# 1 Advanced Topics

## 1.1 Cancellation Points in OpenMP

On slide 6, a new concept called *cancellation points* were introduced. They are a new concept since OpenMP 4.0, and they allow the programmer to dynamically stop the execution of a parallel region based on a condition. This is useful when the parallel region is running for a long time and the programmer wants to stop it if a certain condition is met, like one solution (out of possible many or even infinite) is found. The syntax for cancellation points is as follows:

Use `#pragma omp cancel for` to signal cancellation to all necessary threads. The `for` can be replaced with `parallel` or `sections` depending on the context. Now that we have signaled the cancellation, we also have to check for it in all the threads, which is done with `#pragma omp cancellation point for`. If a cancellation is signaled, all threads will stop executing the corresponding parallel region and continue with the code after the parallel region.

The following example demonstrates the use of cancellation points in OpenMP:

```
#pragma omp parallel // start parallel region
{
#pragma omp for schedule(dynamic)
    for (int i = 0; i < biggest_possible_number; ++i) {
        if (is_solution(i)) { // find some solution, not necessary the
             smallest
            final_solution = i;
#pragma omp cancel for // signal cancellation, because we found a
     solution
    }
#pragma omp cancellation point for // check for cancellations signalled
     from other threads
    // cancel for loop if cancellations signalled
    }
} // end parallel region
```

Figure 1: Example of using cancellation points in OpenMP (slide 6)

Now, the reader may ask why we need additional directives for this behavior, couldn't we just use a global variable and check it in the parallel region? Well, we definitely could do that, but the cancellation points are more efficient and easier to use. The OpenMP runtime system can check the cancellation condition at the end of the loop iterations, and if the condition is met, it will stop the execution of the parallel region. This is more efficient than checking the condition in every iteration of the loop, and it is also easier to use since the programmer doesn't have to write the code for checking the condition. Furthermore, imagine the scenario of a for loop with *dynamic* scheduling. If we used a global variable for the cancellation condition, we would only cancel the current batch of iterations the thread is working on, and potentially there are many more batches of iterations left which are not yet cancelled. Needless to say, this is not very efficient.

## 1.2 CUDA

CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general-purpose processing. The CUDA platform is used in many scientific applications, where high performance is required. It is also used in many machine learning applications, where large amounts of data need to be processed quickly. The CUDA platform provides a set of libraries and tools for developing parallel applications, and it is supported by many programming languages, including C, C++, and Fortran. Thus, let us lake a look at the basic structure of a CUDA program shown in figure 2.

The CUDA code in figure 2 demonstrates the basic structure of a CUDA program that performs element-wise addition of two arrays. It begins with a kernel function, `add`, which defines the computation executed on the GPU. The kernel uses a unique thread index, calculated from `blockIdx.x` and `threadIdx.x`, to identify the array element each thread should process. In the `main` function, memory is allocated on both the host (CPU) and the device (GPU), and data is initialized and transferred to the GPU. The kernel is launched using a grid of blocks, with each block containing multiple threads, enabling parallel processing of the arrays. After the kernel execution, the results are copied back from the device to the host for verification. Finally, all allocated memory is freed. This structure illustrates key concepts of CUDA programming, including thread management, memory allocation, data transfer, and kernel execution.

## 1.3 MPI

MPI stands for Message Passing Interface, and it is a standard for parallel programming. It is used for distributed memory systems, where each process has its own memory. The processes communicate with each other by sending messages. The MPI library is available in many programming languages like C, C++, and Fortran, and is used in many scientific applications, where high performance is required. To this end, MPI provides functions for sending and receiving messages, and for synchronizing the processors.

One of the fundamental features of MPI is its ability to send and receive messages between processors. The `MPI_Send` function is used to send data from one processor to another, while `MPI_Recv` is used to receive data. These functions ensure point-to-point communication, which is critical for many parallel algorithms. For example, `MPI_Send` requires parameters such as the data buffer, count of elements, data type, destination rank, tag, and communicator. Similarly, `MPI_Recv` specifies the source rank, among other parameters, to successfully retrieve messages. Figure 3 demonstrates how one might use these functions.

In this little program every process sends one message to the next one based on their rank. The data that is send is a single integer which acts as an accumulator. The program starts with process 0, which sends the data to process 1. Process 1 receives the data, adds its rank to it, and sends it to process 2. This continues until the last process, which sends the data back to process 0, which receives it and prints the final result.

Beyond basic point-to-point communication, MPI also provides more advanced collective communication methods. One such function is `MPI_Gather`, which allows a root processor to collect data from multiple processors and combine it into a single array. This is particularly useful for operations like gathering results from parallel computations into a central location for further analysis or visualization.

```
// Kernel function to add elements of two arrays
__global__ void add(int *a, int *b, int *c, int n) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < n) {
        c[index] = a[index] + b[index];
    }
}

int main() {
    int n = 10240; // Array size
    int size = n * sizeof(int);

    // Allocate memory on the host (CPU)
    int *h_a = (int *)malloc(size);
    int *h_b = (int *)malloc(size);
    int *h_c = (int *)malloc(size);

    // Initialize host arrays
    for (int i = 0; i < n; i++) {
        h_a[i] = i;
        h_b[i] = i * 2;
    }

    // Allocate memory on the device (GPU)
    int *d_a, *d_b, *d_c;
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Copy data from host to device
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    // Define the number of threads per block and blocks per grid
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;

    // Launch kernel
    add<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, n);

    // Check for errors in kernel launch
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("CUDA Error: %s\n", cudaGetErrorString(err));
        return -1;
    }

    // Copy result back to host
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    for (int i = 0; i < n; i++) {
        printf("%d + %d = %d\n", h_a[i], h_b[i], h_c[i]);
    }
    // Free memory
    ...

    return 0;
}
```

Figure 2: Basic CUDA program

```
#include <iostream>
#include <assert.h>
#include <mpi.h>

#define MPI_Assert(error) assert(error == MPI_SUCCESS)

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

    int l_rank, l_comm_size, l_data = 0;

    MPI_Comm_rank(MPI_COMM_WORLD, &l_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &l_comm_size);

    MPI_Status status;

    // program logic
    if (l_rank) {
        MPI_Assert(MPI_Recv(&l_data, 1, MPI_INT, l_rank-1, l_rank-1,
            MPI_COMM_WORLD, &status));
        std::cout << "process " << l_rank << " received data " << l_data
            << " from process " << status.MPI_SOURCE << ".\n";
    }
    l_data += l_rank;
    std::cout << "process " << l_rank << " sends data " << l_data << "
        to process " << (l_rank+1) % l_comm_size << ".\n";
    MPI_Assert(MPI_Send(&l_data, 1, MPI_INT, (l_rank+1) % l_comm_size,
        l_rank, MPI_COMM_WORLD));
    if (l_rank == 0) {
        MPI_Assert(MPI_Recv(&l_data, 1, MPI_INT, l_comm_size-1,
            l_comm_size-1, MPI_COMM_WORLD, &status));
        std::cout << "process " << l_rank << " received data " << l_data
            << " from process " << status.MPI_SOURCE << ".\n";
    }
    MPI_Finalize();
}
```

Figure 3: Basic MPI program