PARALLEL PROGRAMMING...

By Patrick Lemoine 2025.

Kokkos

Κόκκος

SESSION 3

Kokkos?

Basic Concepts

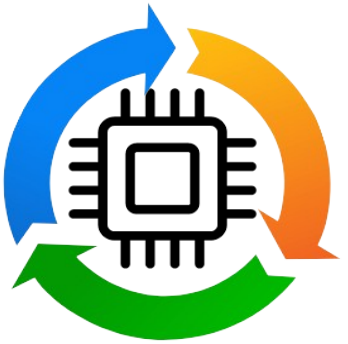
- Views,
- Memory spaces,
- Memory access patterns,
- Mirrors

Advanced Concepts

- Asynchronicity, Streams and Task Parallelism
- Multi-level hierarchical parallelism
- Multidimensional Loops and Data Structures
- Advanced Reductions
- MPI (Message Passing Interface) and PGAS (Partitioned Global Address Spaces)
- SIM (Single Instruction Multiple Data) Task Flow

Kokkos MPI

(Message Passing Interface)



Kokkos MPI: Concept

Basic concept

Kokkos MPI represents the integration of two complementary parallel programming paradigms:

- Kokkos: Manages intra-node parallelism, providing an abstraction for heterogeneous architectures (multi-core CPUs, GPUs).
- MPI: Manages inter-node parallelism, enabling communication between distributed processes.

This combination enables the development of applications capable of fully exploiting modern high-performance computing systems, from the node level to massive clusters.

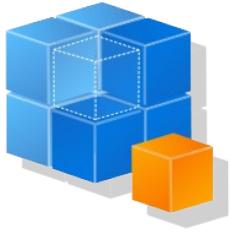
Advantages of the Kokkos MPI approach

- **Performance portability**: The code can run efficiently on various architectures without modification.
- **Scalability**: Allows you to scale from a few cores to thousands of compute nodes.
- **Abstraction of the material:**Kokkos handles the low-level details of the architecture, while MPI handles the communication.
- **Automatic optimizations:**Kokkos can automatically optimize the use of local resources.

Kokkos MPI: MPI and Kokkos integration

Kokkos integrates with MPI for inter-node communications, enabling scalability across distributed systems. A typical Kokkos application initialization with MPI looks like this:

```
#include <mpi.h>
#include <Kokkos_Core.hpp>

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);

  int mpi_rank, mpi_size;
  MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

  Kokkos::initialize(argc, argv);
  {
    // Application code
  }
  Kokkos::finalize();

  MPI_Finalize();
  return 0;
}
```



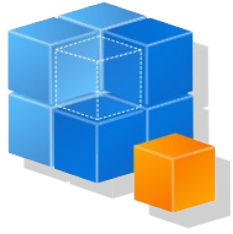
Kokkos MPI: Data Management and Communication

Creating and Using Kokkos Views with MPI

```
// Creating a Kokkos view
Kokkos::View<double*>local_data("local_data", local_size);

// Filling the view
Kokkos::parallel_for("fill_data", local_size,
KOKKOS_LAMBDA(const int i) {
   local_data(i) = mpi_rank * local_size + i;
});

// MPI Communication
if (mpi_rank == 0) {
   Kokkos::View<double*> global_data("global_data", local_size *
mpi_size);
   MPI_Gather(local_data.data(), local_size, MPI_DOUBLE,
            global_data.data(), local_size, MPI_DOUBLE,
            0, MPI_COMM_WORLD);
}
```

Using Kokkos Views with CUDA-aware MPI

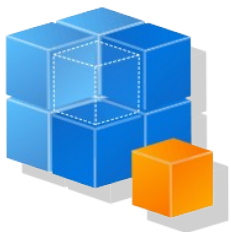
```
#ifdef KOKKOS_ENABLE_CUDA
   Kokkos::View<double*, Kokkos::CudaSpace>gpu_data("gpu_data", size);
   // Populate gpu_data on GPU
   if (mpi_rank == 0) {
      MPI_Recv(gpu_data.data(), size, MPI_DOUBLE, 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   } else if (mpi_rank == 1) {
      MPI_Send(gpu_data.data(), size, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);
   }
#endif
```



Kokkos MPI: Advanced Optimizations

Calculation-communication recovery:

```
Kokkos::View<double*>send_buffer("send_buffer", buffer_size);
Kokkos::View<double*>recv_buffer("recv_buffer", buffer_size);

MPI_Request send_request, recv_request;

// Start communications in a non-blocking manner
MPI_Isend(send_buffer.data(), buffer_size, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD, &send_request);
MPI_Irecv(recv_buffer.data(), buffer_size, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &recv_request);

// Perform calculations while communications are in progress
Kokkos::parallel_for("compute", compute_size, KOKKOS_LAMBDA(const int i) {
   // Local calculations
});

// Wait for communications to end
MPI_Wait(&send_request, MPI_STATUS_IGNORE);
MPI_Wait(&recv_request, MPI_STATUS_IGNORE);
```



Kokkos MPI: Advanced Optimizations

Using multiple Kokkos runspaces:

```
Kokkos::DefaultExecutionSpacedefault_space;
Kokkos::DefaultHostExecutionSpacehost_space;

Kokkos::View<double*, Kokkos::DefaultExecutionSpace>device_data("device_data", size);
Kokkos::View<double*, Kokkos::DefaultHostExecutionSpace>host_data("host_data", size);

// Calculations on the device
Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::DefaultExecutionSpace>(default_space, 0, size),
   KOKKOS_LAMBDA(const int i) {
      device_data(i) = /* calculation */;
   });

// Copy to host for MPI communication
Kokkos::deep_copy(host_data, device_data);

// MPI communication using host data
MPI_Allreduce(MPI_IN_PLACE, host_data.data(), size, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

// Copy back to device
Kokkos::deep_copy(device_data, host_data);
```

Kokkos MPI: Optimizations and Considerations

Thread Management and Affinity: For optimal performance, it is crucial to bind MPI tasks and threads to physical cores. This can be achieved using specific MPI flags or environment variables.

Using Kokkos Views for MPI Communications: Kokkos views can be used directly in MPI calls, simplifying data management between CPU and GPU.

Calculation-communication recovery: Kokkos enables the use of multiple execution spaces, facilitating the overlap of GPU computations with MPI communications.

Choice of granularity: There is a balance to be struck between the number of MPI tasks and the utilization of GPUs. Generally, it is better to have one MPI task per GPU rather than multiple MPI tasks sharing a GPU.

Portability: One of the main advantages of Kokkos is the portability of the code. The same code can be compiled for different architectures (CPU, NVIDIA GPU, AMD GPU) without major modifications.

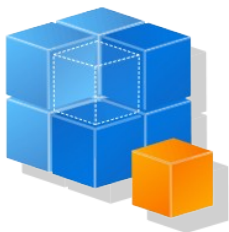
Kokkos MPI: The Challenges

Programming complexity: Using MPI, Kokkos, and GPUs together can increase code complexity. It is recommended to take a phased approach, starting with the GPU implementation before adding MPI.

Memory management: Efficient memory management between the CPU and GPU remains a challenge, requiring careful design of data structures and transfers.

Load balancing: Work balancing between GPUs and MPI nodes should be optimized to maximize overall performance.

```
struct WorkItem { /* ... */ };
Kokkos::View<WorkItem*>work_queue("work_queue", total_work);

while (!work_queue.empty()) {
   int chunk_size = adaptive_chunk_size();
   Kokkos::parallel_for(chunk_size, KOKKOS_LAMBDA(const int i) {
      process_work_item(work_queue[i]);
   });
   update_work_distribution(MPI_COMM_WORLD);
}
```



Kokkos MPI : The Challenges

Advanced Performance Portability: Using adaptive execution policies:

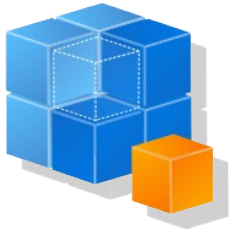
```
auto policy = Kokkos::AUTO;
if (Kokkos::DefaultExecutionSpace::concurrency() > threshold) {
    policy = Kokkos::TeamPolicy<>(N, Kokkos::AUTO);
} else {
    policy = Kokkos::RangePolicy<>(0, N);
}
Kokkos::parallel_for(policy, kernel);
```

Advanced memory management:

- Using Kokkos unified memory for heterogeneous systems:

```
Kokkos::View<double*, Kokkos::CudaUVMSpace>unified_data("unified_data", N);
// Efficiently accessible from CPU and GPU
```

- Challenge: Managing implicit memory migrations
- Solution: Explicitly prefetch data to the correct device



Kokkos MPI : The Challenges

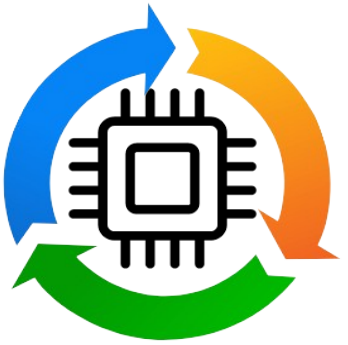
Advanced multi-level parallelism:

```
Kokkos::parallel_for(Kokkos::TeamPolicy<>(N, Kokkos::AUTO), KOKKOS_LAMBDA(const member_type& team_member)
{
    const int league_rank = team_member.league_rank();
    const int team_size = team_member.team_size();
    const int team_rank = team_member.team_rank();

    Kokkos::parallel_for(Kokkos::TeamThreadRange(team_member, M), [&](const int& i) {
      Kokkos::parallel_for(Kokkos::ThreadVectorRange(team_member, K), [&](const int& j) {
        // Vector work
      });
    });
  });
```

Resilience and fault tolerance:

- Challenge: Managing hardware failures at exascale
- Approach: Integrating checkpoint mechanisms with Kokkos and MPI

Kokkos PGAS (Partitioned Global Address Space)



Kokkos PGAS (Partitioned Global Address Space): Concept

PGAS Concept: The Partitioned Global Address Space (PGAS) model is a parallel programming paradigm that combines the advantages of shared memory and distributed memory. In this model:

1. The global address space is logically partitioned.
2. Each partition is associated with a specific process or thread.
3. Processes can access local memory faster than remote memory.

Kokkos PGAS:extends this concept by integrating it into the Kokkos ecosystem:

1. High-level abstraction: Allows multi-GPU and multi-node programming without explicitly handling communications.
2. Compatibility: Integrates with existing Kokkos abstractions (Views, runspaces, etc.).
3. Performance: Aims to deliver hardware-like performance while simplifying programming.
4. Portability: Works on different architectures (CPU, GPU) and with different backends (SHMEM, NVSHMEM, MPI One-sided).

Key components of Kokkos PGAS

1. DefaultRemoteMemorySpace: New memory space for remote access.
2. PartitionedLayout: Specific layout for distributed data.
3. RemoteSpaces: Execution spaces for operations on remote memory.
4. Extended API: New functions to manage distributed memory (get_local_range, get_range, etc.).

Kokkos MPI vs PGAS

In terms of performance under Kokkos C++, the choice between MPI and PGAS depends on the specific application context and the target hardware architecture. Here are the key points to consider:

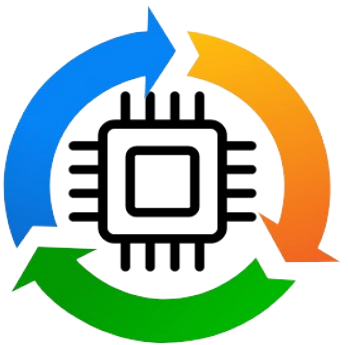
MPIis generally more mature and widely used for distributed programming. It offers good performance for two-sided communications and is well optimized on many platforms.

PGAS, on the other hand, can offer performance benefits for certain use cases:

- It better exploits data locality, which can improve efficiency and scalability compared to traditional shared memory approaches.

- PGAS enables one-sided communication operations, which can be more efficient in some scenarios.

Kokkos Tasking, Streams and SIMD



Kokkos Tasking, Streams, and SIMD (Single Instruction Multiple Data)

Tasking

- Management of asynchronous tasks and dependencies
- Allows the expression of complex task graphs

Streams

- Concurrent execution of independent kernels
- Optimizes the use of hardware resources

SIMD (Single Instruction Multiple Data)

- Portable primitive for explicit vectorization
- Works on CPU and GPU without performance loss

Benefits

- Increased portability between heterogeneous architectures
- Performance optimization via explicit vectorization
- Flexibility in the expression of parallelism

Kokkos Tasking, Streams, and SIMD (Single Instruction Multiple Data)

Using Tasking:

```
Kokkos::TaskScheduler<> scheduler(Kokkos::DefaultExecutionSpace());
auto task1 = scheduler.create_task([] (const member_type& member) {
   // Code for task 1
});
auto task2 = scheduler.create_task([] (const member_type& member) {
   // Code for task 2
});
task2.depend(task1);
scheduler.spawn(task1);
scheduler.spawn(task2);
```

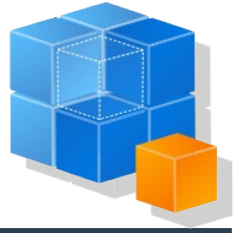
Using Streams:

```
Kokkos::Cuda cuda_space1, cuda_space2;
Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Cuda>(cuda_space1, 0, N), KOKKOS_LAMBDA(int i) {
   // Kernel 1
});
Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Cuda>(cuda_space2, 0, M), KOKKOS_LAMBDA(int i) {
   // Kernel 2
});
```

Kokkos Tasking, Streams, and SIMD (Single Instruction Multiple Data)

Advanced use of Tasking:

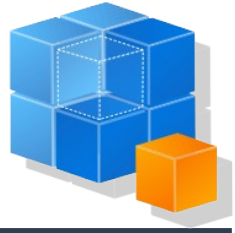
```
Kokkos::TaskScheduler<Kokkos::DefaultExecutionSpace>sched
uler(Kokkos::DefaultExecutionSpace());

// Creating a task with dependencies
auto task1 = scheduler.create_task([] (const
Kokkos::TaskScheduler<>::member_type& member) {
    // Code for task 1
    return 0; // Return value indicating success
});

auto task2 = scheduler.create_task([] (const
Kokkos::TaskScheduler<>::member_type& member) {
    // Code for task 2
    return 0;
});

auto task3 = scheduler.create_task([] (const
Kokkos::TaskScheduler<>::member_type& member) {
    // Code for task 3
    return 0;
});

// Defining dependencies
task3.depend(task1, task2);

// Launching tasks
self future1 = scheduler.spawn(task1);
auto future2 = scheduler.spawn(task2);
auto future3 = scheduler.spawn(task3);

// Wait for all tasks to complete
Kokkos::wait(scheduler);

// Checking the results
if (future3.is_ready()) {
    std::cout << "All tasks are completed" << std::endl;
}
```



Advanced use of Streams:

```
// Create two separate CUDA runspaces
Kokkos::Cudacuda_space1;
Kokkos::Cudacuda_space2;

// Allocate memory on the device
Kokkos::View<double*, Kokkos::CudaSpace>d_a("d_a", N);
Kokkos::View<double*, Kokkos::CudaSpace>d_b("d_b", N);
Kokkos::View<double*, Kokkos::CudaSpace>d_c("d_c", N);

// Running kernels on different streams
Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Cuda>(cuda_space1, 0, N),
   KOKKOS_LAMBDA(int i) {
      d_a(i) = i * 1.0;
   }
);

Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Cuda>(cuda_space2, 0, N),
   KOKKOS_LAMBDA(int i) {
      d_b(i) = i * 2.0;
   }
);
```

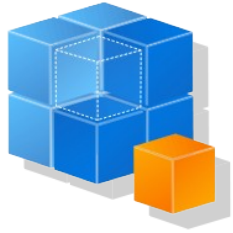
```
// Synchronize streams if necessary
cuda_space1.fence();
cuda_space2.fence();

// Using the results of both streams
Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Cuda
>(cuda_space1, 0, N),
   KOKKOS_LAMBDA(int i) {
      d_c(i) = d_a(i) + d_b(i);
   }
);
```



Advanced use of SIMD:

```
#include <Kokkos_SIMD.hpp>

// Define a SIMD type for double
using simd_double =Kokkos::Experimental::native_simd<double>;

// Function using SIMD
KOKKOS_FUNCTION void vector_add(const simd_double& a, const
simd_double& b, simd_double& result) {
    result = a + b;
}
```

```
// Usage in a Kokkos kernel
Kokkos::parallel_for("SIMD_example", N / simd_double::size(),
KOKKOS_LAMBDA(const int i) {
    simd_double a, b, c;

    // Loading data into SIMD
    Kokkos::Experimental::load(a, &data_a[i * simd_double::size()]);
    Kokkos::Experimental::load(b, &data_b[i * simd_double::size()]);

    // SIMD operation
    vector_add(a, b, c);

    // Storing the result
    Kokkos::Experimental::store(c, &data_c[i * simd_double::size()]);
});
```



Thank you for your attention !