





Master in High Performance Computing

Performance analysis of GPU-to-GPU communications in distributed FFTs in Quantum ESPRESSO

Supervisor(s):
Ivan Girotto,
Fabrizio Ferrari-Ruffino, Pietro Delugas, Laura Bellentani

Candidate:
Francesco Andreucci

10th edition
2023–2024



Contents

1 Preface **5**

2 NCCL **9**
2.1 The NCCL library 9
2.1.1 Initialization of NCCL 10
2.1.2 Bootstrapping of the communicator 10
2.1.3 Topology detection and path optimization 11
2.2 NCCL Kernels 12
2.2.1 Point-to-point directives 14
2.3 Benchmark details and results 15
2.3.1 Benchmark on Leonardo cluster 15

3 Density Functional Theory and Fourier transforms **25**
3.1 Density functional theory and the Kohn-Sham equation 25
3.1.1 The Hohenberg-Kohn theorems 26
3.1.2 The Kohn-Sham approach 26
3.2 Plane-waves based Kohn-Sham approach 28
3.2.1 Kohn-Sham equation in Fourier space 29
3.2.2 The cut-off and its consequences 31
3.2.3 Solution of the Kohn-Sham equation 32

4 FFTXlib library and benchmarks **35**
4.1 Real space and reciprocal space grids 35
4.2 FFT implementation in FFTXlib 37
4.2.1 Domain decomposition 37
4.2.2 Inverse FFT procedure 39
4.2.3 The batching mechanism 39
4.2.4 NCCL implementation 40
4.3 Results of the benchmarks on the Leonardo cluster 42

4.3.1 HPC-X performance with default batching 43
4.3.2 HPC-X performance with custom batching 43
4.3.3 NCCL performance 44

5 Conclusions **47**
5.0.1 Future developments 48

Appendices **49**

A Crash course on lattices and Fourier transform **51**
A.0.1 The reciprocal lattice 53
A.0.2 Fourier transform on lattices 54

Chapter 1

Preface

QUANTUM ESPRESSO is a widely used open-source software for computation of electronic structure and atomistic simulations of materials [1, 2]. It is based on the Plane Wave Pseudopotential (PW) approach of Density Functional Theory (DFT). One of the most time consuming steps of this method lies in the computation of Fast Fourier Transforms (FFTs), which can account for between 30% and 70% of the total time of a single simulation. The QUANTUM ESPRESSO library devoted to the computation of FFTs is FFTXlib, which supports both serial and distributed transforms. This thesis focuses on the latter case, specifically on multi-node FFTs. Since in QUANTUM ESPRESSO typical use cases rarely exceed grids of $(1000)^3$ points, the distributed FFT can, in principle, suffer from performance issues related to the small communication buffers, which prevent full exploitation of the network bandwidth. Fortunately, in the PW procedure, it is necessary to concurrently compute the FFT of thousands of functions. This allows an overlap between computation and communication for uncorrelated groups of function, processing them at the same time as a single larger buffer (this procedure goes under the name of *batching*). Consequently, this unlocks a new level of parallelism, improving both communication efficiency and network performance.
The focus of this thesis is the communication step of the distributed FFT, which consists of an all-to-all communication pattern. Until now, the only option present in the code was to execute this communication using GPU-aware MPI-based libraries, such as openMPI, MPICH, HPC-X (the NVIDIA optimized version of openMPI [3]). Indeed, the latter has been demonstrated to outperform OpenMPI and to support GPUdirect Peer-to-Peer in collective communications. The main contribution of this thesis consists in implementing the same communication pattern using the NVIDIA Collective Communication Library (NCCL) [4]. The latter is a relatively

new library made by NVIDIA to optimize collective GPU-to-GPU communication according to the specific hardware of the machine on which the code is running.

The output of this thesis can be summarized in two parts:

- Several miniapps were written to compare the performance of both point-to-point and collective communication patterns using NCCL and GPU-aware MPI, using HPC-X as MPI library.

- The backend of FFTXlib for distributed GPU runs was modified to leverage NCCL and its performance was benchmarked against the standard MPI-GPU implementation built with HPCX-MPI

These comparisons were performed on the CINECA EUROHPC JU Leonardo cluster. Regarding the first point above, the miniapps were written from scratch to ensure a greater control over initialization, instrumentation for profiling and performance comparison. The standard OSU tests [5] analyses the communication time for several buffer sizes at once, which complicates the profiling of the application. By using the JUBE [6] benchmarking tool to organize the jobs, it is possible to get a finer control over the launch of jobs related to different buffer sizes and other configuration parameters. Furthermore, these results are not only useful for establishing a baseline to compare with the FFTXlib benchmarks, but also independently valuable, as there are few available benchmarks analyses of NCCL on Leonardo [7].

The content of the thesis is structured as follows.
Chapter 2 focuses on the NCCL library. After a brief overview of the library itself, the results of the comparison benchmarks between NCCL and HPC-X obtained on the Leonardo cluster are presented.
Chapter 3 introduces the theory of DFT and the PW method, highlighting why Fourier transforms are so critical in Quantum ESPRESSO . This Chapter is complemented by Appendix *A*, which contains a broad overview of crystal lattices and the related Fourier analysis.
Chapter 4 describes the workings of FFTXlib, focusing in particular on the communication pattern and the batching mechanism, and how it allows for an overlap of computation and communication in the distributed FFT. The implementation of the NCCL communication pattern is described in detail, followed by the comparison of the benchmark results between this version and the one based on HPC-X.

Chapter 5 contains the conclusions and perspectives for further development of the thesis work.

Most of the results presented in this thesis have been obtained during my work as a researcher for the "Programming models solutions for exascale efficiency" task (3.2) of the MaX ("MAterials design at the eXascale") European Center of Excellence. As such, they will be part of MAX continuous reporting activity and of the final report of the initiative.

Chapter 2

NCCL

This chapter is devoted to the description and benchmarking of the Nvidia Collective Communications Library (NCCL). NCCL is an open source [4] inter-GPU communication library based on CUDA, designed to optimize GPU-to-GPU communications, according to the specific properties of the underlying hardware. NCCL is already integrated in the back-end of several popular libraries, such as PyTorch[8], Tensorflow[9].
The first part of this chapter provides a qualitative understanding of the inner workings of the library, following the official documentation [10] and the the NVIDIA webinars [11, 12]. It is worth noting that the official documentation is mainly focused on explaining how to use the library's API, as the actual implementation of NCCL frequently changes with each version of the library. Nonetheless, it is useful to provide, even if only in broad strokes, an overview of how NCCL functions.
The second part of the chapter is devoted to the presentation of the results obtained by benchmarking NCCL against MPI on the Leonardo cluster for three different communication patterns: point-to-point, all-reduce and all-to-all.

2.1 The NCCL library

This section describes the required steps to set up NCCL, starting with the initialization process. The C API of NCCL will be used, but a Fortran interface is also available [13].

2.1.1 Initialization of NCCL

The collective communication patterns handled by NCCL involve a group of GPUs, each identified by a rank. The first step in the initialization of NCCL is the the creation of a communicator, which is an object used by each rank to refer to all the GPUs which will take part in the communication.
It is assumed that an MPI communicator has already been initialized and that each GPU has been assigned to one MPI process. Other configurations are possible, such as associating multiple GPUs to one MPI process.

2.1.2 Bootstrapping of the communicator

The initialization of NCCL begins with the creation of a unique identifier to mark all the ranks the communicator. To generate this identifier, a root rank is selected, in this case rank 0, and the function `ncclGetUniqueId` is called:

```
1  ncclUniqueId id;
2  if (myRank == 0) ncclGetUniqueId(&id);
3  MPI_Bcast(&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD);
```

In this snippet, rank 0 generates an Id, which identifies the communicator and contains additional information, for example the IP port/address of the NIC[1] associated to the root. This Id is then transmitted to the other ranks using `MPI_Bcast`.
Furthermore, when calling `ncclGetUniqueId`, NCCL spawns a thread, called "bootstrap thread", that remains idle while waiting for incoming connections from the other ranks.
Finally, the addresses of the other ranks must be collected by calling the function (from each rank, root included) `ncclCommInitRank`:

```
1  ncclComm_t comm;
2  ncclCommInitRank(&comm, nRanks, id, myRank);
```

This function connects each rank to the bootstrap thread on the root process and communicates, for example, the IP port/address of the other participants.
The outcome of this first step of the initialization is the creation of a communication channel based on IP sockets (and thus out-of-band communication), which is then

[1] NIC stands for "Network Interface Controller" (or "Network Interface Card") and is the circuit board that handles the connection to a network.

used to exchange information between the ranks.

2.1.3 Topology detection and path optimization

NCCL gathers various data about the intra-node hardware: CPU model and architecture (using `cpuid` and `arch`), PCI (inspecting `/sys`), NVLink (using NVLM), InfiniBand (using IB Verbs) . This data is then used by NCCL to build a graph representing the node.[2]
The second step is a multi-path search to find the optimal path(s) on this graph based on the nominal performance of the detected components. These paths are constrained to go through all the GPUs inside the node and are called *chains*.
The beginning and the end of the chains can vary. An optimal choice is to start and end the chain on the same NIC, if allowed by the internal topology of the node, anyway the goal is always to maximize network locality. At this stage NCCL has identified several non-competing optimal paths within the node connecting all the GPUs.

Inter-node topology

Once the intra-node chains have been established, NCCL connects the optimal paths (and therefore the nodes) in various ways to implement different algorithms for collective communication. These algorithms always assume that all the equivalent NICs on each node can communicate with each other on independent channels, called *rails*. This independence is important to avoid competition for resources and to maximize the bandwidth. For the rails to be truly independent each set of NICs should in theory have its own, separate, network switch, or the cluster should have adaptive routing enabled to automatically switch to a non-active NIC if some NICs gets overloaded by traffic. Ideally, the only exchanges between different NIC rails should happen intra-node, where communication is typically very fast thanks, for example, to NVlink.

[2]The information collected by NCCL can be dumped in an xml file by using the environment variable `NCCL_TOPO_DUMP_FILE`.

Examples of topologies

Two possible NCCL inter-node connection topologies are presented here, the ring and the tree,[3] as depicted in Fig. 2.1.
The ring topology is illustrated in Fig. 2.1 (a): in this case the nodes are connected so that the NICs on a rail form a ring. Specifically, the output coming from, NIC i of node k will flow into NIC i of node $k+1$, and so on.
The tree topology, illustrated in Fig. 2.1 (b), is slightly more complex than the ring. Although the NICs still communicate on rails, the nodes are connected to form two dual binary trees, as depicted schematically in Fig. 2.2.
Two trees are needed for the following reason: if only one tree were used, each NIC would have at most one incoming data flow and two outgoing ones, which would halve the total bandwidth. On the other hand, using two dual binary trees (the nodes in one tree become the leaves in the dual one and vice versa) allows each node to have two incoming and two outgoing data flows. This allows for the data to be split evenly across the two trees and thus to achieve full bandwidth exploitation.

2.2 NCCL Kernels

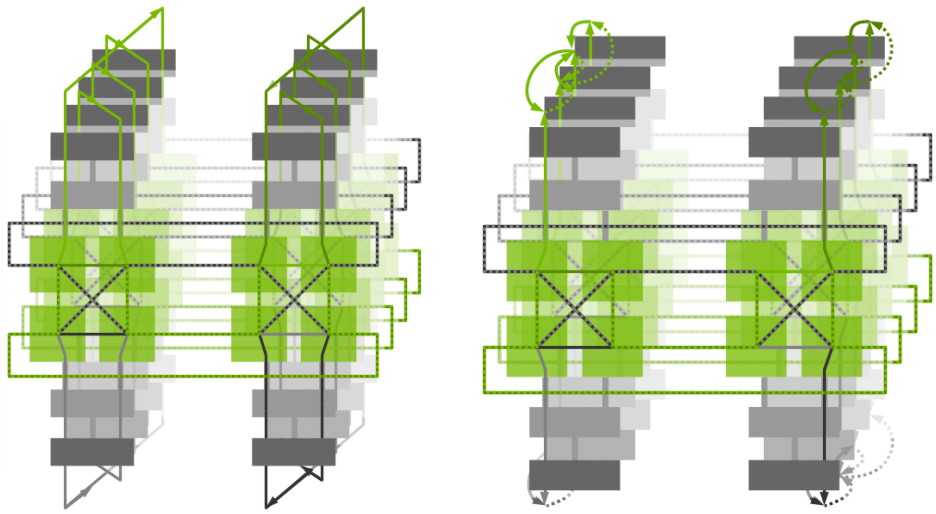
The NCCL API for collective communication routines closely mimics the structure of the corresponding MPI calls. For instance, the interface for the all-reduce operation is:

```
1    ncclall-reduce(const void* sendbuff, void* recvbuff, size_t count,
2                   ncclDataType_t datatype, ncclRedOp_t op,
3                   ncclComm_t comm, cudaStream_t stream)
```

The main difference from the corresponding MPI call is the presence of a `cudaStream` argument. When an NCCL routine is invoked, a CUDA kernel[4] is launched on the specified stream and executes the communication over the optimal

[3]The current set of possible algorithm can be read on the NCCL documentation page: `https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-algo`. In theory, it is possible to set the kind of algorithm used with the environment variable `NCCL_ALGO`. However, not all the collectives support all the algorithms.

[4]There can be multiple kernels related to the same routine. For example, the all-reduce operation can be performed using a ring or a tree algorithm: NCCL selects the best one according to the topology and hardware properties detected during the initialization phase. To check which kernel is actually launched during the code execution one can use the NVIDIA NSights profiler, since NCCL kernels are instrumented with NVTX markers.



(a) (b)

Figure 2.1: Schematic representation of the inter-node connections for the ring (panel (*a*)) and tree (panel (*b*)) topology for 4 nodes of 8 GPUs each. The NICs of each node are "fused together" in the dark grey rectangles for clarity. Figure taken from [11].

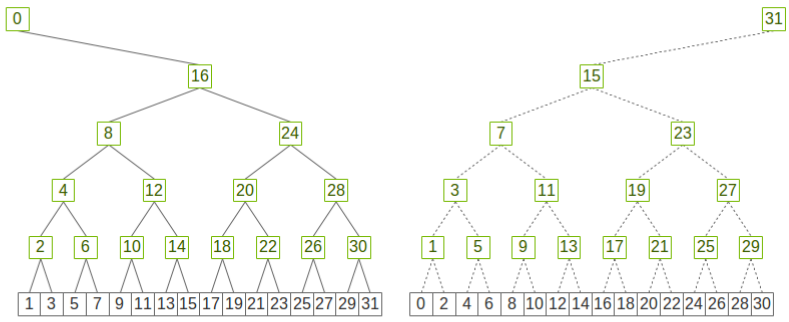


Figure 2.2: Schematic representation of the tree topology depicted in Fig. 2.1 (*b*): each square corresponds to a node and as we can see there are two incoming fluxes and two outgoing fluxes per node, neglecting the first and last node. (Which corresponds to two incoming and two outgoing fluxes per NIC).

paths determined during the initialization phase.
The kernel launched on a particular GPU will receive the data from the the previous GPU on the ring or tree, perform any necessary operations on the data (such as partial reductions or a copy to a local buffer) and then send the data to the next GPU.
The NCCL kernels handle both communications and computations required by the collectives, allowing for a high level of coordination between ranks. This enables efficient pipelining of the necessary operations and therefore to maintain a constant flow of data and maximize throughput.

2.2.1 Point-to-point directives

Since version 2.7, NCCL supports point-to-point communication which requires for the source rank to call `ncclSend` and the destination rank to call `ncclRecv`.
These routines are blocking: at variance with MPI, NCCL does not provide a direct counterpart of the non-blocking `Isend` and `Irecv` calls. On the other hand, NCCL includes a mechanism to coalesce multiple `ncclSend` and `ncclRecv` calls, in a non-blocking way, by encapsulating them within a `ncclGroup` environment using the `ncclGroupStart` and `ncclGroupEnd` routines. This structure is also useful to generate communication patterns which are not directly implemented in NCCL. For example, the all-to-all communication pattern can be implemented as:

```
1  ncclGroupStart();
2  for (int k =0; k<nprocs; k++){
3       ncclSend(&send[dim*k], dim, ncclInt, k, comm, stream);
4       ncclRecv(&recv[dim*k], dim, ncclInt, k, comm, stream);
5  }
6  ncclGroupEnd();
7  cudaStreamSynchronize(stream);
```

where `nprocs` is the number of ranks and `dim` is the number of elements in each chunk sent by one rank to the others. In the snippet above, NCCL takes care that all the `ncclSend` and `ncclRecv` calls inside the group are executed concurrently in a non-blocking way.

2.3 Benchmark details and results

This section presents the results of the benchmarks for various communication patterns and compares the throughput obtained with NCCL and MPI.
Several benchmarks codes were developed to test different communication patterns. Each code runs at first `nwarm` warm-up iterations of the communication and then `niters` timed iterations.
The following communication patterns were tested: bidirectional point-to-point exchange, all-reduce and all-to-all. The HPC-X and NCCL performances are compared through their "throughput" TP, defined as the ratio between the buffer size and the time per iteration:

$$TP = \frac{\text{buffer size}}{\text{total communication time /niters}}. \tag{2.1}$$

For the bidirectional point-to-point communication, the buffer size has to be doubled, since two buffers are simultaneously exchanged.

2.3.1 Benchmark on Leonardo cluster

The tests were performed on the pre-exascale LEONARDO cluster, located at CINECA and funded by the EUROHPC JU. LEONARDO is in the top 10 of the top500 ranking of high-performance-computing supercomputers in the world [14]. It possesses both a data centric CPU-only partition and a partition equipped with GPUs (the Booster partition), which is the one used for the benchmarks. The technical specifications of the Booster partition and the schematics of the intra-node and inter-node communications are summarized in Table 2.1 and Fig. 2.3, respectively, see [15] for more details. The reported intra-node NVLink bandwidth is bidirectional. The software stack used for the benchmarks included the NVIDIA SDK 24.3 and the latest HPC-X, based on CUDA 12, and the NCCL version 2.18.5 contained therein.

Bidirectional point-to-point exchange

The first communication pattern tested is the bidirectional ping-pong, where two processes simultaneously exchange two buffers. The MPI implementation of this pattern is done using non blocking point-to-point directives (`MPI_Isend` and `MPI_Irecv`):

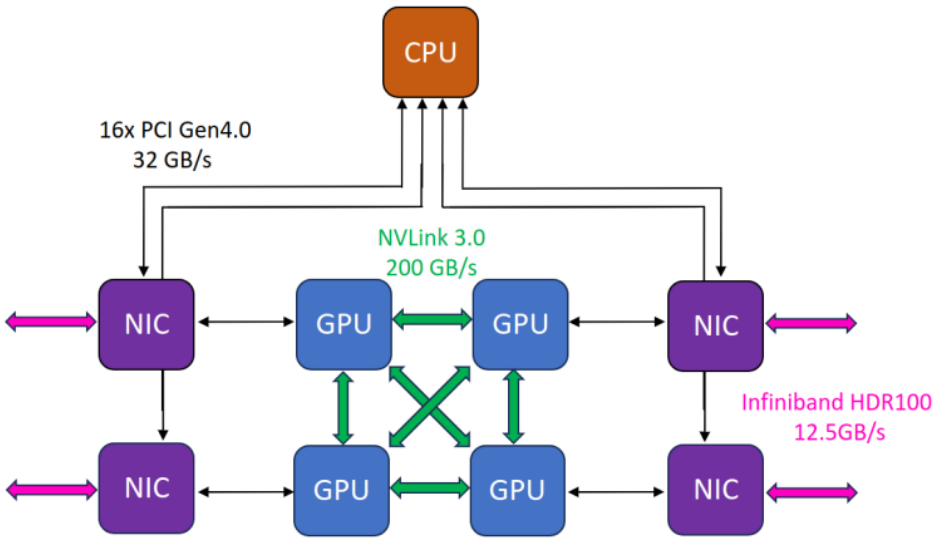


Figure 2.3: Schematics of inter/intra node communications bandwidths on LEONARDO Booster partition. Picture taken from [15]

.

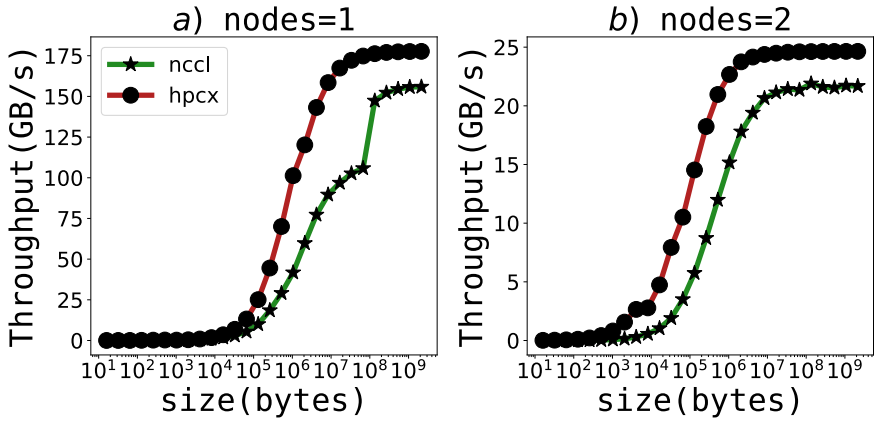


Figure 2.4: Throughputs for the bidirectional exchange between two ranks with HPC-X and NCCL, intra-node and inter-node on panel a) and b), respectively. For the intra-node case, the GPUs involved are 0 and 1, while on the inter-node case the GPUs communicating are the 0 on each node. It can be checked that the choice of GPUs in both cases does not lead to change in performances, consistently with the symmetry of the node depicted in Fig. 2.3.

HPC cluster	Leonardo
Centre	CINECA
Model	Atos Bull Sequana X2135 "Da Vinci" single-node GPU blade
Nodes	3456
Processors (per node)	1 x 32 cores Intel Xeon Platinum 8358 @2.60GHz
GPUs (per node)	4 x NVIDIA Ampere 64GB HBM2e NVLink 3.0 (200GB/s)
Cores	32 cores/node
RAM	512 (8x64) GB (node)
Node performance	peak about 309 Pflop/s
Network cards	(NVIDIA Mellanox Infiniband HDR) 200 Gbps
Network topology	DragonFly+

Table 2.1: System specifications of LEONARDO Booster partition

```
if(rank==0){
    MPI_Isend(send, dim, MPI_INT, 1, 0, MPI_COMM_WORLD,&req_0[0]);
}else if(rank==1){
    MPI_Irecv(recv, dim, MPI_INT, 0, 0, MPI_COMM_WORLD,&req_1[0]);
}
if(rank==1){
    MPI_Isend(send, dim, MPI_INT, 0, 1, MPI_COMM_WORLD, &req_1[1]);
} else if(rank==0){
    MPI_Irecv(recv, dim, MPI_INT, 1 , 1, MPI_COMM_WORLD,&req_0[1]);
}
if(rank==0){
    MPI_Waitall(2,req_0,MPI_STATUS_IGNORE);
}else{
    MPI_Waitall(2,req_1,MPI_STATUS_IGNORE);
}
```

while the NCCL one is done via `nccl_send` and `nccl_recv` encapsulated within a `nccl_group`, as follows:

```
ncclGroupStart();
if(rank==0){
   ncclSend(send, dim, ncclInt, 1, comm, s);
}else if(rank==1){
   ncclRecv(recv, dim, ncclInt, 0, comm, s);
}
```

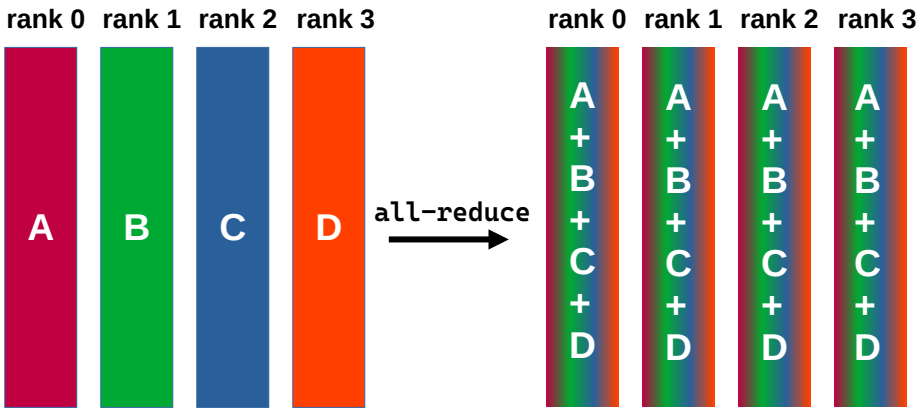


Figure 2.5: Schematization of the all-reduce communication pattern for the sum operation. Different colours correspond to data initially in different ranks.

```
if(rank==1){
   ncclSend(send, dim, ncclInt, 0, comm, s);
}else if(rank==0){
   ncclRecv(recv, dim, ncclInt, 1, comm, s);
}
ncclGroupEnd();
cudaStreamSynchronize(s);
```

The throughputs for HPC-X and NCCL are shown in Fig. 2.4. The intra-node communication, Fig. 2.4(a), is between GPU 0 and GPU 1 of the same node, while the inter-node one, Fig. 2.4(b), involves the GPUs 0 of two different nodes. The choice of GPUs in both cases does not lead to any change in performance, consistent with the symmetric architecture of the node depicted in Fig. 2.3. These results, at saturation, can be compared with the nominal bandwidths reported in Fig. 2.3, modulo halving the inter-node since the nominal values refer to the unidirectional bandwidth. HPC-X saturates 75% of the nominal intra-node band and fully saturates the inter-node band. For a point-to-point communication we don't expect to have any gain in performance due to NCCL, and indeed it actually performs worse than HPC-X, consistent with other benchmarks [7]

All-reduce

The second tested communication pattern is the all-reduce[16]. In brief, this pattern takes a buffer from each process, combines the corresponding entries across all ranks

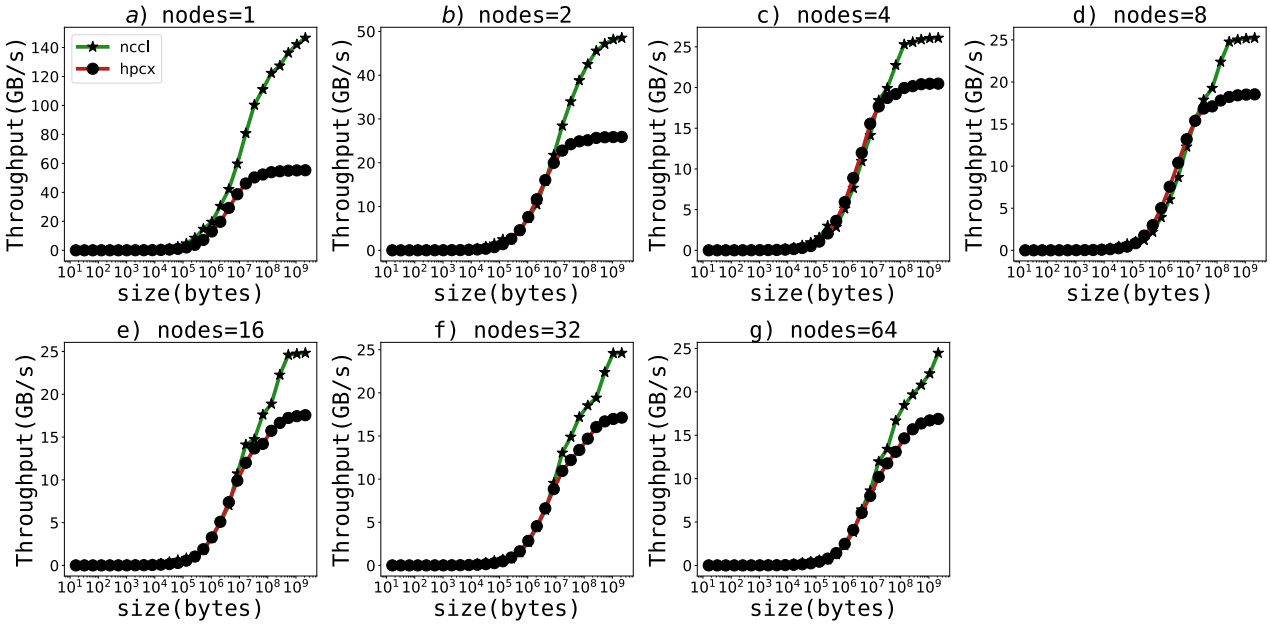


Figure 2.6: Throughputs for the all-reduce communication pattern with HPC-X (circles) and NCCL (stars), on 1 to 64 nodes on panel a) to g), respectively. The reduced buffer is fixed irrespectively of the number of nodes. For each number of nodes, all of the available GPUs are involved in the communication, i.e. 4 times the number of nodes.

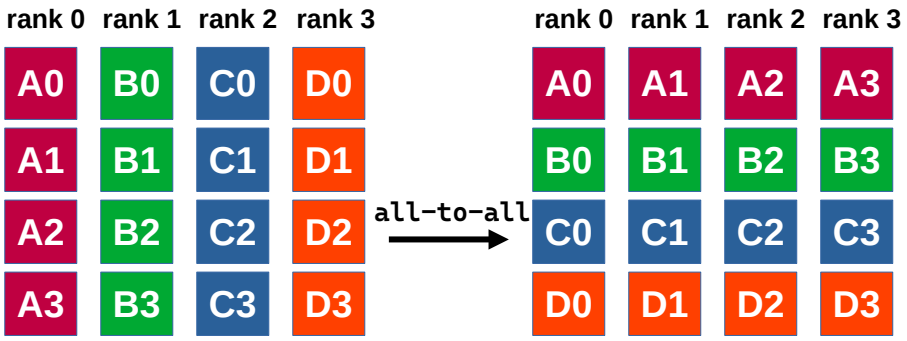


Figure 2.7: Schematization of the all-to-all communication pattern for the sum operation. Different colours correspond to data initially in different ranks.

using a specified operation (e.g. sum, max...), and then distributes the resulting buffer to all processes. This pattern is schematically represented in Fig. 2.5.
For the actual implementation, the built-in collective was used for both HPC-X and NCCL. The results for 1 to 64 nodes are shown in Fig. 2.6 in panels *a*) to *g*). NCCL outperforms HPC-X in all cases. On top of the hardware-specific optimization, the gain in performance is also due to the partial reductions performed by NCCL directly on the GPUs.
It is interesting to note that, in the case of two nodes, the NCCL throughput saturates approximately at 50GB/s, which is double the plateau obtained for higher numbers of nodes. By using the NVIDIA NSights profiler it was found that on 2 nodes NCCL implements the tree version of the all-reduce kernel, while on 4 nodes and above, the ring one is used. The fact that the tree version of the algorithm performs better than the ring one on two nodes stems from the fact that the former has less inter-node communications[5]. Indeed, as shown by the NIC traffic, it turns out that the tree algorithm performs roughly half of the inter-node communication of the ring one.

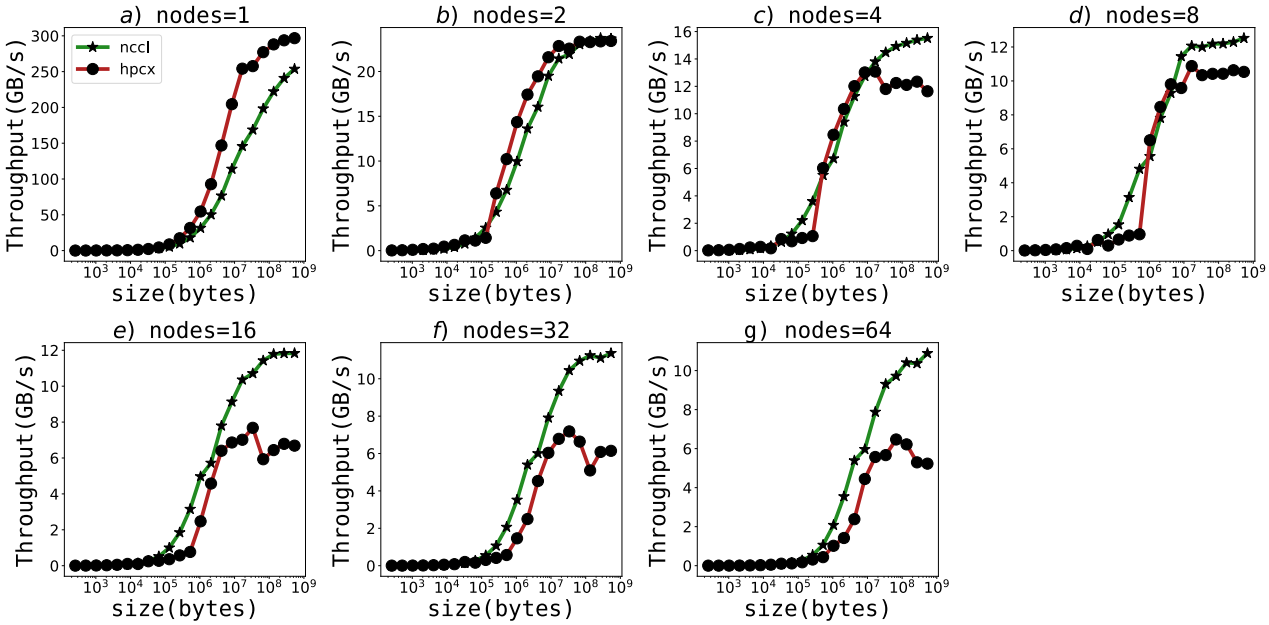


Figure 2.8: Throughputs for the all-to-all communication pattern with HPC-X (circles) and NCCL (stars), on 1 to 64 nodes on panel *a*) to *g*), respectively. For each number of nodes, all of the available GPUs are involved in the communication, i.e. 4 times the number of nodes.

All-to-all

The final tested communication pattern is the all-to-all[16]. In this case, each process starts with a buffer that is sliced in a number of chunks equal to the number of processes. These chunks are then distributed among the processes as illustrated in Fig. 2.7.
Since NCCL does not include a built-in all-to-all collective, the all-to-all algorithm suggested by NVIDIA [17] was employed:

```
int dim_b=dim_tot/nranks;
ncclGroupStart();
for (int k =0; k<nranks; k++){
    ncclSend(&send[dim_b*k], dim_b, ncclInt, k, comm, stream);
    ncclRecv(&recv[dim_b*k], dim_b, ncclInt, k, comm, stream);
}
ncclGroupEnd();
cudaStreamSynchronize(stream);
```

where `dim_tot` is the total dimension of the processed buffer and `nranks` is the number of ranks. While MPI includes a built-in collective for the all-to-all pattern, a custom implementation was written to mimic the one implemented in FFTXlib:

```
int dim_b=dim_tot/nprocs;
int kk=0;
for (int k =0; k<nprocs; k++){
   if(k!=rank){
      MPI_Irecv(&recv[dim_b*k], dim_b, MPI_INT, k, 0, MPI_COMM_WORLD,
                &reqs[kk]);
      MPI_Isend(&send[dim_b*k], dim_b, MPI_INT, k, 0, MPI_COMM_WORLD,
                &reqs[kk+nprocs-1]);
      kk+=1;
   }
}
MPI_Waitall(2*nprocs-2, reqs, MPI_STATUS_IGNORE);
```

The results for 1 to 64 nodes are reported in Fig. 2.8 in panels *a*) to *g*). For $4, 8$ nodes, NCCL begins to outperform HPC-X at a buffer size of approximately $10MB$: the ratio of the throughputs saturates at 1.5 for buffer sizes of 1GB. Increasing

[5]This is mentioned in one issue on the NCCL github page: `https://github.com/NVIDIA/nccl/issues/812`.

the number of nodes (from 16 onwards), NCCL performs significantly better than HPC-X after leaving the region dominated by the latency. More precisely, the ratio of the throughputs increases from $1.6 \div 1.7$ for buffer sizes between $1 \div 10$MB to approximately 2 for buffer sizes of 1GB.

Addendum

While this thesis was nearing completion, an opportunity arose to run a few jobs on Leonardo after the maintenance period, without any other user activity at the same time. This provided the chance to run tests on a larger number of nodes. The outcomes of the tests are shown in Table 2.2 and 2.3 for the all-reduce and all-to-all collective, respectively.
For the all-reduce pattern, the throughput values reported in Table 2.2 confirm the result discussed in Sec. 2.3.1: on large number of nodes, HPC-X and NCCL essentially perform the same until the saturation regime is reached. In the saturation regime, NCCL's throughput saturates at around $20 \div 25 GB/s$, while the HPC-X's one saturates at around $16 GB/s$.
For the all-to-all pattern, the result reported in Table 2.3 confirm the trends observed in Fig. 2.8. In particular, NCCL outperforms HPC-X not only around the saturation regime (as it was for the all-reduce, see Fig. 2.6), but also in the ramp-up regime. The ratio between the throughputs is also consistent with those discussed earlier.
As a further line of development, it would be interesting to extend the same analysis to other clusters, not necessarily equipped with NVIDIA GPUs, as other vendors have implemented their own version of a library optimized for collectives. For example, AMD and Intel provide the RCCL library [18] and the oneCCL [19] one, respectively.

	Size	$TP_{NCCL}\,(GB/s)$	$TP_{HPC-X}\,(GB/s)$
128 Nodes	$60MB$	12.0	12.5
	$120MB$	16.1	15.5
	$1GB$	21.7	16.5
	$2GB$	–	16.7
256 Nodes	$60MB$	12.6	12.0
	$1GB$	21.1	16.4

Table 2.2: Throughput values for the all-reduce collective on 128 and 256 nodes obtained through runs at empty machine.

	Size	$TP_{NCCL}\,(GB/s)$	$TP_{HPC-X}\,(GB/s)$
128 Nodes	$60MB$	7.6	3.4
	$120MB$	8.9	5.0
	$1GB$	11.6	4.0

Table 2.3: Throughput values for the all-to-all collective on 128 and nodes obtained through runs at empty machine.

Chapter 3

Density Functional Theory and Fourier transforms

QUANTUM ESPRESSO [1] "is an integrated suite of Open-Source computer codes for electronic-structure calculations and materials modelling at the nanoscale. It is based on density-functional theory, plane waves, and pseudopotentials.". The aim of this chapter is to provide a general overview of this technique, with an emphasis on the role played by Fourier transforms, as this is the focus of the thesis. For a more detailed treatment of these topics, specialized texts are available [20, 21].

3.1 Density functional theory and the Kohn-Sham equation

QUANTUM ESPRESSO deals with systems composed of periodically ordered sets of atoms[2]. The electrons of these atoms interact with the nuclei and with each other and exhibit an incredibly rich phenomenology.
The nuclei are treated as fixed under that assumption that their dynamics is much slower than that of the electrons. This leads to the following Hamiltonian:

$$\hat{H} = \frac{\hbar}{2m} \sum_i \nabla_i^2 + \sum_i V_{ext}(\mathbf{r}_i) + e^2 \sum_{i \neq j} \frac{1}{\|\mathbf{r}_i - \mathbf{r}_j\|}, \tag{3.1}$$

[1] `https://www.quantum-espresso.org/`

[2] This is not as big as a restriction as it may seem. Indeed, the code can also take into account non-periodic systems (for example, defects, surfaces, liquids etc..) by introducing an artificial periodicity via the so-called *supercell*.

where the first term encodes the kinetic energy of the electrons, the second one is the interaction with an external potential (which includes the interaction with the atomic nuclei) and the last one is the Coulombian interaction between the electrons. A solution in terms of the many-electrons wavefunctions is difficult and in any too large to be stored. Density functional theory (DFT) provides a foundation for treating this problem in terms of the electronic density.
The first part of the Chapter contains a presentation of the foundations of DFT, the Kohn-Sham method and the approximations introduced therein.
The second part of the Chapter describes how the approximate model introduced in the Kohn-Sham method is solved in the plane-wave and pseudopotential method.

3.1.1 The Hohenberg-Kohn theorems

DFT turns the many-electron problem (3.1) into a problem expressed in terms of the ground-state electron density. The correspondence between the wavefunctions and the ground-state density is established by the two Hohenberg and Kohn theorems.

- The first theorem establishes a bijective correspondence between the ground state electronic density $n_{GS}(\mathbf{r})$ and the external potential V_{ext}. Since $V_e xt$ fixes $\hat{H}$, all the many-body exact wavefunctions are fixed by n_{GS}. Therefore, all of the system properties are fixed by the ground-state electron density.

- The second theorem states that it is possible to define a universal functional for the energy of the system in terms of the density. For a given external potential V_{ext}, the minimum of this functional corresponds to the ground-state energy and the corresponding density is n_{GS}. In mathematical terms:

$$E_{HK}[n] = F_{HK}[n] + \int d^3\mathbf{r}\, V_{ext(\mathbf{r})} n(\mathbf{r}), \tag{3.2}$$

$$\left.\frac{\delta E_{HK}[n]}{\delta n(\mathbf{r})}\right|_{n=n_{GS}} = 0, \tag{3.3}$$

  where $F_{HK}[n]$ is the universal Hohenberg-Kohn functional, which includes the contribution of the kinetic energy and the many-body interaction energy of the electrons. This makes $F_{HK}[n]$ essentially impossible to compute exactly.

3.1.2 The Kohn-Sham approach

The Kohn-Sham approach introduces a fictitious system of non-interacting electrons subject to an external effective potential V_{KS} that depends on the electron density.

The key assumption is that it is possible to find a potential V_{KS} such that the ground state density of this fictitious system matches the ground-state density of the real, interacting, system.
In this non-interacting electron system the total energy functional can be written as:

$$E_{KS}[n] = T_{free}[n] + E_h[n] + \int d^3\mathbf{r}\, V_{ext(\mathbf{r})} n(\mathbf{r}) + E_{xc}[n] \tag{3.4}$$

where

- $E_H[n]$ is the Hartree density-density Coulomb interaction:

$$E_H = \frac{e^2}{2} \int d^3\mathbf{r}\, d^3\mathbf{r'} \frac{n(\mathbf{r})n(\mathbf{r'})}{\|\mathbf{r} - \mathbf{r'}\|}. \tag{3.5}$$

- $E_{xc}[n]$ is the exchange-correlation term, which accounts for the many-body effects and is implicitly defined by Eq. (3.4).

The ground-state density of the fictitious system minimizing the energy (3.4) is assumed to be the same of the interacting system described by (3.2).
At a first glance, this procedure does not appear to simplify the problem, since the many-body interaction remains present. However, compared with the Hohenberg-Kohn method, this approach separates the contribution of both the kinetic and long-range Coulomb energies from the functional F_{HK}. This separation allows E_{xc} to be written as a local functional based on reasonable physical assumptions. An introductory review of the various functionals present in the literature can be found in [20].
The expression of the energy (3.4) corresponds to the energy of a gas of non-interacting electrons moving in an external potential $V_{KS}(\mathbf{r})$:

$$E_{KS}[n] = T_{free}[n] + \int d^3\mathbf{r}\, V_{KS}(\mathbf{r}, [n]) n(\mathbf{r}), \tag{3.6}$$

where the density-dependent Kohn-Sham potential is defined as:

$$V_{KS}(\mathbf{r}, [n]) = V_H(\mathbf{r}) + V_{ext}(\mathbf{r}) + v_{xc}(\mathbf{r}, \tag{3.7}$$

where $V_H(\mathbf{r})$ is the Hartree potential:

$$V_H(\mathbf{r}) = \frac{\delta E_H[n]}{\delta n(\mathbf{r})} = \frac{e^2}{2} \int d^3\mathbf{r'} \frac{n(\mathbf{r'})}{\|\mathbf{r} - \mathbf{r'}\|}, \tag{3.8}$$

and $V_{xc}(\mathbf{r})$ is the exchange-correlation potential:

$$V_{xc}(\mathbf{r}) = \frac{\delta E_{xc}[n]}{\delta n(\mathbf{r})}. \tag{3.9}$$

Since the fictitious system consists of non-interacting electrons, the density and the kinetic energy can be written in terms of single-particle wavefunctions ψ_i, called Kohn-Sham orbitals:

$$T_{free} = -\frac{\hbar^2}{2m}\int d^3\mathbf{r}\,\psi_i^*(\mathbf{r})\nabla^2\psi_i(\mathbf{r}), \qquad n(\mathbf{r}) = \sum_i \|\psi_i(\mathbf{r})\|^2. \tag{3.10}$$

Minimizing the energy (3.6) with respect to ψ_i^* leads to the *Kohn-Sham equation* for ψ_i:

$$H_{KS}\psi_i(\mathbf{r}) = \left(-\frac{\hbar^2}{2m}\nabla^2 + V_{KS}(\mathbf{r},[n])\right)\,\psi_i(\mathbf{r}) = \varepsilon_i\;\psi_i(\mathbf{r}), \tag{3.11}$$

The operator H_{KS} is sometimes called *Kohn-Sham Hamiltonian*
Since the Kohn-Sham potential (3.7) depends on the electron density, which in turn depends on the wavefunctions via 3.10, equation (3.11) must be generally be solved iteratively (in the following, the explicit dependence of V_{KS} on the electronic density n will be dropped for simplicity):

1) Start with a initial guess for the density $n(\mathbf{r})$

2) Compute $V_{KS}(\mathbf{r})$

3) Solve KS equation (3.11) by diagonalizing the Hamiltonian H_{KS}, obtaining $\{\psi_i(\mathbf{r})\}_i,\ \varepsilon_i$

4) Compute the electron density using Eq. (3.10)

5) Repeat until the density converges

3.2 Plane-waves based Kohn-Sham approach

Once the density is obtained, it can be used to compute the desired material properties. However, the solution of Eq. (3.11) for a given density remains computationally challenging. Therefore, a practical strategy is needed to solve the equation efficiently. QUANTUM ESPRESSO adopts the *Plane-Wave plus*

pseudopotential (PW+PP) method which consists of projecting Eq. (3.11) on the basis of plane waves defined on the lattice [21]. The presence of a crystalline structure introduces specific characteristics in the Fourier transform analysis which are crucial to understand the method and its implementation. For this reason Appendix A provides a brief introduction to lattices and their Fourier properties.

3.2.1 Kohn-Sham equation in Fourier space

Consider the Kohn-Sham equation (3.11) and impose periodic boundary conditions on the wavefunctions ψ_i over a large volume V^3. This allows the Fourier expansion of $\psi_i(\mathbf{r})$ as:

$$\psi_i(\mathbf{r}) = \sum_{\mathbf{q}} c_{i,\mathbf{q}} \frac{e^{i\mathbf{q}\cdot\mathbf{r}}}{\sqrt{V}}. \tag{3.12}$$

Taking the Fourier transform of (3.11) with respect to momentum $\mathbf{q}'$ and substituting in 3.12 leads to:

$$\int d^3r\, e^{-i\mathbf{q}'\cdot\mathbf{r}} \left(-\frac{\hbar}{2m}\nabla^2 + V_{KS}(\mathbf{r}) - \varepsilon_i \right) \sum_{\mathbf{q}} c_{i,\mathbf{q}} \frac{e^{i\mathbf{q}\cdot\mathbf{r}}}{\sqrt{V}} = 0. \tag{3.13}$$

The kinetic term immediately yields the expression:

$$\int d^3r\, e^{-i\mathbf{q}'\cdot\mathbf{r}'} \frac{\hbar^2}{2m}\nabla^2 e^{i\mathbf{q}\cdot\mathbf{r}} = \frac{\hbar^2}{2m}\mathbf{q}^2 \delta_{\mathbf{q},\mathbf{q}'}. \tag{3.14}$$

The potential term is more complex. Since the potential describes a crystalline solid, it is assumed to be periodic. Consequently, according to the analysis presented in Appendix A its Fourier expansion contains only the modes belonging to the reciprocal lattice:

$$V_{KS}(\mathbf{r}) = \sum_{\mathbf{G}} \tilde{V}_{KS}(\mathbf{G}) e^{i\mathbf{G}\cdot\mathbf{r}}, \quad \tilde{V}_{KS}(\mathbf{G}) = \frac{1}{v_c}\int_{v_c} d^3\mathbf{r} V_{KS}(\mathbf{r}) e^{-i\mathbf{G}\cdot\mathbf{r}}. \tag{3.15}$$

The matrix elements of V_{KS} in momentum space then reduces to:

$$\int d^3r\, e^{-i\mathbf{q}'\cdot\mathbf{r}} V_{KS}(\mathbf{r}) e^{i\mathbf{q}\cdot\mathbf{r}} = \sum_{\mathbf{G}} \tilde{V}_{KS}(\mathbf{G})\, \delta_{\mathbf{q}'-\mathbf{q},\mathbf{G}}. \tag{3.16}$$

[3] It is also possible to impose periodic boundary condition on a volume commensurate to the primitive cell (called Born-Von Karman conditions). This choice reproduces the same results as the full periodic boundary conditions in the limit of large volume.

It is convenient to redefine $\mathbf{q}$ and $\mathbf{q}^{'}$ as:

$$\mathbf{q} = \mathbf{k} + \mathbf{G}, \quad \mathbf{q}^{'} = \mathbf{k} + \mathbf{G}^{'}, \tag{3.17}$$

where $\mathbf{G}$ and $\mathbf{G}^{'}$ are vectors in the reciprocal lattice (and thus also their difference belongs to the reciprocal lattice) and $\mathbf{k}$ belongs to the Brillouin zone (BZ).
In conclusion, the Kohn-Sham differential equation (3.11) is cast as an algebraic equation for the Fourier coefficients of the wavefunction:

$$\sum_{\mathbf{G}^{'}} \left(\frac{\hbar^2}{2m} (\mathbf{k} + \mathbf{G}^{'})^2 \delta_{\mathbf{G},\mathbf{G}'} + \tilde{V}_{KS}(\mathbf{G} - \mathbf{G}^{'}) \right) c_{i,\mathbf{G}'}(\mathbf{k}) = \varepsilon_i(\mathbf{k}) c_{i,\mathbf{G}}(\mathbf{k}), \tag{3.18}$$

where $c_{i,\mathbf{G}}(\mathbf{k}) = c_{i,\mathbf{q}=\mathbf{k}+\mathbf{G}}$ and the dependency of ε_i on $\mathbf{k}$ has been explicitly highlighted.

Core electron and pseudopotentials

It is important to note that plane waves, in principle, are not a good basis for solving the Kohn-Sham equation. The reason is that, when describing a crystal, two qualitatively different kinds of electrons must be considered: the *core electrons* and the *valence electrons*. Core electrons are localized near the nuclei, are essentially inert and don't contribute to the chemical properties of the material. Valence electrons occupy higher-energy orbitals and are delocalized between the nuclei: these are the ones that determine the material's properties of interest.
The core wavefunction are non-zero only close to the nuclei, where they resemble the atomic orbitals and tend to oscillate rapidly. The valence electrons, on the other hand, can be considered nearly free between the nuclei. However, near the nuclei they are subject to the same potential as the core electrons, causing their wavefunction to oscillate too. Since the energy of the valence electrons is higher than that of the core electrons, their wavefunctions are expected to oscillate even more rapidly near the nuclei. In both cases to resolve the wavefunctions an high spatial resolution is required. This, in turn, entails the need to consider an high number of plane waves, making the method computationally and memory expensive and potentially reducing its practical usefulness.
The pseupotential (PP) method alleviates this problem by replacing the Coulomb-like potential of the nuclei and the contribution of the core electrons with a smooth effective "pseudopotential". This pseudopotential reproduces the relevant features of the real one away from the nucleus, preserving the correct interaction with the

valence electron. Several pesupotential have been proposed in the literature[20], and the details of the method go beyond the scope of this thesis. The key point is that the pseudopotential method introduces a non-local term in the Kohn-Sham Hamiltonian. This non-local term will be treated in section 3.2.3.

3.2.2 The cut-off and its consequences

The procedure leading to Eq. (3.18) is exact if all of the $\mathbf{G}$ vectors are considered, which are infinite. For numerical implementations, however, a cut-off on the number of harmonics must be imposed. This is done by setting a cut-off on the energy of the modes, which defines a sphere in reciprocal space:

$$\frac{\hbar^2}{2m}(\mathbf{k}+\mathbf{G})^2 < E_{cut} \equiv \frac{\hbar^2}{2m}G_{max}^2. \tag{3.19}$$

In the code, E_{cut} is given as an input parameter (as are the primitive vectors of the direct lattice that we want to model).
However, the introduction of a cut-off has additional consequences.
Consider the Kohn-Sham equation in Fourier space (3.18). The product of local functions in real space becomes a convolution in Fourier space:

$$V_{KS}(\mathbf{r})\psi_i(\mathbf{r}) \rightarrow \sum_{\mathbf{G}'} \tilde{V}_{KS}(\mathbf{G}-\mathbf{G}')c_{i,\mathbf{G}'}(\mathbf{k}). \tag{3.20}$$

From the constraint in (3.19) it follows that $\|\mathbf{G}-\mathbf{G}'\| < 2G_{max}$. Therefore a Fourier space grid twice as large is required compute $V_{KS}(\mathbf{r})\psi(\mathbf{r})$ in real space.
A similar necessity emerges for the electronic density, defined in Eq. (3.10), since it is given by a sum of products of functions in real space. Taking the Fourier transform of (3.10) yields:

$$n(\mathbf{G}) = \sum_{\mathbf{G}'}^{\|\mathbf{G}'\|<G_{max}} \sum_{i,\mathbf{k}} c^*_{i,\mathbf{G}'}(\mathbf{k})c_{i,\mathbf{G}-\mathbf{G}'}(\mathbf{k}), \tag{3.21}$$

which contains modes up to $2G_{max}$. The constraint on the wavefunction mode $\|\mathbf{G}-\mathbf{G}'\| < G_{max}$ implies that $\|G\| < 2G_{max}$. The conclusion is once again that Fourier transforms must be performed on a grid with cut-off $2G_{max}$.
For each value of $\mathbf{k}$ (called k-point) the number of reciprocal vectors (i.e. plane waves) to consider is equal to the number of reciprocal vectors inside the sphere of

radius $2G_{max}$. Since each point of the reciprocal lattice is associated to a volume v_{BZ} the total number of plane waves considered is:

$$N_{PW} = \frac{(4/3)\pi(2G_{max})^3}{v_{BZ}}. \tag{3.22}$$

Aliasing

Doubling the cut-off in reciprocal space also solves another issue, called *aliasing*. To illustrate it, consider a one-dimensional case where the direct and reciprocal lattices are one-dimensional periodic grids. The spatial resolution L/N_{max} is fixed by the cut-off on reciprocal vectors G_{max}:

$$q_n = \frac{2\pi}{L}n, \quad n = -N_{max}/2, , ..., N_{max}/2 - 1, \quad N_{max} = \frac{2G_{max}L}{2\pi} \tag{3.23}$$

Now consider the product $V\psi$ in Eq. 3.20. The Fourier transform of the potential, $\tilde{V}$, has harmonics up to $2G_{max}$, while c_i (the Fourier transform of the wavefunction) has harmonics up to G_{max}. Therefore their product contains harmonics up to $3G_{max}$.
In the reciprocal grid, all the vectors larger than $2G_{max}$ are folded back: a vector $3G_{max}$ would be folded back to $-N_{max}/2 + G_{max}$. This is important because these spurious vectors will never reach the grid region where the wavefunction modes are defined, namely $[-G_{max}, G_{max}]$. A similar mechanism applies in higher dimensions.

3.2.3 Solution of the Kohn-Sham equation

In the iterative scheme presented in section 3.1, the Kohn-Sham equation is linear in the wavefunction, since the density is considered fixed at each step. Therefore, the problem reduces to the diagonalization of a linear operator H_{KS}, which is schematically given by:

$$H_{KS} = \nabla^2 + V_{nonloc} + V_{loc}, \tag{3.24}$$

where V_{nonloc} contains the non local contribution of the pseudopotential, and V_{loc} contains the other local contributions to the potential. Some of this terms are more convenient to compute in Fourier space while others are more convenient to compute in real space. For example, the terms of the potential which were local in real space now become convolutions. Schematically:

$$V_{KS}(\mathbf{r})\psi_i(\mathbf{r}) \rightarrow \sum_{\mathbf{G}'} \tilde{V}_{KS}(\mathbf{G} - \mathbf{G}')c_{i,\mathbf{G}'}(\mathbf{k}) \sim \sum_{\mathbf{G}'}\sum_{\mathbf{r}} V_{KS}(\mathbf{r})e^{i\mathbf{r}\cdot(\mathbf{G}-\mathbf{G}'}c_{i,\mathbf{G}'}(\mathbf{k}), \tag{3.25}$$

which implies that the computation of the corresponding terms scales quadratically with the number of grid points[4] N. Since there is a very efficient algorithm to perform the Fourier transform, namely the Fast Fourier Transform, it is more convenient to compute the product in real space, followed by a Fourier transform of the result. This procedure scales as $Nlog(N)$ [22]. The same reasoning applies for the other terms in 3.11. On the other hand, some terms, such has the Laplacian term or the non-local pseudopotential, are easier to compute in Fourier space. Therefore, the computation is performed in the space where it is more efficient and an FFT is applied to move between spaces as required.
The same technique is used also to compute the electronic density and to perform other operations in the code that we have not detailed here.
The possibility of efficiently switching between real and Fourier space efficiently eliminates the need to store the full H_{KS} operator in memory as a matrix. Instead, its action on vectors can be computed by summing the various contributions computed as described above.

[4]The domain decomposition of both real and Fourier space will be described in detail in Chapter 4.

Chapter 4

FFTXlib library and benchmarks

As shown in the previous chapter, (Fast) Fourier transforms play a crucial role in QUANTUM ESPRESSO . In this chapter a more precise description of their implementation in the code is provided. The discussion begins with an explanation of how the data is stored (i.e. the grids in real and reciprocal space) and then details how the FFTs are actually performed. Finally, the implementation of NCCL inside FFTXlib is presented, followed by a discussion of the benchmark results. In this chapter units where $\hbar = m = 1$ are adopted.

4.1 Real space and reciprocal space grids

The input of QUANTUM ESPRESSO contains both the primitive vectors of the lattice to be studied, $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3$, and the value of the energy cut-off E_{cut}. Given the primitive vectors, the code computes the corresponding primitive vectors of the reciprocal lattice $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$, and defines the following reciprocal grid:

$$\mathbf{G} = n_1\mathbf{b}_1 + n_2\mathbf{b}_2 + n_3\mathbf{b}_3, \quad n_i = 0, ..N_i - 1, \ i = 1, 2, 3, \tag{4.1}$$

where N_i are large enough to accommodate all Fourier modes within the cut-off and are defined as (here [ ] is the ceiling function) :

$$N_i = \left[\sqrt{\frac{4G^2_{max}}{\|\mathbf{b}_i\|^2}} \right]. \tag{4.2}$$

The corresponding grid in real space, which samples the primitive cell, is defined as:

$$\mathbf{r} = m_1\frac{\mathbf{a}_1}{N_1} + m_2\frac{\mathbf{a}_2}{N_2} + m_3\frac{\mathbf{a}_3}{N_3} \quad m_i = 0, ..N_i - 1, \ i = 1, 2, 3. \tag{4.3}$$

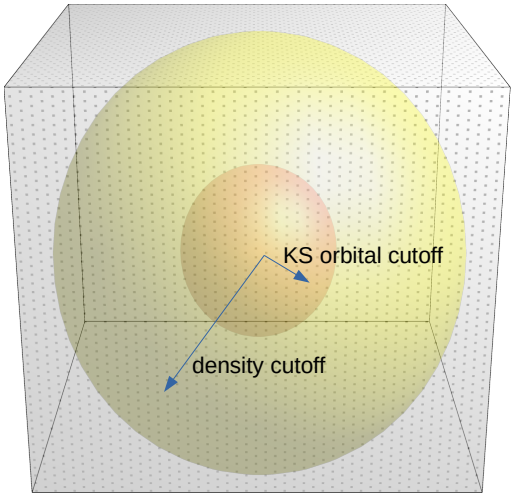


Figure 4.1: Cubic grid in Fourier space. The KS wavefunctions and the electronic density are set to zero in all points outside of the smaller, orange, sphere and the bigger, yellow sphere, respectively. Figure taken from [PAPAER]

Both the grids have the same number of points $N_{FFT} = N_1 N_2 N_3$, as required for the FFT algorithm to work.
For simplicity, the rest of the thesis will focus on the case of cubic grids, where $N_1 = N_2 = N_3 \equiv N$. The situation in Fourier space is depicted in Fig. 4.1. The points (n_1, n_2, n_3) form a cubic grid of edge N (in grey), which is circumscribed to the sphere of the density cut-off. The density is set to zero for points outside of this sphere. On the other hand, the KS orbitals are set to zero for all points outside a smaller sphere (in orange) of radius 1/4 smaller than that of the density sphere.[1]

The Fourier expansion for a function with the direct lattice periodicity (where only the reciprocal lattice modes contribute) is given by:

$$f(\mathbf{r}) = \sum_{\mathbf{G}} f_{\mathbf{G}} e^{i\mathbf{G}\cdot\mathbf{r}}, \quad \tilde{f}_{\mathbf{G}} = \frac{1}{v_c} \int_{v_c} d^3r \, f(\mathbf{r}) e^{-i\mathbf{G}\cdot\mathbf{r}}, \tag{4.4}$$

where the integral is over the primitive cell. Their discretized counterpart are given

[1]The padding of the wavefunctions and the density with zeroes occurs only in the region of the code where the FFT is performed. In general, only the non-zero components (i.e. those inside the spheres) are stored in memory. Furthermore, only the wavefunctions in reciprocal space are stored in memory, the ones in real space are computed and used as needed.

by three independent one-dimensional (anti) Fourier transforms along each direction of the grids:

$$f(\mathbf{r}) \rightarrow f(n_1, n_2, n_3) = \sum_{m_1,m_2,m_3} \tilde{f}(m_1, m_2, m_3) e^{2\pi i n_1 m_1/N_1} e^{2\pi i n_2 m_2/N_2} e^{2\pi i n_3 m_3/N_3} \quad (4.5)$$

$$\tilde{f}_{\mathbf{G}} \rightarrow \tilde{f}(m_1, m_2, m_3) = \sum_{n_1,n_2,n_3} \frac{\tilde{f}(n_1, n_2, n_3)}{N_{FFT}} e^{-2\pi i n_1 m_1/N_1} e^{-2\pi i n_2 m_2/N_2} e^{-2\pi i n_3 m_3/N_3}. \quad (4.6)$$

4.2 FFT implementation in FFTXlib

The implementation of the FFTs in QUANTUM ESPRESSO must satisfy certain requirements dictated by typical workloads. To begin with, the grid size seldom exceeds 1000^3 points. Therefore, the FFT implementation should target high efficiency on medium to small grids. This could lead to a low throughput, in particular when the grids are distributed among MPI ranks when the code is executed on several GPUs. However another level of parallelism can be unlocked by noting that the KS orbitals typically number in the range of 10^2 to 10^3. This makes it possible to increase the throughput by processing several orbitals concurrently. This mechanism is called *batching* and will explained in detail later. Finally, the implementation must account for the presence of a variable cut-off on Fourier modes, which differs between the KS orbitals and the density.

4.2.1 Domain decomposition

The direct and reciprocal grids are distributed among ranks when the code is executed on multiple GPUs, with each MPI task handling one GPU. The direct space grid is distributed using a standard slab decomposition by default[2]. In reciprocal space, however, only the Fourier components of the KS orbitals inside the appropriate sphere are retained. To achieve a balanced workload distribution, a stick decomposition is adopted. In particular, the sphere is divided in sticks which are distributed between the ranks in such a way as to balance the workload. Each process is assigned a mix of "long sticks" (from the region close to the z axis) and "short sticks", (from the equatorial region).

[2]A pencil decomposition is also available, but typically there is no need to use it as the slabs fit into the memory of the GPUs.

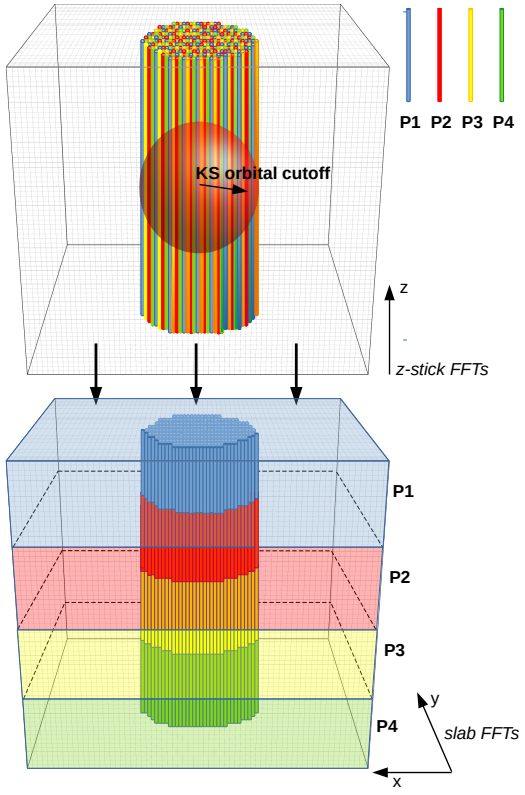


Figure 4.2: Schematics of an inverse FFT performed by FFTXlib. Different colours refer to buffers belonging to different ranks. On top, the stick decomposition, used to perform the FFT along the z direction. On the bottom, the slab decomposition used to perform the $2d$ FFT along the xy plane.

4.2.2 Inverse FFT procedure

For conciseness, only the inverse FFT is explained here; the forward FFT follows analogous principles. Schematically, a $3d$ inverse FFT consists in three separate inverse FFTs along the three directions in momentum space.
When an inverse FFT is required, the sticks are extended across the reciprocal grid by padding them with zeroes to form a cylinder of height N and radius equal to the one of the cut-off sphere. At this point an inverse FFT is performed along the z axis in momentum space, which fills the padding with meaningful values. The result is a fully populated cylinder, as depicted in the top panel of Fig. 4.2.
To perform the inverse FFT on the xy planes, the data must be distributed among the processes using a slab decomposition. Each rank must have the whole xy domain for a contiguous set of values of z, as shown in the lower panel of Fig. 4.2. This distribution is achieved via an all-to-all communication pattern[3]. At the end of the communication step each process holds the full xy domain for a certain range of z values. At this point an inverse $2d$ FFT is performed along the xy plane, thereby completing the calculation of the full $3d$ inverse FFT.

4.2.3 The batching mechanism

As noted at the beginning of this chapter, the grids processed in QUANTUM ESPRESSO are typically of medium size. To increase the throughput the high number of KS orbitals to be transformed must leveraged, noting that they can be treated independently form each other. This idea was already exploited when QUANTUM ESPRESSO was first ported to GPUs [23].
More precisely, several wavefunctions can be processed concurrently in groups called *batches*. All the wavefunctions in the same batch are merged in a single buffer, which is then distributed among the processes. A single batch is further divided in subgroups of bands, called *sub-batches*, each of which is transformed independently of the others. This approach unlocks a new level of parallelism. Indeed, using dedicated GPU streams, it is possible to overlap the computation, communication

[3]For the sake of completion, it is worth mentioning that some `cudaMemCpy` and `cudaMemCpy2D` calls are involved to properly prepare the data for the all-to-all pattern. In particular, the data need to be reshaped to send contiguous chunks at constant z. After the all-to-all step each process will hold various chunks corresponding to the values of (x, y) (in momentum space) associated to the original sticks. However these chunks are contiguous in memory, while the sticks likely had non-adjacent (x, y) coordinates. Therefore, these values must be properly inserted in the full xy grid in their correct positions. These operations will not be described in detail. Nevertheless, it is important to mention them because, as will be discussed below, they are overlapped with communication in the batched version of the FFT.

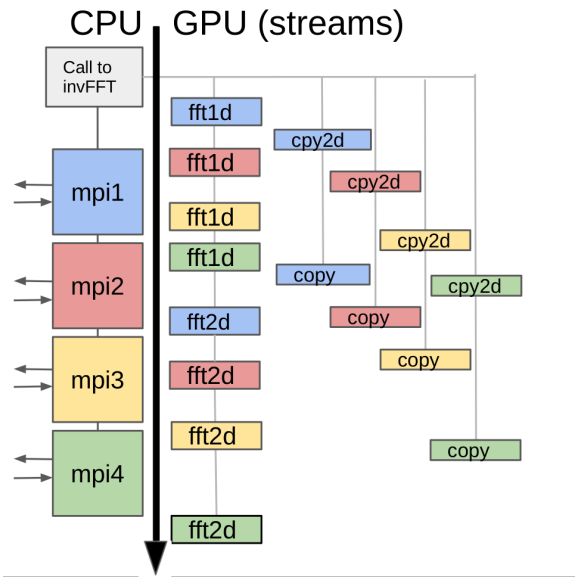


Figure 4.3: Schematics of an inverse batched FFT. Panel (a) represents the inverse FFT of a batch composed by four sub-batches: different colours corresponds to operations performed on different sub-batches. While the communication step related to one sub-batch is performed, the computations related to the other sub-batches can be executed asynchronously using GPU streams.

and memory copies steps of the various sub-batches, as depicted in Fig. 4.3. The details of the communication pattern are more complex, but it is remains essentially an all-to-all pattern. The implementation it is built using potentially GPU-aware MPI non-blocking calls `MPI_Isend`, `MPI_Irecv`.

4.2.4 NCCL implementation

NCCL was implemented in FFTXlib by replacing the all-to-all communication pattern with the corresponding NCCL counterpart. The original communication pattern in MPI is shown in the following code snippet:

```
1 DO iter = 2, nprocp
2    IF(IAND(nprocp, nprocp-1) == 0) THEN
3      sorc = IEOR( me-1, iter-1 )
4    ELSE
```

```
    sorc = MOD(me-1 - (iter-1) + nprocp, nprocp)
  ENDIF
  call MPI_IRECV( f_in_d((sorc)*sendsiz + 1), sendsiz, \\
  MPI_DOUBLE_COMPLEX, sorc, 0, gcomm, srh(iter-1), ierr )
ENDDO
DO iter = 2, nprocp
  IF(IAND(nprocp, nprocp-1) == 0) THEN
    dest = IEOR( me-1, iter-1 )
  ELSE
    dest = MOD(me-1 + (iter-1), nprocp)
  ENDIF
  call MPI_ISEND( f_aux_d((dest)*sendsiz + 1), sendsiz,\\
  MPI_DOUBLE_COMPLEX, dest, 0, gcomm, srh(iter+nprocp-2), ierr )
ENDDO
call MPI_WAITALL(2*nprocp-2, srh, MPI_STATUSES_IGNORE, ierr)
```

The input and output buffers, `f_in_d` and `f_aux_d`, are complex buffers residing on the device. Each rank computes the source and destination rank, determines the proper buffer slice to send and identifies the correct position in the buffer where the data should be received. The various parameters in the snippet are defined elsewhere and are not relevant for the current discussion.
The correspondent NCCL communication pattern is shown in the following snippet:

```
c_send = c_loc(f_aux_d)
c_recv = c_loc(f_in_d)
call c_f_pointer(c_send, r_send, [2*nxx_])
call c_f_pointer(c_recv, r_recv, [2*nxx_])
nccl_status = ncclGroupStart()
DO iter = 2, nprocp
  IF(IAND(nprocp, nprocp-1) == 0) THEN
    sorc = IEOR( me-1, iter-1 )
    dest = IEOR( me-1, iter-1 )
  ELSE
    sorc = MOD(me-1 - (iter-1) + nprocp, nprocp)
    dest = MOD(me-1 + (iter-1), nprocp)
  ENDIF
  nccl_status = ncclSend(r_send(2*((dest)*sendsiz+1)-1), 2*sendsiz,\\
  ncclDouble, dest, dfft%nccl_comm, dfft%nccl_streams(1)) !iter-1
```

```
  nccl_status = ncclRecv(r_recv(2*((sorc)*sendsiz+1)-1), 2*sendsiz,\\
  ncclDouble, sorc, dfft%nccl_comm, dfft%nccl_streams(1))
ENDDO
nccl_status = ncclGroupEnd()
```

The main modification is the replacement of the MPI calls with their NCCL counterparts, properly grouped using `ncclGroup` as explained in Chapter 1. A more subtle adjustment involves handling complex buffers, since NCCL does not directly support them. The solution adopted in this work is to reinterpret the complex buffers as real ones (`r_send` for `f_in_d` and `r_recv` for `f_aux_d`) using the `c_f_pointer` subroutine (`n_xx` is the number of complex entries of `f_in_d` and `f_aux_d`) :

```
c_send = c_loc(f_aux_d)
c_recv = c_loc(f_in_d)
call c_f_pointer(c_send, r_send, [2*nxx_])
call c_f_pointer(c_recv, r_recv, [2*nxx_])
```

These real pointers are then passed as input to the NCCL APIs, specifying that we are sending real double variables (`ncclDouble`). Finally, the buffer slices for both sending and receiving must be adjusted, since each complex entry corresponds to two real ones. For example, if the original the MPI call was:

```
MPI_ISend(complex_buffer(i), sendsiz,.....)
```

the corresponding NCCL call is:

```
ncclSend(real_buffer(2*i-1), 2*sendsiz,.....)
```

4.3 Results of the benchmarks on the Leonardo cluster

The benchmarks were performed on the Leonardo cluster, described in Chapter 1, using a mini-app that extracts the relevant FFTXlib routines from the full QUANTUM ESPRESSO code. The time-to-solution for an inverse FFT was compared on four grids of different sizes N_{FFT}, 128^3, 256^3, 512^3, 1024^3, using both HPC-X and NCCL, for several number of nodes.

The results of this benchmarks are reported in Fig. 4.4. The various bars represent the value of the time of a single inverse FFT, normalized over the batch size (i.e. over the total number of functions transformed).

4.3.1 HPC-X performance with default batching

First, consider the diagonally striped cyan bars, labelled "hpcx default": these correspond to the default batching parameters. KS orbitals are transformed in batches of 16 functions each, grouped in 4 sub-batches of 4 functions. As shown in the figure, scaling over the chosen node configurations is only achieved for the largest grids, which are at the high-end of the typical QUANTUM ESPRESSO use cases.
To understand this behaviour, consider the all-to-all benchmarks reported in Fig. 2.8. There, a threshold value can be identified between the latency/linear ramp regime and the bandwidth saturation regime, which is between $10MB$ and $100MB$, depending on the number of nodes. For the smaller grids the buffer corresponding to 4 sub-batches (i.e. the one that is processed by the all-to-all) is not large enough to reach the saturation regime for any of the nodes configurations examined.

4.3.2 HPC-X performance with custom batching

To achieve scaling, the solution is to increase the sub-batch size, thereby increasing the the communicated buffer and reaching the saturation regime. In principle, this mechanism allows for an unlimited increase of the communicated buffer, which in turn enables bandwidth saturation for any grid size. However, there are two practical limitations. First, the sub-batch size cannot exceed the batch size, which determines the memory footprint on each GPU. Second, increasing the sub-batch size increases the time needed for the `memcopy` operations after the communication step. In some cases, this time seem to become comparable to the communication time itself thereby causing the overlapping strategy presented in Fig. 4.3 to break down. Further investigation is needed to properly understand and address the issue.
Several combinations of the batch and sub-batch size were tested for the sample grids. Empirically, the best results were obtained by using four sub-batches, each of size equal to twice the number of nodes for the grids 128^3, 256^3, 512^3. For the larger grid of 1024^3 the sub-batch size was set equal to the number of nodes, mainly due to memory constraints. This choice of parameters allows for scaling across all the nodes configuration tested for all grid sizes. In Fig. 4.4 this optimized batching configuration is labelled "hpcx custom", and is depicted in criss-crossed orange bars. As shown, the code now scales consistently across all the node configurations.

4.3.3 NCCL performance

The NCCL implementation of the miniapp was executed using the same batching configuration as the "hpcx custom" case. The results are shown in the green horizontally striped bars in Fig. 4.4. Additionally, the "gain" of NCCL over HPC-X, defined as the ratio of the "hpcx custom" time over the NCCL one, is reported as the blue dashed line.
Consistent with the all-to-all benchmarks shown in Fig. 2.8, the gain of NCCL over HPC-X increases with both the size of the communicated buffer and the number of nodes. Interestingly, the best result is obtained for the smallest grid, with a gain of 2.3 on 32 nodes. However, the corresponding communicated buffer lies in the ramp region seen in Fig. 2.8, which may lead to instabilities in this case. The best gains for the other grids are: 1.4 for the grid of 256^3 on 16 nodes, 1.7 for the grid of 512^3 on 32 nodes and 1.9 for grid of 1024^3 on 128 nodes.

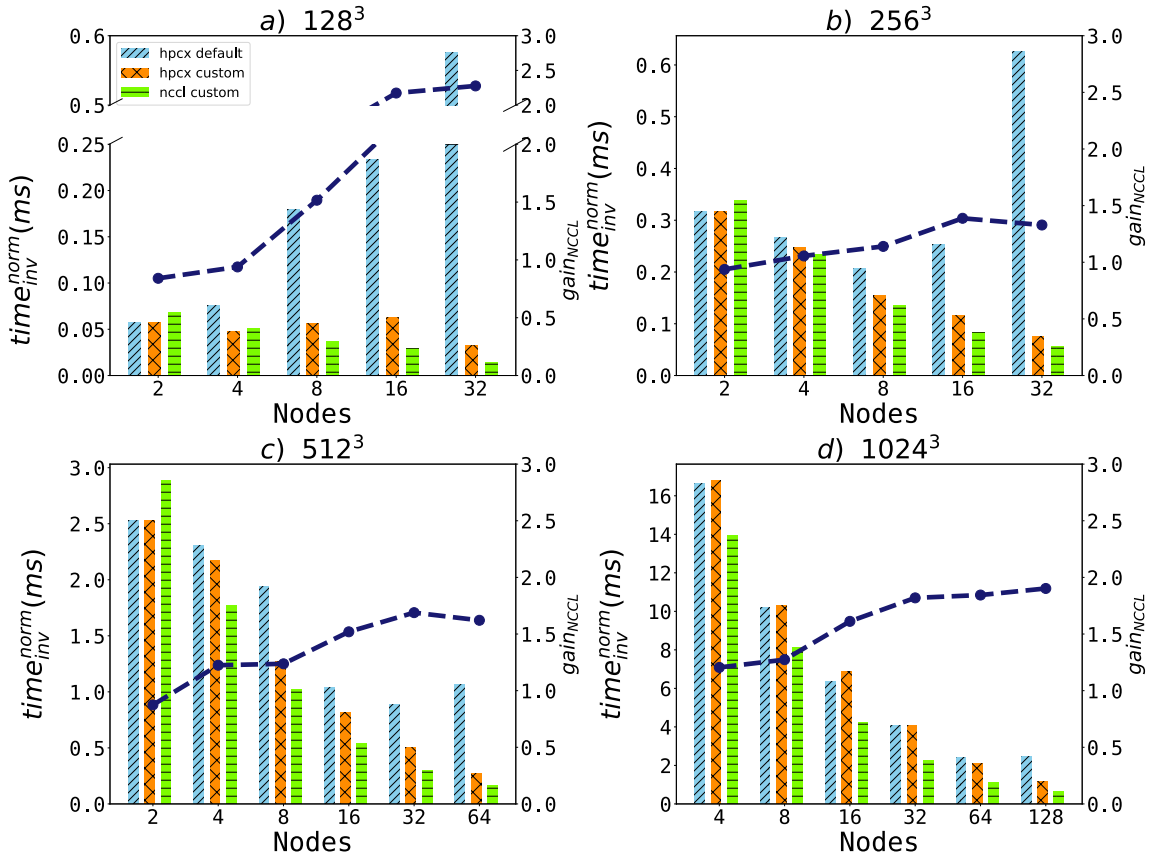


Figure 4.4: Comparison between inverse FFT time to solution, normalized over the batch size, for four different grid sizes: 128^3, 256^3, 512^3 and 1024^3 in panels *a*) to *d*). The diagonally striped cyan bars refer to the default batching parameters and are labelled "hpcx default". The criss-crossed orange bars, labelled "hpcx custom", refer to inverse FFT with optimized batching parameters: four sub-batches of size equal to twice the number of nodes for the three smaller grids and equal to the number of nodes for the largest one. The green horizontally striped bars refer to the NCCL implementation of the library: the same batching parameters as the "hpcx custom" case were used. The blue dashed line (which refers to the y-axis on the right) represents the gain obtained by using NCCL over HPC-X (defined as the ratio between the time to solution obtained with HPC-X and that obtained with NCCL) in the "hpcx custom" configuration.

Chapter 5

Conclusions

This thesis presented several comparisons between the performances of NCCL and HPC-X (the NVIDIA optimized version of openMPI) on the CINECA EUROHPC JU Leonardo cluster.
The first group of benchmarks relied on mini-apps written to test specific communication patterns, both point-to-point (bidirectional ping-pong) and collective (all-reduce and all-to-all). The all-to-all pattern is particularly relevant as the same pattern is implemented in FFTXlib.
For the point-to-point, the results of these benchmarks are that NCCL performs similarly to or worse than HPC-X, which is not surprising since NCCL is specifically written to organize GPU-GPU collective communications. For the all-reduce benchmarks, NCCL consistently outperforms HPC-X for all the node configurations considered. For the all-to-all pattern, the performance gain becomes noticeable starting from buffer sizes of approximately $10^7 B$ for 4 and 8 nodes, and $10^6 B$ for 16 to 64 nodes.
For the FFTXlib benchmarks, the time-to-solution of an inverse distributed FFT (which involves and all-to-all communication pattern) was compared using HPC-X and NCCL as communication library for different grid sizes. The results align with the pure all-to-all benchmarks: for more than 4 nodes and large enough message sizes (but still within values used in QUANTUM ESPRESSO typical use cases) NCCL outperforms HPC-X.

5.0.1 Future developments

An interesting extension of this work would be to repeat the tests presented in this thesis on other machines with different hardware configurations and from different vendors. In particular, several vendors have implemented their own flavour of a library optimized for collectives. AMD has implemented the RCCL library [18] while Intel has released oneCCL [19].
Concerning QUANTUM ESPRESSO , one future line of development is to determine whether the gains in FFT times obtained in the miniapp are reproducible in real applications of QUANTUM ESPRESSO .
Another direction is to investigate other bottlenecks related to MPI collectives and test whether the implementation of the NCCL counterpart can enhance the overall performance of the code.

Appendices

Appendix A

Crash course on lattices and Fourier transform

The aim of this section is to introduce the basic notions about lattices in order to understand the Plane-Wave decomposition of the Kohn-Sham equation. For a detailed treatment, specialized texts are available [24] and [25].
A crystalline solid can be defined as an ordered arrangement of atoms organized in an ordered structure, generated by the repetition in space of an "elementary cell". To properly describe this structure we need two notions:

- (Bravais) Lattice: mathematical geometrical structure invariant under discrete translations.
- Basis: set of vectors associated to each lattice point that specifies the position of the atoms that generate the actual crystal.

In the literature, the word "lattice" typically stands for "generic ordered structure": from here onwards, unless otherwise specified, the word "lattice" will stand for "Bravais lattice".
Let be $(\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3)$ the vectors generating the smallest translation under which the lattice is invariant (these vectors are not uniquely defined). These vectors are called *primitive vectors* and their magnitude is typically of the order the Angstrom ($\sim 10^{-8}m$). The position of a generic lattice point can then be represented as a linear combination of the primitive vectors with integer coefficients:

$$\mathbf{R}_\ell = \sum_i^3 \ell_i \mathbf{a}_i, \quad \ell_i \in \mathbb{Z}. \tag{A.1}$$

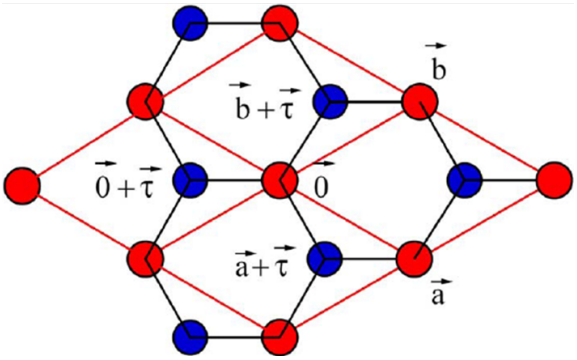


Figure A.1: Representation of the honeycomb lattice and decomposition as a Bravais lattice with a basis. Figure taken from `https://wiki.physics.udel.edu/phys824/Band_structure_of_graphene,_massless_Dirac_fermions_as_low-energy_quasiparticles,_Berry_phase,_and_all_that`

To prove this, consider an operator $\hat{T}_m$ that implements a translation belonging to the symmetry group of the lattice. Its action on a generic vector $\mathbf{v}$ is:

$$\hat{T}_m \mathbf{v} = \mathbf{v} + \sum_i m_i \mathbf{a}_i, \quad m_i \in \mathbb{Z}. \tag{A.2}$$

Then, from definition A.1 it immediately follows that $\hat{T}\mathbf{R}_\ell$ is of the same form as A.1, and thus belongs to the lattice:

$$\hat{T}_m \mathbf{R}_\ell = \sum_i (\ell_i + m_i)\mathbf{a}_i \equiv \sum_i (\ell_i') = \mathbf{R}_{\ell'}, \quad \ell_i' \in \mathbb{Z}. \tag{A.3}$$

Two points are said to be *equivalent* if they see the same landscape: from the considerations above, it follows that in a (Bravais) lattice all points are equivalent. Indeed this is a difference between a generic lattice and a Bravais one. Consider, for example, the so-called honeycomb lattice, depicted in black in Fig. A.1. The red and blue points are not equivalent because they see a different landscape. On the other hand, this structure can be decomposed as a Bravais lattice (depicted in red) with a basis given by the vectors $\vec{0}$ and $\vec{\tau}$, which point to the blue and red atoms, respectively.

Since the lattice is invariant under discrete translation, it can be constructed by iterating a "basic structure" in space. The choice of the latter is not unique: a generic structure that generates the lattice if iterated in space is called *unit cell*, and the unit cell with the smallest volume is called *primitive cell*. Even the primitive cell is not

uniquely defined, as different cells with the smallest volume can be constructed. A possible choice for the primitive cell is the parallelepiped generated by the primitive vectors whose volume is:

$$v_c = |\mathbf{a}_1 \cdot (\mathbf{a}_2 \times \mathbf{a}_3)|. \tag{A.4}$$

Another way to construct the primitive cell, is to select a point on the lattice and draw the segments joining it with the nearest points. Next, the planes perpendicular to these segments are traced through their middle points. The solid delimited by the intersection of these planes is called *Wigner-Seitz cell*. The advantage of this construction is that this cell preserves the symmetries of the lattice, as opposed to the cell generated by the primitive vectors. Finally, by construction the Wigner-Seitz cell contains exactly one lattice point. This is true also for any other choice of thee primitive cell, although the counting has to be performed in a "fractional" manner. Specifically, if a point belongs to k cells it counts $1/k$ for the total number of points in each cell.

One important consequence of the existence of the primitive cell, is that we can decompose the whole solid in a number N_c of cells, each reachable from the primitive cell via an appropriate lattice vector. Thus, an integral over the solid can be decomposed as a sum over integrals over the primitive cell as follows:

$$\int g(\mathbf{r})\, d^3\mathbf{r} = \sum_{\ell} \int_{v_c} d^3\,\mathbf{r} g(\mathbf{r} + \mathbf{R}_\ell). \tag{A.5}$$

A.0.1 The reciprocal lattice

Experimentally, it is often easier to observe the properties of a "dual lattice", called *reciprocal lattice*, defined as follows. Given the set of primitive vectors of the original lattice (which will now be called *direct lattice*) $\{\mathbf{a}_i\}$ the primitive vectors $\{\mathbf{b}_i\}$ of the reciprocal lattice are defined as:

$$\mathbf{a}_i \cdot \mathbf{b}_j = 2\pi\delta_{ij}, \quad i, j = 1, 2, 3. \tag{A.6}$$

The reciprocal lattice primitive vectors can be explicitly written as:

$$\mathbf{b}_1 = \frac{2\pi}{v_c}\mathbf{a}_2 \times \mathbf{a}_3, \quad \mathbf{b}_2 = \frac{2\pi}{v_c}\mathbf{a}_3 \times \mathbf{a}_1, \quad \mathbf{b}_3 = \frac{2\pi}{v_c}\mathbf{a}_1 \times \mathbf{a}_2, \tag{A.7}$$

where v_c is the volume of the direct lattice's primitive cell.
It follows that each vector denoting a point in the reciprocal lattice $\mathbf{G}$ can be written

as:

$$\mathbf{G} = \sum_i n_i \mathbf{b}_i, \quad n_i \in \mathbb{Z}. \tag{A.8}$$

Furthermore, given a vector of the direct lattice $\mathbf{R}_\ell$, it holds that:

$$\mathbf{G} \cdot \mathbf{R}_\ell = 2\pi n, \quad n \in \mathbb{Z}. \tag{A.9}$$

The cell in the reciprocal lattice corresponding to the Wigner-Seitz cell in the direct lattice is called *First Brillouin zone*, and its volume is:

$$v_{BZ} = \|\mathbf{b}_1 \cdot \mathbf{b}_2 \times \mathbf{b}_3\| = \frac{8\pi^3}{v_c}, \tag{A.10}$$

ù and, as the Wigner-Seitz cell in the reciprocal lattice, it contains only one reciprocal lattice point.

A.0.2 Fourier transform on lattices

Let $f(\mathbf{r})$ be a function expressing a physical property of the crystalline solid. Due to the crystal's periodicity, f has to satisfy the condition:

$$f(\mathbf{r}) = f(\mathbf{r} + \mathbf{R}_\ell). \tag{A.11}$$

The periodicity condition (A.11) imposes severe constraints on the allowed Fourier modes. Consider the Fourier transform of f:

$$\tilde{f}(\mathbf{q}) = \int d^d r \, f(\mathbf{r}) e^{-i\mathbf{q}\cdot\mathbf{r}} \tag{A.12}$$

$$= \sum_\ell \int_{v_c} d^3\mathbf{r} \, e^{-i\mathbf{q}\cdot(\mathbf{r}+\mathbf{R}_\ell)} f(\mathbf{r} + \mathbf{R}_\ell) \tag{A.13}$$

$$= \sum_\ell e^{-i\mathbf{q}\cdot\mathbf{R}_\ell} \int_{v_c} d^3\mathbf{r} \, e^{-i\mathbf{q}\cdot\mathbf{r}} f(\mathbf{r}), \tag{A.14}$$

where in the first step we used the decomposition (A.5) and in the second one the periodicity condition (A.11). The sum over direct lattice vectors is equal to:

$$\sum_\ell e^{-i\mathbf{q}\cdot\mathbf{R}_\ell} = \begin{cases} N_c & \mathbf{q} \text{ belongs to the reciprocal lattice,} \\ 0 & \text{otherwise.} \end{cases} \tag{A.15}$$

In fact, if $\mathbf{q}$ is a generic vector we have to sum over a macroscopic number of rapidly rotating phases, which average to zero. On the other hand, if $\mathbf{q}$ belongs to the

reciprocal lattice the exponent in (A.15) is zero and the sum is given by the number of lattice points, i.e. the number of primitive cells. Substituting (A.15) into (A.12) yields :

$$\tilde{f}(\mathbf{q}) = N_c v_c \sum_{\mathbf{G}} \delta_{\mathbf{q},\mathbf{G}} f_{\mathbf{G}}, \quad f_{\mathbf{G}} = \frac{1}{v_c} \int_{v_c} d^3r \, f(\mathbf{r}) e^{-i\mathbf{G}\cdot\mathbf{r}}. \tag{A.16}$$

Therefore, the only relevant Fourier components are the ones that correspond to reciprocal lattice vectors.
The inverse Fourier transform of (A.16) is then:

$$f(\mathbf{r}) = \sum_{\mathbf{G}} f_{\mathbf{G}} e^{i\mathbf{G}\cdot\mathbf{r}}. \tag{A.17}$$

The fact that only the modes belonging to the reciprocal lattice are relevant has severe practical consequences. In particular, the spacing between two generic vector in Fourier space, if L is the linear dimension of the sample, is given by:

$$\Delta q = 2\pi/L, \tag{A.18}$$

while the typical separation between two vectors in the reciprocal lattice is (if a is the direct lattice spacing) is:

$$\Delta G = 2\pi/a. \tag{A.19}$$

Since typically $L \sim 10^{-2}m$, while $a \sim 10^{-8}m$, it follows that the number of vector in the reciprocal lattice are 10^8 times less than the number of points in Fourier space.

Bibliography

[1] P. Giannozzi, O. Andreussi, T. Brumme, O. Bunau, M. Buongiorno Nardelli, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, M. Cococcioni, N. Colonna, I. Carnimeo, A. Dal Corso, S. de Gironcoli, P. Delugas, R. A. DiStasio, Jr., A. Ferretti, A. Floris, G. Fratesi, G. Fugallo, R. Gebauer, U. Gerstmann, F. Giustino, T. Gorni, J. Jia, M. Kawamura, H-Y Ko, A. Kokalj, E. Kucukbenli, M. Lazzeri, M. Marsili, N. Marzari, F. Mauri, N. L. Nguyen, H-V Nguyen, A. Otero-de-la Roza, L. Paulatto, S. Ponce, D. Rocca, R. Sabatini, B. Santra, M. Schlipf, A. P. Seitsonen, A. Smogunov, I. Timrov, T. Thonhauser, P. Umari, N. Vast, X. Wu, and S. Baroni. Advanced capabilities for materials modelling with QUANTUM ESPRESSO. *J. Phys. Condens. Matter*, 29(46):1–30, 2017. 465901.

[2] Paolo Giannozzi, Oscar Baseggio, Pietro Bonfà, Davide Brunato, Roberto Car, Ivan Carnimeo, Carlo Cavazzoni, Stefano de Gironcoli, Pietro Delugas, Fabrizio Ferrari Ruffino, Andrea Ferretti, Nicola Marzari, Iurii Timrov, Andrea Urru, and Stefano Baroni. QUANTUM ESPRESSO toward the exascale. *J. Chem. Phys.*, 152(15):1–11, 04 2020. 154105.

[3] `https://developer.nvidia.com/networking/hpc-x`.

[4] NVIDIA. Nccl source code. `https://github.com/NVIDIA/nccl`.

[5] MVAPICH. Osu micro-benchmarks. `https://mvapich.cse.ohio-state.edu/benchmarks/`.

[6] Juelich Supercomputing Centre. Jube. `https://github.com/FZJ-JSC/JUBE`.

[7] Daniele De Sensi, Lorenzo Pichetti, Flavio Vella, Tiziano De Matteis, Zebin Ren, Luigi Fusco, Matteo Turisini, Daniele Cesarini, Kurt Lust, Animesh Trivedi, Duncan Roweth, Filippo Spiga, Salvatore Di Girolamo, and Torsten Hoefler. Exploring gpu-to-gpu communication: Insights into supercomputer

interconnects. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 1–15. IEEE, November 2024.

[8] PyTorch Foundation. Pytorch documentation. https://pytorch.org/docs/stable/distributed.html.

[9] Google Brain. Nccl in tensorflow. https://www.tensorflow.org/s/results?q=nccl.

[10] NVIDIA. Nccl documentation. https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31880/.

[11] Sylvain Jeauge. Nccl: High-speed inter-gpu communication for large-scale training. Webinar, 2021. https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31880/.

[12] Sylvain Jeauge. Scaling deep learning training: Fast inter-gpu communication with nccl. Webinar, 2023. https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s51111/.

[13] NVIDIA. Nccl fortran interface. https://docs.nvidia.com/hpc-sdk/compilers/fortran-cuda-interfaces/index.html#cfnccl-allreduce.

[14] https://www.top500.org/, last consulted on 10/3/2025.

[15] Matteo Turisini, Giorgio Amati, and Mirko Cestari. Leonardo: A pan-european pre-exascale supercomputer for hpc and ai applications, 2023.

[16] Martin Dietzfelbinger Roman Dementiev Peter Sanders, Kurt Mehlhorn. *Sequential and Parallel Algorithms and Data Structures*. Springer Cham, 1 edition, 2019.

[17] NVIDIA. Nccl tests. https://github.com/NVIDIA/nccl-tests.

[18] AMD. Rccl github page. https://github.com/rocm/rccl.

[19] Intel. oneccl webpage. https://www.intel.com/content/www/us/en/developer/tools/oneapi/oneccl.html.

[20] Richard Martin. *Electronic Structure: Basic Theory and Practical Methods*. Cambridge University Press, 2004.

[21] Warren E. Pickett. Pseudopotential methods in condensed matter applications. *Computer Physics Reports*, 9(3):115–197, 1989.

[22] Frits Agterberg. *Fast Fourier Transform*, pages 1–5. Springer International Publishing, Cham, 2020.

[23] Joshua Romero, Everett Phillips, Gregory Ruetsch, Massimiliano Fatica, Filippo Spiga, and Paolo Giannozzi. A performance study of Quantum ESPRESSO's' PWscf code on multi-core and GPU systems. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 67–87, 2018.

[24] Charles Kittel. *Introduction to Solid State Physics*. Wiley, 8 edition, 2004.

[25] N. W. Ashcroft and N. D. Mermin. *Solid State Physics*. Saunders College, Philadelphia, 1976.