`** was called, **`provided`** will equal the value returned by the MPI initialisation call

→ If **`MPI_INIT`** was called, **`provided`** will equal an implementation specific value

- **Find out if current thread is the main one**

```
int MPI_Is_thread_main (int *flag)
```

MPI_THREAD_FUNNELED

- **Only make MPI calls from outside the OpenMP parallel regions**
  - → Corresponds to the vector model

```
double data[], localData[];

MPI_Scatter(data, count, MPI_DOUBLE,
            localData, count, MPI_DOUBLE,
            0, MPI_COMM_WORLD);

#pragma omp parallel for
for (int i = 0; i < count; i++)
   localData[i] = exp(localData[i]);

MPI_Gather(localData, count, MPI_FLOAT,
           data, count, MPI_DOUBLE,
           0, MPI_COMM_WORLD);
```

No MPI calls from inside the parallel region

MPI_THREAD_FUNNELED

- **Only make MPI calls from outside the OpenMP parallel regions or inside an OpenMP MASTER construct**

```
float data[];

#pragma omp parallel
{
   #pragma omp master
   {
      MPI_Bcast(data, count, MPI_FLOAT, 0, MPI_COMM_WORLD);
   }
   #pragma omp barrier

   #pragma omp for
   for (int i = 0; i < count; i++)
      data[i] = exp(data[i]);
}
```

MPI_THREAD_SERIALIZED

- **MPI calls should be serialized within (named) OpenMP CRITICAL constructs or using some other synchronisation primitive**

```
#pragma omp parallel
{
   MPI_Status status;

   #pragma omp critical(mpicomm)
   MPI_Recv(&data, 1, MPI_FLOAT, MPI_ANY_SOURCE,
            0, MPI_COMM_WORLD, &status);

   result = exp(data);

   #pragma omp critical(mpicomm)
   MPI_Send(&result, 1, MPI_FLOAT, status.MPI_SOURCE,
            0, MPI_COMM_WORLD);
}
```

MPI_THREAD_MULTIPLE

- No need to synchronize between MPI calls
- Watch out for possible data races and MPI race conditions

```
#pragma omp parallel
{
   int count; float *buf;
   MPI_Status status;

   MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
   MPI_Get_count(&status, MPI_FLOAT, &count);

   buf = new float[count];
   MPI_Recv(buf, count, MPI_FLOAT, status.MPI_SOURCE,
            0, MPI_COMM_WORLD);
}
```

- MPI_Probe could match the same message in multiple threads!

Potential trouble in hybrid MPI applications

- ***The call to MPI_FINALIZE should occur in the main thread. The call should occur only after all process threads have completed their MPI calls, and have no pending communications or I/O operations.***
- ***A program where two threads block, waiting on the same request, is erroneous. Similarly, the same request cannot appear in the array of requests of two concurrent MPI_WAIT{ANY|SOME|ALL} calls. In MPI, a request can only be completed once. Any combination of wait or test which violates this rule is erroneous.***
- ***A receive call that uses source and tag values returned by a preceding call to MPI_PROBE or MPI_IPROBE will receive the message matched by the probe call only if there was no other matching receive after the probe and before that receive. In a multithreaded environment, it is up to the user to enforce this condition using suitable mutual exclusion logic. This can be enforced by making sure that each communicator is used by only one thread on each process.***

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. Distributed-memory programming with MPI
9. **Hybrid programming (MPI + OpenMP)**
   - → Hybrid programming basics
   - → MPI + OpenMP
   - → MPI and threads
   - → **Addressing remote threads**
   - → Hybrid programs and LSF
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Addressing in hybrid programs

- **MPI and OpenMP have orthogonal addressing spaces:**
  - → MPI rank ∈ [0, #procs-1] `MPI_Comm_rank()`
  - → OpenMP thread ID ∈ [0, #threads-1] `omp_get_thread_num()`
  - → Hybrid rank:thread ∈ [0, #procs-1] × [0, #threads-1]

- **MPI does not provide a direct way to address threads in a process**

Envelope Field	Value Source	Remark
source rank	Sender process rank	Fixed, contains the rank of the sending process
destination rank	Send call argument	Only one rank per process per communicator
tag	Send call argument	Free to choose
communicator	Send call argument	Many communicators can coexist

Addressing in hybrid programs

- **Using tags to address specific threads**
  - → The MPI standard requires that tags provide at least 15 bits of user defined information (the **`MPI_TAG_UB`** attribute of **`MPI_COMM_WORLD`** could be queried in order to obtain the actual upper limit)

- **Use tag value to address destination thread**
  - → (+) straightforward to implement – threads simply supply their ID as tag value in the call to **`MPI_Recv`**
  - → (+) very large number of threads could be addressed
  - → (-) not possible to further differentiate messages
  - → (-) no information about the sending thread ID

Addressing in hybrid programs

- **Using tags to address specific threads**
  - → The MPI standard requires that tags provide at least 15 bits of user defined information (the **`MPI_TAG_UB`** attribute of **`MPI_COMM_WORLD`** could be queried in order to obtain the actual upper limit)

- **Multiplex destination thread ID and tag value**
  - → e.g., 7 bits for tag value (0..127), 8 bits for thread ID (up to 256 threads)
  - → (+) possible to differentiate messages
  - → (-) receive operation must be split into **`MPI_Probe`** / **`MPI_Recv`** pair if we would like to emulate **`MPI_ANY_TAG`**
  - → (-) no information about the sending thread ID

Addressing in hybrid programs

- **Tags for threads addressing**
  - → The MPI standard requires that tags provide at least 15 bits of user defined information (the **`MPI_TAG_UB`** attribute of **`MPI_COMM_WORLD`** could be queried in order to obtain the actual upper limit)

- **Multiplex sender and destination thread IDs and tag value**
  - → For communication between two BCS nodes @ RWTH 14 bits are required for thread IDs (up to 128 threads – 7 bits)
  - → Suitable for MPI implementations that provide larger tag space
    - → both Open MPI and Intel MPI have **`MPI_TAG_UB`** of 2^{31}-1
  - → (+) sender information retained together with possible message differentiation
  - → (-) code might not be portable to other MPI implementations

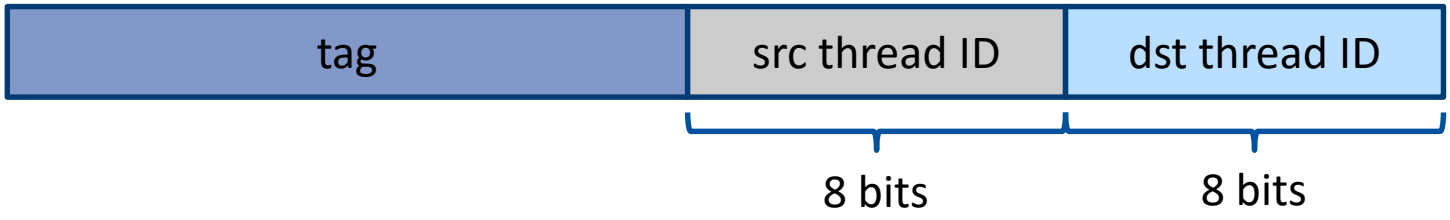
Addressing in hybrid programs

- **Sample implementation – sender thread**

```
#define MAKE_TAG (tag,stid,dtid) \
    (((tag) << 16) | ((stid) << 8) | (dtid))

// Send data to drank:dtid with tag

MPI_Send(data, count, MPI_FLOAT, drank,
         MAKE_TAG(tag, omp_get_thread_num(), dtid),
         MPI_COMM_WORLD);
```



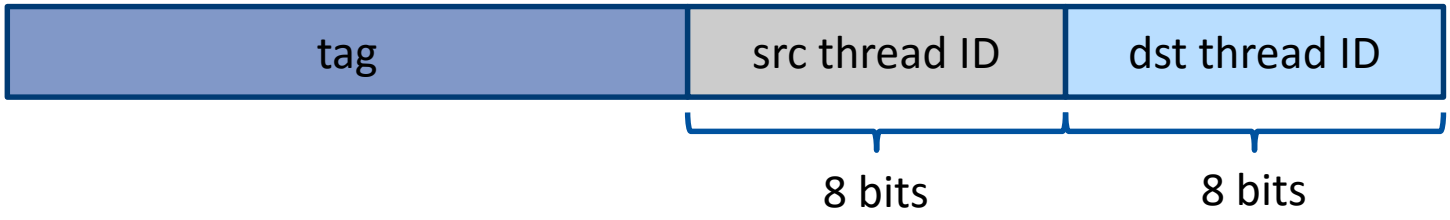
Addressing in hybrid programs

- **Sample implementation – receiver thread**

```
#define MAKE_TAG (tag,stid,dtid) \
    (((tag) << 16) | ((stid) << 8) | (dtid))

// Receive data from srank:stid with a specific tag

MPI_Recv(data, count, MPI_FLOAT, srank,
         MAKE_TAG(tag, stid, omp_get_thread_num()),
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Addressing in hybrid programs

- **Sample implementation with wildcard tag support**

```
#define GET_TAG(val) \
    ((val) >> 16)
#define GET_SRC_TID(val) \
    (((val) >> 8) & 0xff)
#define GET_DST_TID(val) \
    ((val) & 0xff)

// Receive data from srank:stid with any tag

MPI_Probe(srank, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
if (GET_SRC_TID(status.MPI_TAG) == stid &&
    GET_DST_TID(status.MPI_TAG) == omp_get_thread_num())
{
    MPI_Recv(data, count, MPI_FLOAT, srank, status.MPI_TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Addressing in hybrid programs

- **Sample implementation with wildcard tag support**
  - → A significant drawback of this approach is that all threads must keep pumping out messages as MPI provides no way to peek further in the receive queue when wildcard tags are used.
  - → Effectively the reception of messages by the different threads is serialized.
  - → Busy threads delay other threads.

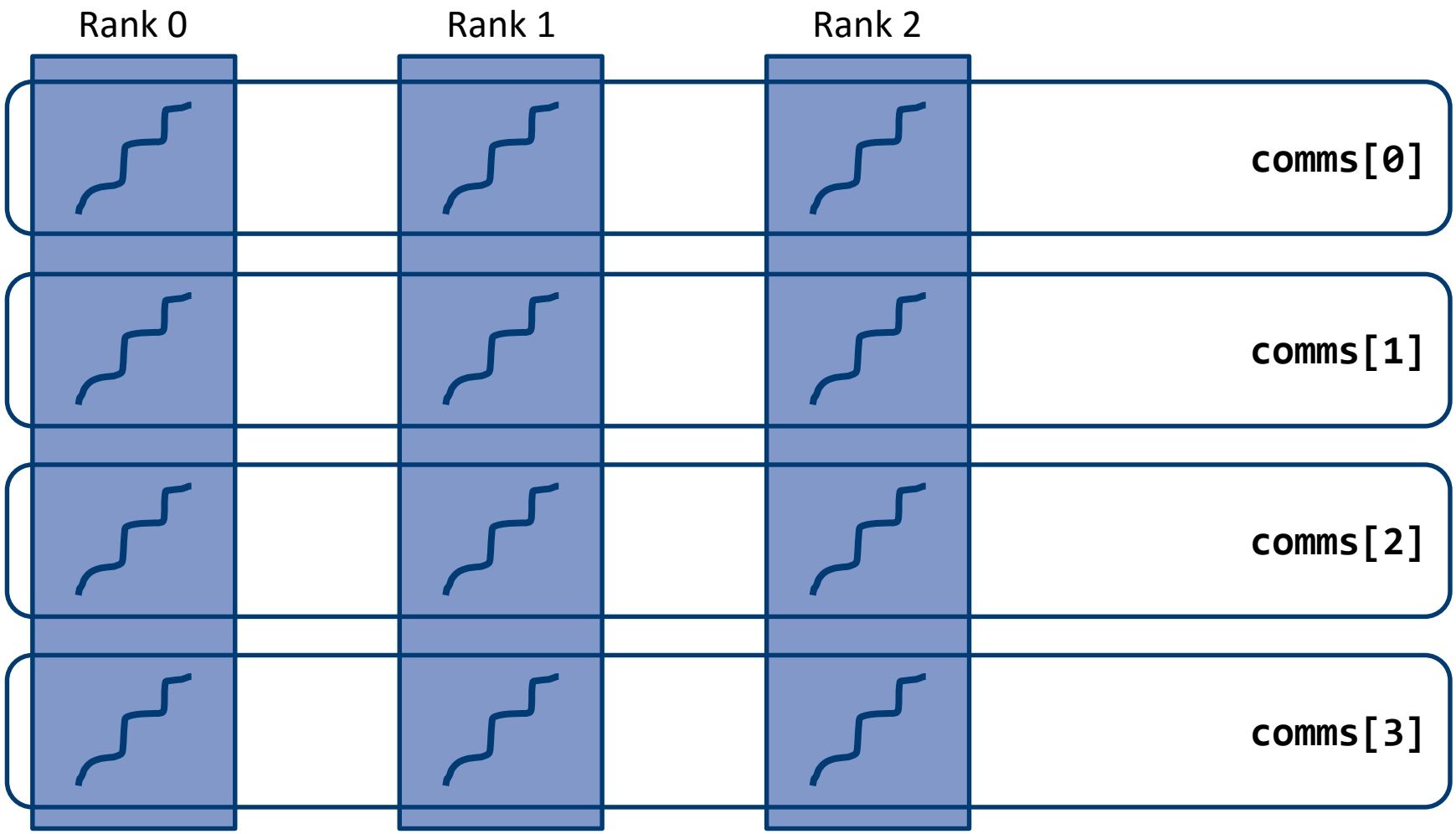
More potential trouble

- ***Matching of collective calls on a communicator, window, or file handle is done according to the order in which the calls are issued at each process. If concurrent threads issue such calls on the same communicator, window or file handle, it is up to the user to make sure the calls are correctly ordered, using inter-thread synchronization.***

- ***Finally, in multithreaded implementations, one can have more than one, concurrently executing, collective communication call at a process. In these situations, it is the user's responsibility to ensure that the same communicator is not used concurrently by two different collective communication calls at the same process.***

Multiple communicators (1/3)

- **In some cases it makes more sense to use multiple communicators**



Multiple communicators (2/3)

- **Easy to create with multiple calls to MPI_Comm_dup**

- **Each MPI process is member of all communicators**
  - → Threads can send messages in any communicator
  - → Threads receive messages only in the communicator that corresponds to their thread ID

- **Allows wildcard receives without the synchronizing side effect of the previous approach**

Multiple communicators (3/3)

- **Sample implementation**

```
MPI_Comm comms[nthreads], tcomm;

#pragma omp parallel private(tcomm) num_threads(nthreads)
{
int tid = omp_get_thread_num();
MPI_Comm_dup(MPI_COMM_WORLD, &comms[tid]);
tcomm = comms[tid];
. . .
// Sender
MPI_Send(data, count, MPI_FLOAT, drank, tid, comms[dtid]);
. . .
// Receiver
MPI_Recv(data, count, MPI_FLOAT, srank, stid, tcomm,
         &status);
. . .
}
```

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. Distributed-memory programming with MPI
9. **Hybrid programming (MPI + OpenMP)**
   - → Hybrid programming basics
   - → MPI + OpenMP
   - → MPI and threads
   - → Addressing remote threads
   - → **Hybrid programs and LSF**
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

MPI thread support @ RWTH cluster

- **Thread support in various MPI libraries**

requested	Open MPI	Open MPI mt	Intel MPI w/o -mt_mpi	Intel MPI w/ -mt_mpi
SINGLE	SINGLE	SINGLE	SINGLE	SINGLE
FUNNELED	SINGLE	FUNNELED	SINGLE	FUNNELED
SERIALIZED	SINGLE	SERIALIZED	SINGLE	SERIALIZED
MULTIPLE	SINGLE	MULTIPLE*	SINGLE	MULTIPLE

- **Open MPI**
  - → Use module versions that end with **`mt`** (e.g. **`openmpi/1.10.4mt`**)
  - → *The native InfiniBand transport doesn't work with **`MPI_THREAD_MULTIPLE`**!
- **Intel MPI**
  - → Enabled when compiling with **`-openmp`**, **`-parallel`**, or **`-mt_mpi`**

Running hybrid programs with LSF

- **Sample hybrid job script (Open MPI)**

```
#!/usr/bin/env zsh
# 16 MPI procs x 6 threads = 96 cores
#BSUB -x
#BSUB -a openmpi
#BSUB -n 16
# 12 cores/node / 6 threads = 2 processes per node
#BSUB -R “span[ptile=2]”

module switch openmpi openmpi/1.10.4mt
# 6 threads per process
export OMP_NUM_THREADS=6
# Pass OMP_NUM_THREADS on to all MPI processes
$MPIEXEC $FLAGS_MPI_BATCH –x OMP_NUM_THREADS \
         program.exe <args>
```

- **For the most up to date information refer to the HPC Primer**

Running hybrid programs with LSF

- **Sample hybrid job script (Intel MPI)**

```
#!/usr/bin/env zsh
# 16 MPI procs x 6 threads = 96 cores
#BSUB -x
#BSUB -a intelmpi
#BSUB -n 16
# 12 cores/node / 6 threads = 2 processes per node
#BSUB -R “span[ptile=2]”

module switch openmpi intelmpi

# 6 threads per process
# Pass OMP_NUM_THREADS on to all MPI processes
$MPIEXEC $FLAGS_MPI_BATCH –genv OMP_NUM_THREADS=6 \
         program.exe <args>
```

- **For the most up to date information refer to the HPC Primer**

What you have learned

- **Hybrid programming basics**
  - → Why we need hybrid programs
  - → Hybrid programming models
- **Threading modes of MPI**
  - → What levels of thread support MPI provides
  - → Potential troubles with multithreaded MPI programs
- **Addressing multiple threads within MPI processes**
  - → How to work around the flat addressing space of MPI
  - → Using multiple communicators in a hybrid context
- **Running hybrid programs on the RWTH cluster**
  - → How to properly instruct LSF to run your hybrid job

What was omitted

- **How to program accelerators and coprocessors**
  - → OpenMP 4.0 makes it easy to move from threaded MPI hybrid programs to MPI programs that use accelerators and/or coprocessors.
  - → Also possible to use another programming model, e.g. CUDA, OpenACC, etc.

- **How to use and to develop tools to avoid bugs in hybrid programs**
  - → Parallel debuggers help
  - → Not many MPI tools fully understand OpenMP yet

References

- **Tobias Hilbrich, Matthias S. Müller, Bettina Krammer: Detection of Violations to the MPI Standard in Hybrid OpenMP/MPI Applications. IWOMP 2008: 26-35**