**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Sérgio Manuel Rodrigues Caldas

# Performance Tuning to Determine Electronic Properties of Materials with Quantum Espresso

November 2019

**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Sérgio Manuel Rodrigues Caldas

# Performance Tuning to Determine Electronic Properties of Materials with Quantum Espresso

Master Dissertation
Master Degree in Computer Science

Dissertation supervised by:
**Alberto José Proença**
**Ricardo Mendes Ribeiro**

November 2019

# AGRADECIMENTOS

A presente dissertação contou com o apoio, incentivo e motivação de algumas pessoas, sem as quais este trabalho não teria sido possível e às quais estarei eternamente grato.

Ao meu orientador, Prof. Alberto Proença, por toda a motivação, disponibilidade e dedicação demonstrado ao longo da orientação deste trabalho. As reuniões semanais sob a sua orientação tiveram um papel preponderante ao longo deste trabalho, quer pelas críticas construtivas, quer pelo seu acompanhamento regular. Ao meu co-orientador, Prof. Ricardo Ribeiro, pelo papel activo que teve, sobretudo na componente de Física deste trabalho, pela motivação, dedicação e interesse demonstrado. Ao Ícaro Jael, pela disponibilidade, ajuda e prontidão revelado na realização de testes que serviram de base de estudo desta dissertação.

À minha família, pela confiança e apoio incondicional. Um agradecimento muito especial à minha mãe, Rita Rodrigues, por ser aquela pessoa que nunca me desamparou e me apoiou em toda a minha vida e em todas as minhas decisões. Sem ela este trabalho nunca teria sido possível e nunca me teria tornado a pessoa que sou hoje. À minha namorada Angelina Vieira por toda a compreensão que teve, pelas noites sem descanso e por todo o apoio e incentivo que me deu, sem dúvida que teve um papel muito importante neste meu percurso.

A todos os meus amigos, por todas as horas de descontração que me proporcionaram nos momentos mais difíceis durante o meu precurso académico e sobretudo pelo apoio incondicional. Um agradecimento muito especial ao Filipe Oliveira, Márcia Lomba, Carlos Sá, Carla Couto, Catarina Barbosa e Mafalda Varanda por serem as pessoas que me acompanharam em todo o meu percurso académico, pelos "puxões de orelha", pela paciência e por o todo apoio na dissertação e na licenciatura.

Por fim, e não menos importante, um agradecimento muito especial à Dra. Elsa Dourado, por todo o suporte que me proporcionou, pelas técnicas e ajuda que me fizeram ultrapassar uma mão cheia de desafios, e por toda a paciência demonstrada. Sem ela este percurso teria sido muito díficil.

Esta dissertação foi realizada no âmbito do Projeto "UTAustin | Portugal - UTA-P Gestão 2013 (Protocolo entre FCR-UNL e UMinho)", financiado pela Fundação para a Ciência e a Tecnologia, através do Programa de Cooperação Internacional UT Austin Portugal para as Tecnologias Emergentes, CoLab.

# STATEMENT OF INTEGRITY

# RESUMO

***Ajuste de Desempenho para Determinar Propriedades Eletrónicas de Materiais com Quantum Espresso***

Desde os anos 70, a teoria do funcional da densidade tem sido uma das técnicas mais utilizadas em física quântica para resolver a equação de Schrödinger, para determinar as propriedades eletrónicas dos materiais, usando funções de onda eletrónica e a energia de cada eletrão. O método de resolução do campo autoconsistente (sigla em inglês SCF) é um processo iterativo que calcula a densidade dos eletrões a partir de funções de onda. Estes cálculos com a equação de Schrödinger são realizados múltiplas vezes de forma sucessiva até se atingir a convergência autoconsistente.

O *SCF* neste processo iterativo é atualmente calculado em pacotes de software dedicado, como o QUANTUM ESPRESSO *(*QE*)*, um produto *open-source* em Fortran 90, para cálculo da estrutura eletrónica dos materiais. Sendo estes cálculos computacionalmente intensivos, a sua execução paralela permite melhorar o desempenho do processo de cálculo. O QUANTUM ESPRESSO *(QE)* suporta paralelismo em ambiente de memória distribuída, com *message passing interface* (*MPI*) e, mais recentemente, em memória partilhada, com *OpenMP*.

A presente dissertação apresenta várias propostas de instalação e configuração desta ferramenta. Estas propostas sugerem diferentes estratégias de paralelismo tendo em vista obter melhorias de desempenho deste tipo de simulações, comparativamente a um estudo anterior realizado nas mesmas condições de experimentação. Para o presente estudo foram utilizados processadores multicore e many-core do cluster *SeARCH*. Estes testes exploraram o impacto de versões multiprocesso com múltiplos fios de execução por processo, introduzidas em versões mais recentes do QE com desenvolvimento de paralelizações híbridas.

Através de diferentes casos de teste, diferentes instalações e parâmetros configuráveis (número de *pools*) este trabalho explorou e procurou obter um ambiente de execução que melhor favorecia o desempenho de simulações do *tungsten diselenide (*$WSe_2$*)* no cluster *SeARCH*. Os resultados obtidos nestes testes, onde se aconselham certas configurações e se desaconselham outras, destinam-se a ajudar as comunidades de Física a encontrar um ambiente de execução afinado em termos de desempenho, para o caso concreto deste tipo de simulações.

**Palavras-Chave:** Biblioteca ELPA, Computação Paralela, Disselénio de Tungsténio, Eficiência Computacional, Intel KNL, Quantum Espresso

# ABSTRACT

*Performance Tuning to Determine Electronic Properties of Materials with Quantum Espresso*

Since the 70's, *density functional theory (DFT)* has been one of the most used techniques in quantum physics to solve the Schrödinger equation. The resolution of this equation assumes a prominent role in the characterization of the electronic properties of the materials, with the use of wave functions and the energy of each electron. The computation method follows an iterative process, known as *self consistent field (SCF)*, to compute the electrons density from an initial set of wave functions. This iterative process successively recurs to the Schrödinger equation until it reaches a self-consistence convergence.

The SCF computation uses QE, an open-source software package written in Fortran 90, to determine the electronic structure of materials. This calculation is computationally very intensive, requiring an adequate support for parallelism to improve the computation performance to reach the convergence point. The QE already has *message passing interface* (*MPI*) support for distributed memory systems and recently introduced support for shared memory parallelization with *OpenMP*.

This dissertation presents alternative approaches to adequately install and configure QE in a compute cluster with distributed memory nodes, where each node contains one or more multicore devices sharing the same memory address space. These approaches suggest different parallelism strategies that will reflect performance improvements on simulations, when compared to an earlier study, conducted under the same experimental conditions. The testbed uses multicore and many-core processors from the *SeARCH* cluster to measure the impact of multi-process simulations with multiple threads per process, recently introduced in QE with hybrid parallelizations.

Using different case studies and through different installations and parameters configurations (number of pools), this work explored and aimed to reach an efficient execution environment for the simulations of tungsten diselenide ($WSe_2$) in the *SeARCH* cluster. The obtained results in the experimental tests aim to help Physics communities to find the best performance environment for this type of simulations.

**Keywords:** Computational Efficiency, ELPA-library, Intel KNL, Parallel Computing, Quantum Espresso, Tungsten Diselenide

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF LISTINGS

# ACRONYMS

**API** application programming interface. 20

**BZ** Brillouin zone. 7, 9, 20

**CP** Car-Parrinello. 20

**DFT** density functional theory. vii, xi, 1–6, 9, 19, 20, 24, 59
**DTD** distributed tag directory. 15

**ELPA** Eigenvalue soLvers for Petaflop Applications. xii, 35, 38, 41–45, 51, 53–57, 60, 61, 65, 66, 68, 69, 71–77

**GPU** graphic processing unit. 22

**HBM** High-Bandwidth memory. 13, 14, 17, 18, 35, 61
**HLL** High-level language. 35
**HPC** high performance computer. 9, 47

**KNL** Knights Landing. 3, 13–15, 17, 18, 34, 35, 38–42, 59–62

**MCDRAM** multi-channel dynamic random access memory. 13–18, 34, 40, 56–58
**MKL** Math Kernel Library. 24, 25, 27, 41, 43, 59
**MPI** message passing interface. v, vii, xi, 2, 11, 20–22, 24–29, 38–41, 49, 53, 66, 68, 69, 71

**NCPP** norm-conserving pseudopotentials. 20
**NEB** nudged elastic band. 20, 28
**NSCF** non-self consistent field. 1
**NUMA** non-uniform memory access. 12–14, 16, 18, 38, 47

**PP** pseudopotentials. 2, 9, 20
**PW** plane waves. 1, 2, 8, 9, 20, 21

**QE** Quantum ESPRESSO. v, vii, ix, xi–xiii, 1–4, 9, 11–13, 19–29, 31, 34–42, 46, 49–51, 53, 54, 57, 59–63, 66–68, 70, 71, 73, 74, 76, 77

**RT** reference time. xi, xii, 31, 45, 49, 50, 52, 53, 55–57

**SCF** self consistent field. v, vii, 1, 5, 8, 9, 19, 20, 36, 37

**SeARCH** **Se**rvices and **A**dvanced **R**esearch **C**omputing with **H**TC/**H**PC. v, vii, xi, xiii, 32–35, 39, 40, 42, 45, 49, 59, 62

**SMT** simultaneous multithreading. 46

**SNC** Sub-NUMA Clustering. 14, 16

**TD** tag directory. 15, 16

**USPP** ultrasoft pseudopotentials. 20

$WSe_2$ tungsten diselenide. v, vii, xi, xiii, 2–4, 6–8, 20, 31, 32, 34–37, 43, 44, 51, 54, 59–61

# 1 INTRODUCTION

The density functional theory (DFT) is the most used method in computational quantum mechanical modeling. This theory is used in physics, chemistry and material sciences to calculate the electronic structure of many-body systems, typically solids and molecules. It departs from the notion that it is not necessary the full knowledge of the wave function of the electrons to determine the electronic properties of the system: the electron density is sufficient. To build the electron density, the *Kohn-Sham* scheme uses single electron wave functions to match the real density. These single particle wave functions can be very close to the real ones, if the electron-electron interaction is not very strong.

To build the single particle wave functions, a basis of functions is needed. The most common ones are: (i) a set of Gaussian functions, which are more widely used by chemists to study molecules, and (ii) *plane waves (PW)* basis, which are more used by physicists to study the solid state. For small systems, the calculations of the electronic structure of materials are readily done and with fast calculation times. The size of the calculations is given by the number of included electrons and the numerical precision of the calculus. The numerical precision is defined by the size of the PW basis used to describe the single electron wave functions. In general, for each electron and electron state, one full PW basis is needed. Once computed, these wave functions are usually stored in disk for later use.

These calculations are an iterative process, known as SCF, in which we first use a set of trial wave functions to build an initial density and then, using the potential that this electron density generates, a new set of wave functions is calculated that will be used to build another electronic density. This process is repeated until it converges. Once the electronic density has converged, it is saved and used latter to calculate as many electronic states (wave functions and their energies) as needed for the desired properties. These calculations are called *non-self consistent field (NSCF)*. Some properties like the dielectric function (which defines the optical properties of the material) are then calculated using these electronic states.

There are several software packages available to perform the described calculations, for instance, the QE package. The QE is a very complete open-source software package for the calculation of the electronic structure of materials. Furthermore, this software already supports parallelism in distributed systems (with MPI) and in shared memory systems (with *OpenMP*), which is an added value since these calculations can take several hours or even days.

The Quantum ESPRESSO is composed by multiple computation modules, among which, *PWscf*. This is the core module of Quantum ESPRESSO using pseudopotentials and plane wave basis. The module contains the DFT functions for electronic structure, density functional perturbation, and density functional theory computations. Before being integrated in Quantum ESPRESSO, the *PWscf* module already existed as an independent project. *PWscf* is the main software package used to solve Kohn-Sham equations in the simulation of the inorganic compound tungsten diselenide ($WSe_2$) presented in chapter 2. The module can be used for atomic forces, structural optimizations, molecular dynamics on the ground-state Born-Oppenheimer surface simulations, macroscopic polarization, electric fields through modern polarization theory (associated with Berry Phases), and many others.

Throughout this dissertation a performance study is introduced using several variations of the use of the *PWscf* module. A set of different *PWscf* installations were made using multiple configurations in different computational platforms. The main goal to achieve is to determine the best performance execution environment to simulate in Quantum ESPRESSO the case study: the tungsten diselenide coumpound.

## 1.1 motivation and goals

The ***PWscf*** is the core component of the QE package that performs the electronic properties calculations of materials in any crystalline structure, both insulating and metals. The process of calculation uses density functional theory, plane waves (PW) and *pseudopotentials (PP)*. Since this calculation requires the solution of an equation system with a high number of PW, the computational performance plays a relevant role.

Over the past decades, DFT simulations were being seen as physics problems that require high performance tools, for applications in areas like industrial research in order to develop novel materials. QE, and namely his module *PWscf*, stands out as being one of the key performance bottlenecks to reach efficient executions of these simulations. This software is currently considered the state-of-the-art application to explore DFT related simulations in multiple areas, such as research of nanomaterials or catalysis. During the past decade, several scientific papers used DFT to explore new materials for future technologies, such as $WSe_2$, which is the case study of this work [5].

The QE tool already supports parallelism in shared and distributed memory computing environments. However, the increase in the simulations performance is related not only with parallelism strategies, but also with the software installation and configuration on the computer systems where the simulations are performed. Different strategies of parallelism, together with an adequate installation and configuration of the QE tool, present different performance levels and challenges. This dissertation aims to study and analyze the performance levels on computer systems based on multicore and many-core devices, such as the Intel Xeon Phi with the *Knights Landing (KNL)* architecture.

The many-core approach offers the possibility of improving the performance and go further in the solution accuracy by taking advantage of the parallel features available in QE, whether using shared memory or distributed memory paradigms. There is an opportunity to explore performance improvements by tuning the installation and configuration process for these architectures. The performance of these simulations are a trade-off between the performance achieved by solving the Schrödinger equation, to obtain energies and forces, and the accuracy of the final solution.

## 1.2 CONTRIBUTION

The work developed for this dissertation aimed to be a direct contribution to the scientific community of Quantum Physics. The main contribution of this work is to offer an execution environment with better performance than the default QUANTUM ESPRESSO installations used to run $WSe_2$ simulations on a massively parallel supercomputer. The present work proposes an overall approach with better performance, running these simulations on *clusters*, with more efficient solver libraries, and tuning the parameters of the software package that affects performance. The simulations of $WSe_2$ case study, run in the core package of QE - the *PWScf* - used for self-consistent computations of electronic structure properties in density functional theory (DFT). This package uses Plan-Waves basis and a set of pseudopotentials of the case study as input. To achieve a better performance environment, multiple *PWscf* installations are used, with a comparative analysis of their performance based on the simulation execution time.

The performance results achieved with these installations takes previous study results as a comparison basis, using the same case study, similar test conditions, and same hardware. The tests include simulations using multithread and multiprocess implementations of *PWscf*. The evaluation of the results was made based on the solution's convergence criteria, on the total energy variation of the system, and estimated Harris-Foulkes energy value. These values were obtained with the tool, according to the case study at each process iteration.

After reading this study, Quantum Physics community will be able to pack and make custom installations of *PWscf* with better performance for $WSe_2$ simulations in a cluster environment.

## 1.3 DISSERTATION STRUCTURE

Following this introduction, this document is split in 4 more chapters. The chapter 2 corresponds to the state of the art which approaches the electronic structure of materials used as a case study to validate the performance improvements, tungsten diselenide ($WSe_2$) and the computation side of DFT calculations. Chapter 3 presents the computational efficiency in QE, starting with the description of the target computing platforms, the QE as the key software package, the installations and tuning of the QE package followed by the challenges to improve the computational efficiency of the QE. Chapter 4 explores the whole experimental work, the results validation with different configuration strategies and installations of QE. The work is done in two computational nodes, the first equipped with a multicore and the other with a many-core device. This chapter also explores the execution in a distributed and shared memory environments using one or more nodes.

As a closing of the developed work, chapter 5 summarizes the achieved experimental results, drawing the main conclusions with a critical review and suggestions of new challenges for future work.

# 2 THE ELECTRONIC STRUCTURE OF MATERIALS

To characterize the electronic properties of materials, the wave function is needed for all electrons in the system. This wave function is a function of $3N$ variables, where $N$ is the number of electrons, and it is a solution of the $N$-electron Schrödinger equation, which is prohibitively difficult to solve, even for a very small number of electrons.

Density functional theory (DFT) solves this problem by noticing that, to obtain the properties of a system, one does not need to have all the information that an all-electron wave function has, but just the electron density $n(\vec{r})$ is enough. The electron density is a function of just the three spatial variables which is a much easier solution. The biggest problem to overcome was how to obtain an accurate density value without having to use the actual all-electron wave functions.

The solution came with the Kohn-Sham scheme [7] which uses single electron wave functions to build the density. These wave functions are a solution of a single electron Schrödinger equation, constructed using a potential that is obtained by an iterative method. This iterative process, named self consistent field (SCF), is the starting point of the procedure used in DFT and is used to determine the potential that will be used as the input to the single electron Schrödinger equation. As a result of this equation, a set of single electron wave functions is obtained and used to build the electron density. This process is repeated until the electron density converges, as stated in Figure 1.

Once the electronic density has converged, it is saved and used latter to calculate as many electronic states (wave functions and their energies) as needed for the properties wanted.

The properties of the system (including the total energy) can be obtained by an appropriate function of the electron density. In many cases, the exact function is unknown, and approximations have to be made.

One approximation often used is to consider the single electron wave functions and their energies as the real ones, and make calculations using this single electron approximation. An example is the dielectric function that defines the optical properties of the material.

For more information on DFT see references [6] and [7].

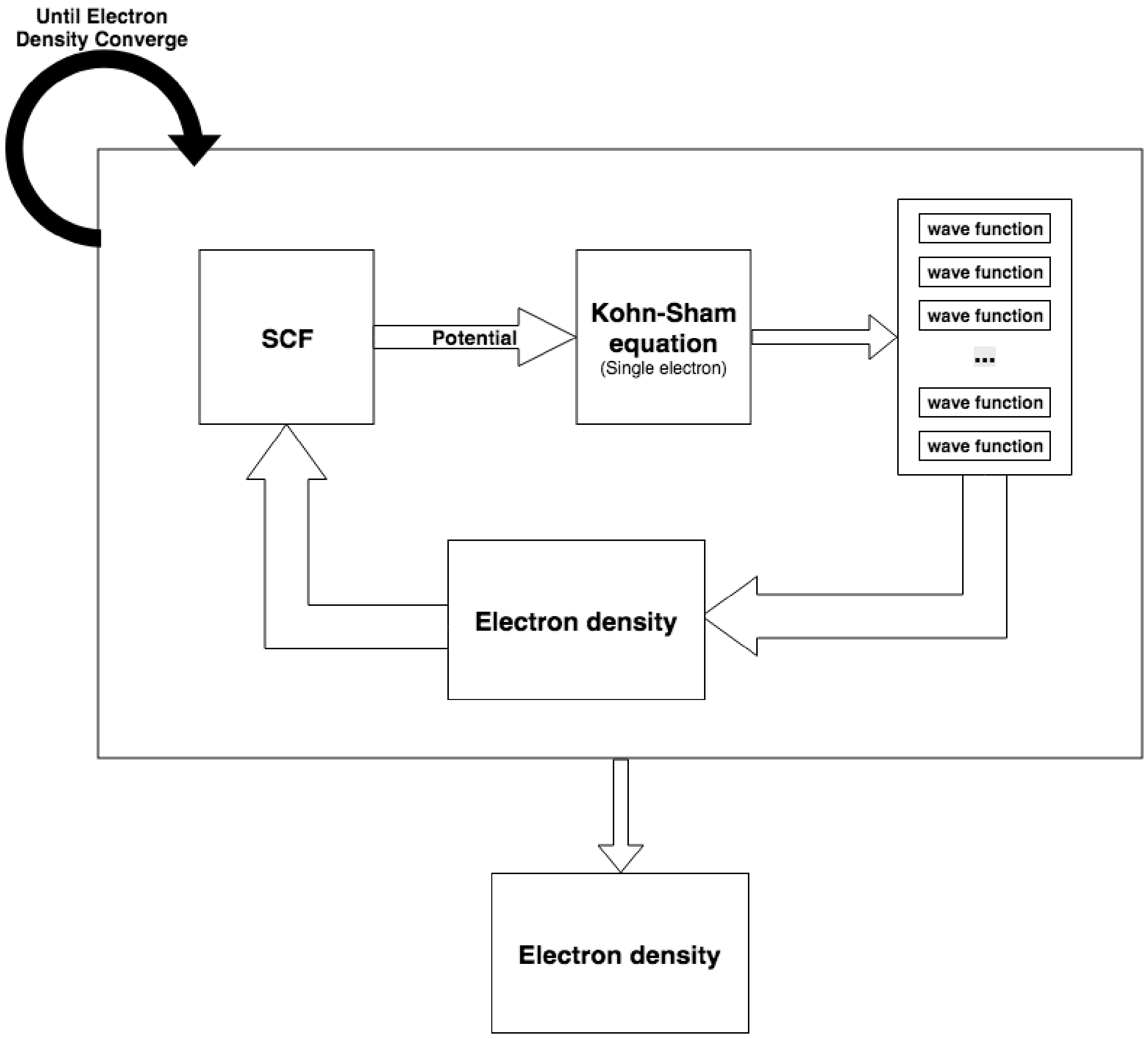


Figure 1.: Density functional theory (DFT) calculation diagram

## 2.1 CASE STUDY: $WSe_2$ MULTILAYERS

The systems used as a case study are the tungsten diselenide ($WSe_2$) multilayers [1] that may be built by stacking $WSe_2$ single layers, like the one shown in Figure 2.

These materials are two dimensional crystals, which means they have translation symmetry in the plane but not in the perpendicular direction to the plane. To have translational symmetry means that there is a unit that is repeated indefinitely in the plane, and that unit is called unitcell, as shown in Figure 3.

We can use this repetition to simplify the calculations by using the reciprocal space (meaning the Fourier transform of the crystal). With this, the problem is then reduced to a single

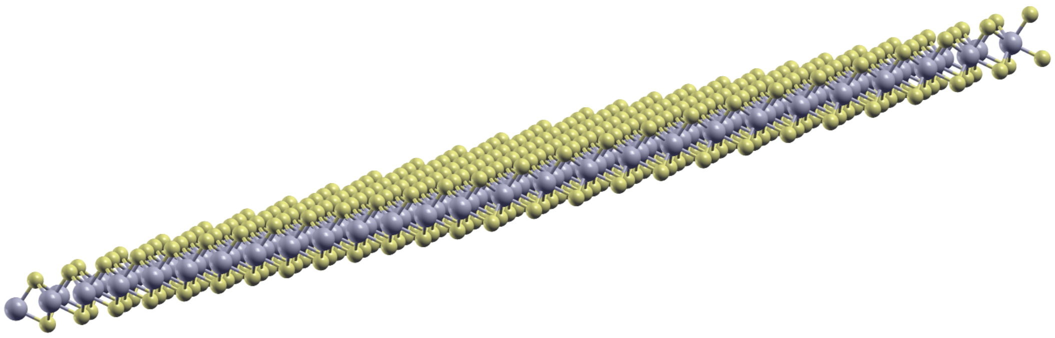


Figure 2.: $WSe_2$ single layer.

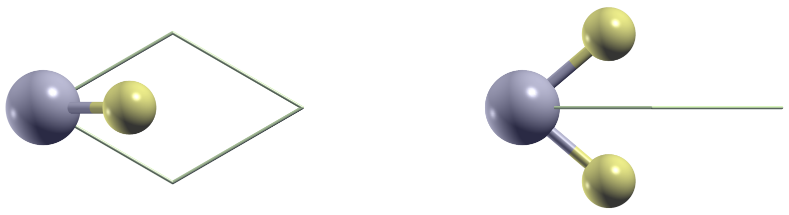


Figure 3.: $WSe_2$ single layer unitcell (left: top view; right: side view).

cell in reciprocal space, which is called the *Brillouin zone* (*BZ*). The coordinates in the reciprocal space are wave vectors $\vec{k}$ and not space coordinates, since the reciprocal space is the Fourier transform of the real space.

In our calculations we will frequently need to integrate in the reciprocal space, which means that we will have to choose a sample of wave vectors $\vec{k}$ (usually called $k$ points) in the *Brillouin zone (BZ)*. The more points chosen, the better the accuracy of the calculated integral.

Besides that, for each $k$ point we will have a set of wavefunctions $\psi_{m,k}$ and energies $E_{m,k}$. Each set of wavefunctions with the same $m$ is called a *band*. Each band can have a maximum of one or two electrons, depending if the spin is included or not in the calculation.

For this material, we use full relativistic calculations; then, for each electron there will be a spinor with four components i.e. there will be four functions to describe the state $\psi_{m,k}$. Each band will have at most one electron [6].

To make the SCF calculations, one needs one band for each electron in the unit cell. A tungsten atom has 6 valence electrons and a sulfur atom has another six. Then each unit cell of a single layer will have 18 electrons. For multilayers, we will have 18 electrons per layer. Figure 4 shows eight single layers stacked to form a multilayer.

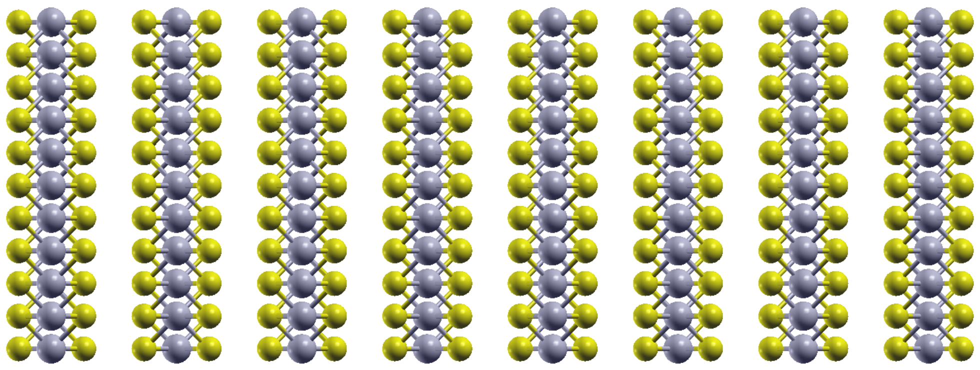


Figure 4.: $WSe_2$ multilayer (8 layers).

## 2.2 THE COMPUTATIONAL POINT OF VIEW

To describe the wave functions, a set of well known functions is needed in order to form a basis. The Gaussian and plane waves (PW) are the most used functions to form a basis set, in which by applying a linear combination, the wave functions are obtained.

In this work, only PW are used. For a PW basis, the more basis functions that are used, the better and more accurate the description of the wave function will be, but the more processing time is required. This leads to a trade off between accuracy and resources: a higher number of basis functions leads to a better accuracy, but a larger number of computational time and resources is needed. In practice, there is a cutoff in the series expansion that is determined by an energy (the energy that an electron with that wave vector would have), which is called $E_{cut}$. The larger the $E_{cut}$, the better the accuracy, but also the larger the computational resources that are needed [6]. The $E_{cut}$ value determines the accuracy of the calculations necessary to predict forces, pressures or other properties of the material.

As shown in Figure 1, the Kohn-Sham equations are central in the iteration process. These equations are a set of coupled partial differential equations. They are translated to a set of matrix operations, which are then computed. The size of these matrix depends on the number of plane waves in the basis set, which depends on the variable $E_{cut}$.

Each wave function describes one electron state. In a SCF calculation for a crystal (which is the case in this work), one state is used for each electron and for each point considered in the reciprocal space of the crystal. These points in the reciprocal space are a set of so called $k$ points that cover uniformly the Brillouin zone of the crystal, and are used for sampling when integration in the reciprocal space is needed. The larger the number of $k$ points, the better the quality of the integration, and so the better the numerical precision of the full calculation.

There are three aspects that define the weight of the computation: the size of the basis set ($E_{cut}$), the number of $k$ points and the number of electrons. The first two parameters affect the quality of the numerical precision and they can be modified in order to obtain a preferred balance between accuracy and resources. The number of electrons depends on the system under study but, usually, only the valence electrons are used since they are the most relevant for these calculations. Most of the electrons of the atoms are then hidden in pseudopotentials, and are not explicitly taken into account leading to an increase of the computational efficiency involved in the simulation [9] [11]. The quality associated with pseudopotentials is defined by the accuracy they provide in electron computations.

The scientific community has multiple DFT software codes available to explore different kinds of materials. These codes are mainly distinguished from each other by the basis set they use and the algorithm to solve the linear system in the Kohn-Sham equations. Most of them allow the use of several different types of pseudopotentials and exchange correlation functionals. QUANTUM ESPRESSO or *VASP* are examples of codes that use pseudopotentials and plane waves, and enable high performance calculations of electronic structures. This study uses the QE package.

DFT material simulations usually do not require a *high performance computer (HPC)*. A simulation can be made using a farm of multiple computer nodes forming a cluster with multiple multicore devices, with an high amount of memory and fast inter-communication buses.

# 3 EFFICIENCY IN QUANTUM ESPRESSO INSTALLATION

There are multiple platforms supported by Quantum ESPRESSO (QE). The support level of this tool guarantees their compatibility with homogeneous platforms.

QE offers installation support and configuration for multiple computing platforms. Homogeneous architectures have one or more multicore/many-core devices with the same architecture on the motherboard with parallelism capabilities on which QE can be used. However, there are some concerns regarding the efficient use and exploitation of these devices to determine the electronic properties in QE: the efficient map of pools, plane waves and task group parallelization among available processing units are some of the challenges that are needed to be addressed.

## 3.1 TARGET COMPUTING PLATFORMS

When using single-thread applications, multicore devices should reach a higher level of performance, as vector single-threaded applications. The QE implementation used in this dissertation is based on a multi-thread and multi-process (MPI) paradigm, which offers parallelism at both thread and process levels.

The main focus for this dissertation is to measure the impact of these implementations, running the electronic simulations on these computing platforms. As stated in Figure 5, multicore based platforms, usually adopt multiple processing units (known as "*cores*") in a single device (chip). The CPU instructions are addressed to different cores and processed at the same time (in parallel). Each multicore device usually has its own L3 and Random Access Memory address space, shared by all cores in the chip. To scale the parallel capabilities, multiple multicore devices can be attached together using interconnected based buses for communication purposes.

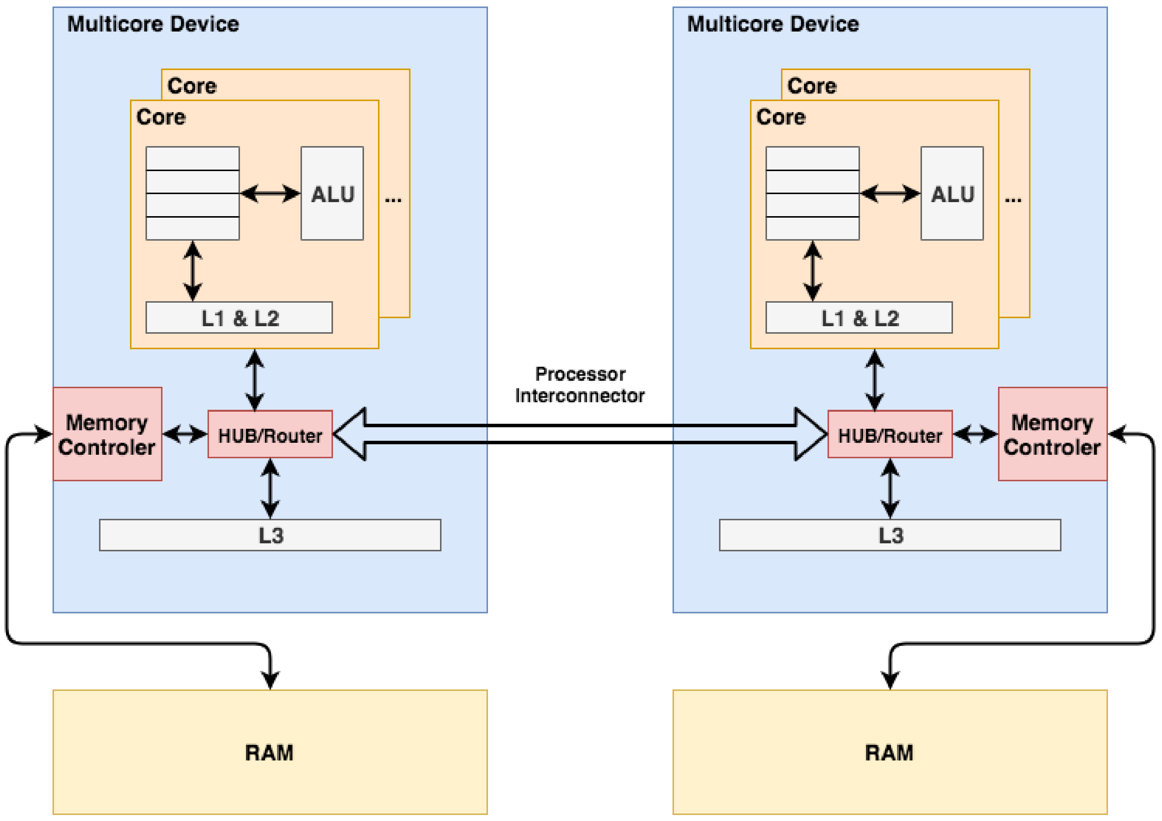


Figure 5.: Overview of a NUMA dual-multicore device

Due to the Amdahl's law, computing platforms evolved in a sense of increasing the number of cores with lower clock frequencies instead of increasing the clock frequency. This approach quickly gave rise to many-core devices, with a significant higher number of cores compared to multicore devices, surpassing 40 cores on a single chip.

The parallel work in *PWscf* (and in QE in general) is made by splitting the calculations and by the division of data structures (pools, bands or images) between the cores to be processed in parallel. These images and pools are loosely coupled so they usually imply a low inter-processor communication. The same does not apply for processors within each pool which are tightly coupled so the communication costs can be higher. These characteristics mean that the interconnect technology is important if the number of pools extends over more than a few processors on different *non-uniform memory access (NUMA)* nodes.

Alongside with multicore processors, many-core devices have emerged with tens and hundreds of processing units. These devices offer the capacity for a high level of explicit parallel processing and higher throughput than multicore devices. In multicore devices, the number of processing units are significantly lower compared with many-core devices. They are usually designed to run both serial and parallel applications, with higher number of superscalar units and larger caches compared to many-core devices. Multicore and many-core devices have been studied over the last few years since they were commonly adopted as the main computing platforms to run the eletronic-structure calculations and materials modeling in *PWscf*. However, performance concerns arised in the simulation process on these computing platforms which lead the performance tuning to be the frontside challenge in the scientific community and QE developers. Some of this challenges related to the installation process and case study parameters configuration will be explored using both multicore and many-core devices. An example of a many-core device commonly adopted to run *PWscf* (and also used in this study) is represented in Figure 6.

Regarding this many-core device, the cores are organized in a mesh fashion within the chip, with private L1 and L2 cache shared only between cores of a single tile. The memory accesses are performed by the cores using an *XY* routing rule. Following this strategy, memory accesses and message communications are first performed in a vertical propagation until the target row is reached and then, in a horizontal propagation through the interconnected buses until the destination is reached [10]. Data propagation between cores and *multi-channel dynamic random access memory (MCDRAM)* is controlled by heuristics responsible for the minimization of the access distance of the propagation. The number of hops is a metric that classifies the distance implied in a memory access between the cores and the main memory which corresponds to the number of bus interconnect crossings implied in a memory access.

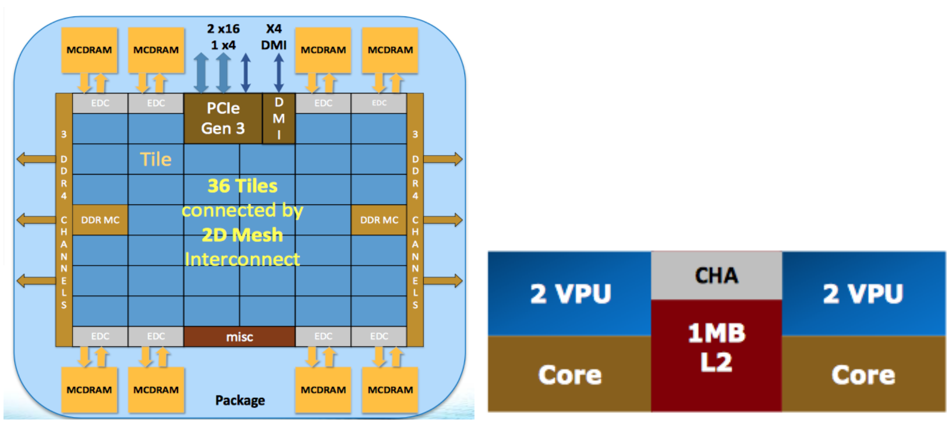


Figure 6.: **Left:** Intel Xeon KNL. **Right:** A Tile of Intel Xeon KNL [10]

The Knights Landing offers the possibility of reconfigure the cluster mode and *High-Bandwidth memory (HBM)* modes. The cluster mode re-configurations affects the organization of the NUMA nodes leading to a direct impact in the way memory accesses are made, which affects performance. The HBM modes offer the possibility to reconfigure the MC-

DRAM to be used as cache, main memory or a mix of both. In the following sections the KNL cluster and HBM modes are explored in detail.

### 3.1.1 *Cluster modes in KNL*

The affinity control between software threads and the hardware is a typical challenge when parallelism and performance are the main goal. The way software threads are assigned to specific processing units and the physical distance between them and the main memory has a direct impact in the latency of memory accesses.

The multicore and many-core devices faced these concerns over the years since computer architectures have provided a design solution where every processing unit is located at the same distance to the main memory and with the same memory access latency. This is also a problem faced in the KNL architecture - depending on the relative position of a tile in a processor grid and the MCDRAM he needs to access, the memory access is made in a non-uniform way similar to NUMA in multicore devices. This is specially problematic in a many-core architecture where the number of cores is significantly higher which aggravates the communication delay between cores and MCDRAM. The bottleneck in communication can be minimized using affinity.

The KNL provides three clustering modes to configure the way tiles access the MCDRAM. Depending on the activated cluster mode, each tile privileges the memory access to the closest MCDRAM. This type of affinity minimizes the distance that is necessary to go through in a L2 miss and minimizes the communication delay between tile and memory. The available cluster modes are:

- All-to-all
- Hemisphere/Quadrant mode
- *Sub-NUMA Clustering (SNC)* Mode (SNC-2 and SNC-4)

The all-to-all mode (represented in the Figure 7) is the less restrictive memory access mode. In this mode each tile accesses MCDRAM randomly and there is no restriction defined for the way the chosen tiles and the MCDRAM obtains the data. There is no affinity defined so the expected performance is the lowest of all the cluster modes available, since each memory access can take a long path between the tile that incurred in the miss and the memory that owned the data [10].

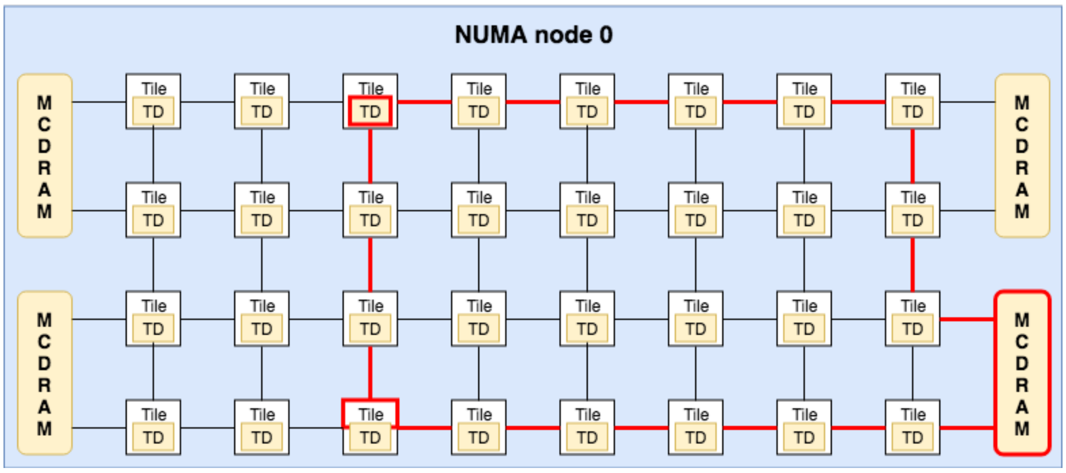


Figure 7.: All to all

The hemisphere/quadrant mode defines a logical division between all the tiles. The tiles are equally divided into 2 groups (hemisphere) or 4 groups (quadrants), and each group is assigned to the closest MCDRAM. The tiles in a group privileges the memory access to the MCDRAM of its own group. However, this group mode does not restrict the memory access to its own group because a tile can communicate with a tile from a different group and access its MCDRAM to obtain the data as is illustrated in Figure 8.

The yellow tile in the figure performs the request line of cache across yellow and red tiles until reach the *tag directory (TD)*[1] tile (red).

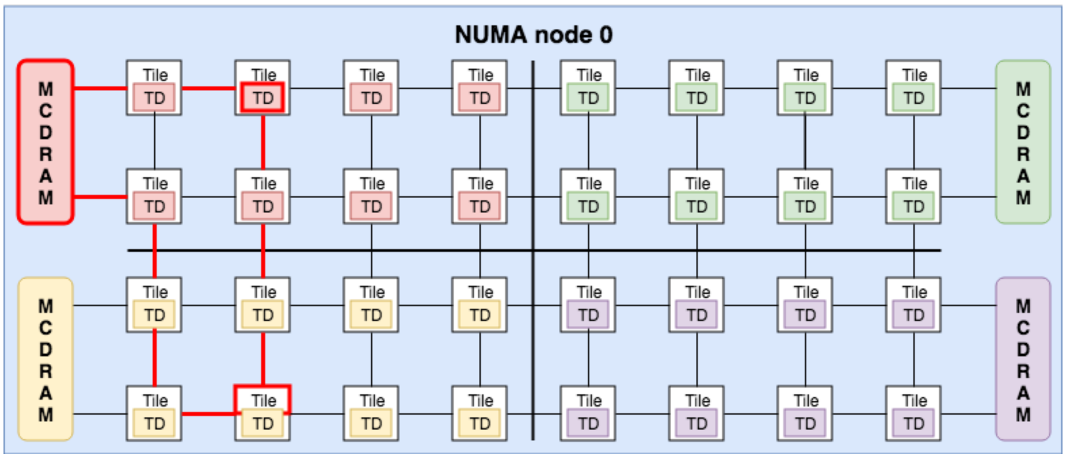


Figure 8.: Quadrant mode

1 KNL uses a *distributed tag directory (DTD)* for cache coherency, each tile has his own TD that identifies the location and the state of each line of cache on the chip.

Since the clustering mode is defined to quadrant mode, the TD obtains the data from the MCDRAM of its own quadrant only. The line of cache returns back to the yellow tile where the miss occurred. The longest path in a memory access in quadrant mode is shorter than in the all-to-all mode (and so the incurred latency)[2] leading the quadrant mode to have a more effective memory access [10].

Similar to the quadrant mode, the Sub-NUMA Clustering (SNC) configuration (shown in Figure 9) is also based in a logical division of tiles and MCDRAM, which can be divided in two or four parts (hemisphere and quadrant respectively). Compared to the other two, this mode configuration is more restrictive to how the MCDRAM accesses are made: a miss incurred by a certain tile is satisfied by a directory and a memory controller of the NUMA domain where that tile belongs, and no one from the others. This division establishes an affinity commitment between each group of tiles and the correspondent memory controllers of each NUMA node. This cluster mode provides the best minimization of communication latency when a miss occurs since all the traffic is self-contained in each NUMA node. However, in order to take full advantage of this cluster mode, the software needs to be NUMA aware: the memory allocations and software threads must occur in the same NUMA node.

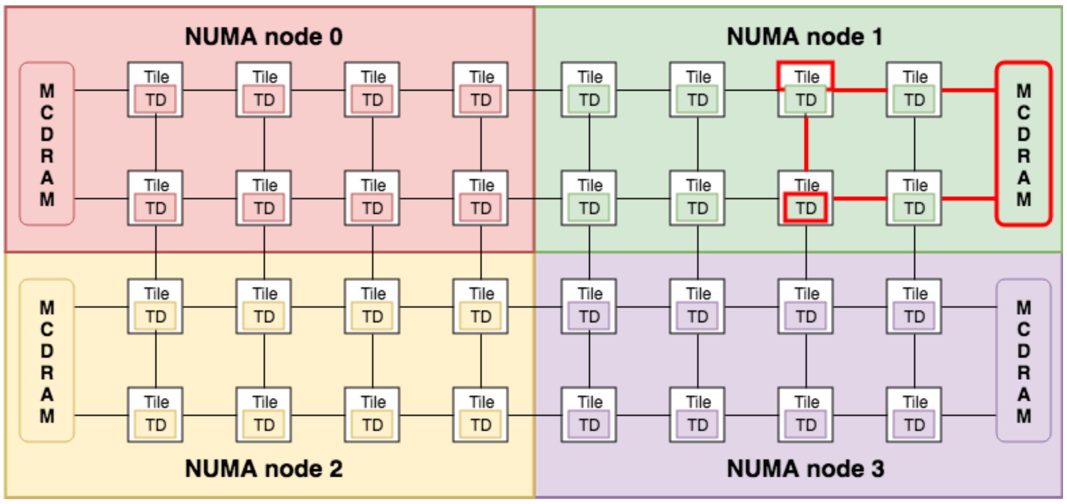


Figure 9.: Sub-NUMA Clustering mode

There are some tools that allows to take control over the affinity policy as the `numactl` for memory allocations and *taskset*/OpenMP for thread pinning in both quadrant or SNC cluster configurations.

2 Source: https://colfaxresearch.com/knl-numa/

### 3.1.2 *High-Bandwidth memory modes in KNL*

In the last decades, processor and memory manufacturers focused their attention to reduce memory bottlenecks in computer systems. This efforts has been translated into caching mechanisms, multiple memory hierarchy levels, and improved memory access technology.

Computer applications based in memory bound algorithms are the most negatively affected by memory bottlenecks. This types of applications have more frequent memory accesses leading to a higher impact in performance.

The high memory bandwidth demanded by memory bound applications is handled by the on package *High-Bandwidth memory (HBM)* modes available in KNL, using MCDRAM memory technology. These modes have different caching strategies and offers the possibility of using the MCDRAM as cache, as regular addressable memory or both. The technology behind MCDRAM offers a memory storage solution with up to 5 times the memory bandwidth when compared to regular DDR4 memories [10].

Since HBM modes have a direct impact on memory operations, the performance of memory bound applications in KNL is directly affected by the chosen HBM mode. The KNL offers three different HBM modes:

- Cache mode;
- Flat mode;
- Hybrid mode.

In the *cache mode*, shown in Figure 10, the whole MCDRAM is used as cache. The memory address requests are first made to the MCDRAM, and if the result of the request is a miss, then the request is addressed to the DDR. However MCDRAM is a much wider memory compared to usual cache memories so the number of collisions is smaller due to cache direct mapping. The *cache mode* is commonly seen as the "standard" for most applications. However, if a low hit rate in MCDRAM is verified, flat or hybrid modes should be considered and can be configured in boot time.

The MCDRAM can also be configured as regular main memory (not only as cache) in the same systems address space. This is the configuration used by *Flat mode* and shown in Figure 11. The *Flat mode* is particularly more adequate for streaming applications, since the bandwidth of the embedded RAM (MCDRAM) is far better than external RAM, while its use as cache may impair performance due to high latency of the embedded stacked RAM.

The bandwidth in *flat mode* is higher compared to *cache mode* since it does not depend on the hit rates - specially for allocated memory datasets over 16 GiB. The reasons for that

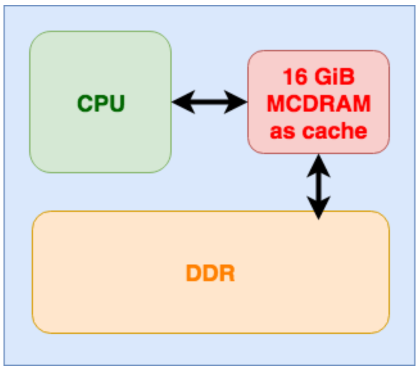


Figure 10.: Cache mode

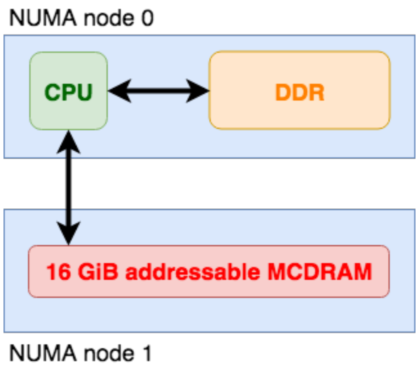


Figure 11.: Flat mode

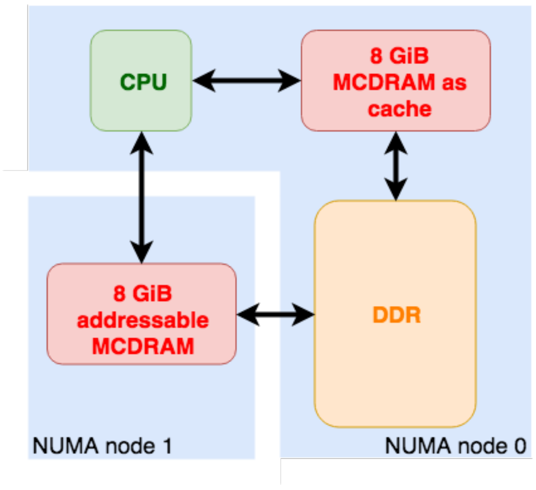


Figure 12.: Hybrid mode

is that accesses to memory objects do not have to query the on-package HBM in the first place [10]. The flat mode is the mode that offers more control over memory allocations to the developer. The developer can take the control over memory page allocations in MCDRAM and DDR memory at runtime (using `numactl`), or using *Memkind Library* for manual allocations in KNL HBM using C/C++ memory allocators injected in the code.

In this mode, MCDRAM is seen by the hardware as a separate NUMA node with no processor, and the processing units has its own internal addressable memory in the NUMA node where it belongs[3].

The *Hybrid mode*, shown in Figure 12, is the most flexible. This mode offers the possibility to combine the advantages of both *cache* and *flat* modes. The developers can therefore manipulate and configure the amount of HBM memory used as addressable memory, leaving the rest to be used as cache. This memory distribution can be configured at boot time. In

3 Source: https://colfaxresearch.com/knl-mcdram

practice, the cache portion acts as cache memory in *cache mode* and the addressable memory portion acts the same way as in *flat mode*. However in this configuration the processing units can perform a dual higher throughput to MCDRAM configured as addressable memory, and to MCDRAM used as cache simultaneously.

Figure 13 shows the theoretical peak floating point operations per Watt with double precision comparison between Intel Xeon multicore devices (in blue) and Intel Xeon Phi's co-processors (black). For a dense matrix-matrix multiplication algorithm, the Intel Xeon Phi's has 3 to 4 advantage in GFLOPS/Watt in comparison with multicore devices. The higher theoretical peak performance by many-core devices suggest a potential for a better performance compared with multicore devices.

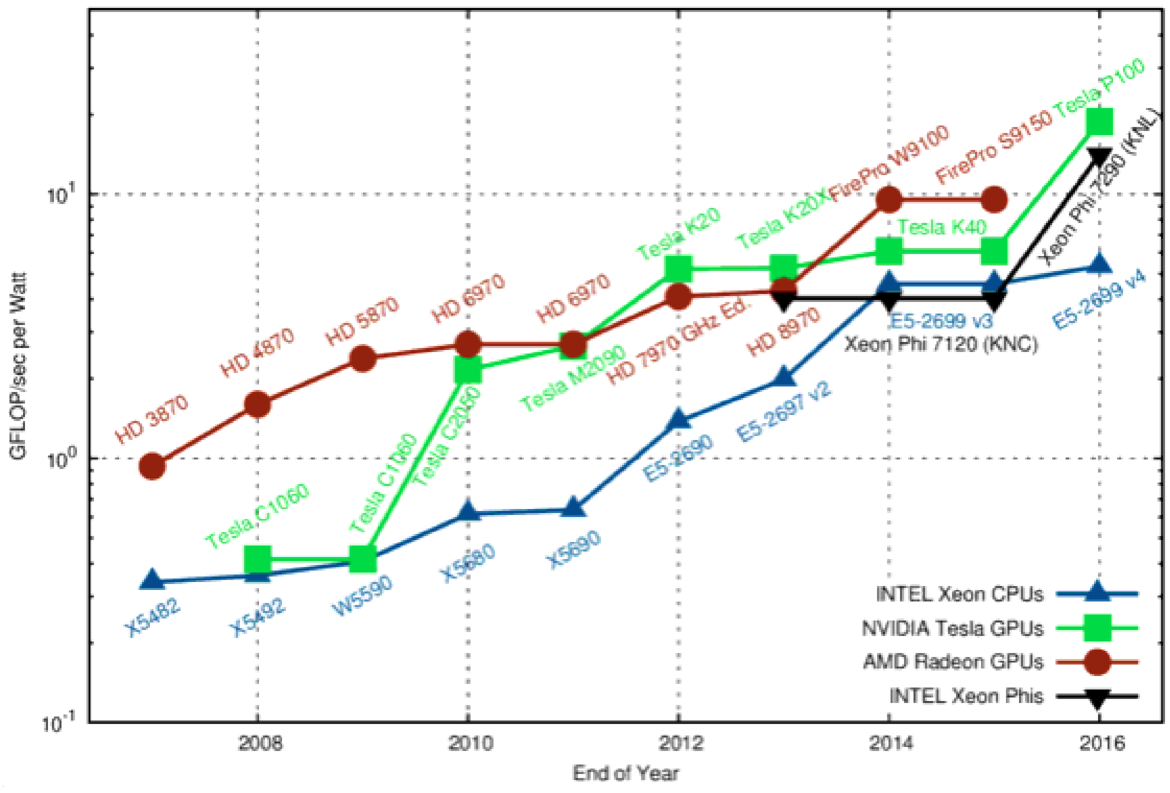


Figure 13.: Theoretical Peak Floating Point Operations per Watt: Double Precision[4]

## 3.2 Quantum ESPRESSO as a key tool

The Quantum ESPRESSO is an open-source software package, mostly implemented in Fortran 90. This software is used to calculate the electronic properties of materials at a nanoscale. The *PWscf* (a module of QE to explore DFT), is used to compute the self consistent field (SCF) using plane waves. The QE supports the utilization of two pseudopotentials classes: *ultrasoft pseudopotentials (USPP)* and *norm-conserving pseudopotentials (NCPP)*, both implemented in *PWscf*[5].

The self consistent field (SCF) is obtained in the *PWscf* through a modified Broyden method, with addition of some refinements. The reciprocal space region is defined by a set of $k$ points (wave vectors) in the Brillouin zone (BZ). A selection of a region with a greater number of $k$ points gradually improves the accuracy of calculations. The $k$ points can be distributed by different computer cores for parallel execution. The performance costs can be reduced by restricting the Brillouin zone (BZ) sample region that will be integrated [3].

Other software packages in the QE suite, besides *PWscf*, can be used in other study cases[6]. In this dissertation only the *PWscf* module and the $WSe_2$ material will be used as a main case study. The use of other modules of this software package is outside the scope of this dissertation.

The *Car-Parrinello (CP)* module, like *PWscf*, is a core package for calculations of the properties of the electronic structure of materials, through DFT and with the use of plane waves. This module implements the same functions of *PWscf*, with the exception of hybrid functionalities [2]. There are other modules beyond *PWscf* with other specifications not explored on this dissertation.

The *PHonon* module is used to calculate the vibration properties of materials, using functional density perturbation theory. The ballistic conductance is explored on *PWcond* module, *nudged elastic band (NEB)* calculations are made with *PWneb* module, and on *PostProc* have codes and utilities for post-processing data.

The *PWscf* already has support for parallelism, which was implemented in MPI for distributed memory, *OpenMP* for shared memory environments and hybrid parallelizations that include the simultaneous implementation of MPI with *OpenMP*. These are explored in the next section.

### 3.2.1 Quantum ESPRESSO *parallelism*

The software package Quantum ESPRESSO (QE), mainly developed in Fortran 90 and C, offers the possibility of exploring the hardware parallelism, through *application programming interfaces (APIs)*, *OpenMP* and libraries for *MPI* function calls. The application parallelism boils down to the distribution of tasks and data structures (e.g., vectors) by the available

5 Source: http://www.quantum-espresso.org/pseudopotentials/about/
6 According to user manual available in: http://web.mit.edu/espresso_v6.1/i386_linux26/qe-6.1/Doc/user_guide.pdf

cores, allowing the development of new functionalities and methods that guarantee the possibility of being executed in parallel.

Over the years, QE was developed with special focus on the execution of intensive computation simulations. In this calculation process, the computation effort involved is directly related to the number of wave functions used. Over the years, this package has been developed to increase the performance of these calculations on computer architectures with support for parallelism. The parallelism of this software is organized in a multi-level hierarchy. On the software documentation, four levels of parallelism are highlighted: *image, pool, plane waves* and *task groups* parallelizations [3].

The *image parallelization* is the process where tasks are distributed among images and grouped in sets (groups), assigned to different cores to be executed in parallel (used on *neb* calculations). On the second level (*Pool parallelization*), the cores are divided in sets (*pools*), where each one deals with one or more *k* points. The third level is implemented assigning the plane waves (PW) groups to the available cores in each *pool*. The last level of parallelism is the *task group* where the cores are divided into *task groups* according the formula:

$$n_{FFT} = \frac{n_{PW}}{n_{task}} \tag{1}$$

Each core handles different electron states and Fourier transforms, which are parallelized by each *task groups*. Alongside with the third level (*plane waves*), there is an additional level of parallelism that involves linear algebra calculations. Different libraries, like SCALAPACK, use alternative calculations and work distribution methods to obtain a higher level of parallelism and therefor increase the performance of the simulations. The input specifications (for instance, the number of *k* points, cut energies and the number of electrons) define the computational problem size [3].

In this dissertation, the simulations use a fixed number of *k* points. The parallelism approach is based on the *pool* parallelization level, with parameters choise set in the execution command (`-nk`).

Lately, Quantum ESPRESSO (QE) development focused specially in parallelism exploitation in multicore architectures. The latest versions of QE have support to *OpenMP* paralelizations and hybrid simulations. The main approach followed in this study is the development and analysis of hybrid, MPI and *OpenMP* tests and the main focus was on the optimization of the *PWscf* module execution environment for the case study referred in section 2.1.

## 3.3 installation and tuning of Quantum ESPRESSO

The Quantum ESPRESSO (QE) is a versatile multi-platform tool compatible with multiple operating systems, offering support to take advantage of multiple architectures. This support goes from multicore devices to the use of computing accelerators, such as *graphic processing units (GPUs)*. The package is distributed in a source mode with pre-compiled binary files for 32 and 64 bits based devices with Linux, Windows and MacOS. For specialized IBM and CRAY based hardware the tool has dedicated compilation and configuration rules. This broad level of support is followed by multiple installation possibilities: not restricted to the type of server hardware but to different parallel paradigms and libraries as well. Multiple performance libraries as BLAS, LAPACK and FFTW are available for different installations of QE: OpenMP or MPI only, OpenMP and MPI used in conjunction for hybrid environment codes.

This work explores the performance impact of using different types of installations alongside with some allowed refinements in the installation process, for performance tuning, to specific Intel's multicore and many-core devices. On the following sections, the installation process will be explored from the pre-defined installation to a customizable one, with the goal of improving performance.

### 3.3.1 *Requirements*

The installation of Quantum ESPRESSO (QE) has multiple solutions available: pre-compiled executables for Windows platforms, or directly from source code for Linux and MacOS. An Unix environment with *make* and other compilation utilities is needed, even to Windows installation with help of Cygwin for an Unix-like emulation. Alongside with the Unix environment, a C and Fortran 90/95/2003 compiler is also needed for libraries compilation and parallelism support. The parallelism is implemented with MPI libraries (OpenMPI or IntelMPI for example) and OpenMP.

### 3.3.2 *Configuration*

The Quantum ESPRESSO (QE) package has a configuration script to prepare the environment according to the system hardware, compilers and operating system. The package is supposed to work on most Linux 32-bit and 64-bit x-86 devices (all Intel and AMD CPUs) and multiple GPU-accelerated hardware. The following sequence will produce a parallel executable if a parallel environment is detected[7]:

7 Source QE user guide: http://web.mit.edu/espresso_v6.1/i386_linux26/qe-6.1/Doc/user_guide.pdf

```
cd quantum-espresso.X.Y.Z/
./configure
make all
```

After running the second line, multiple files are generated within the package, as shown in the figure below:

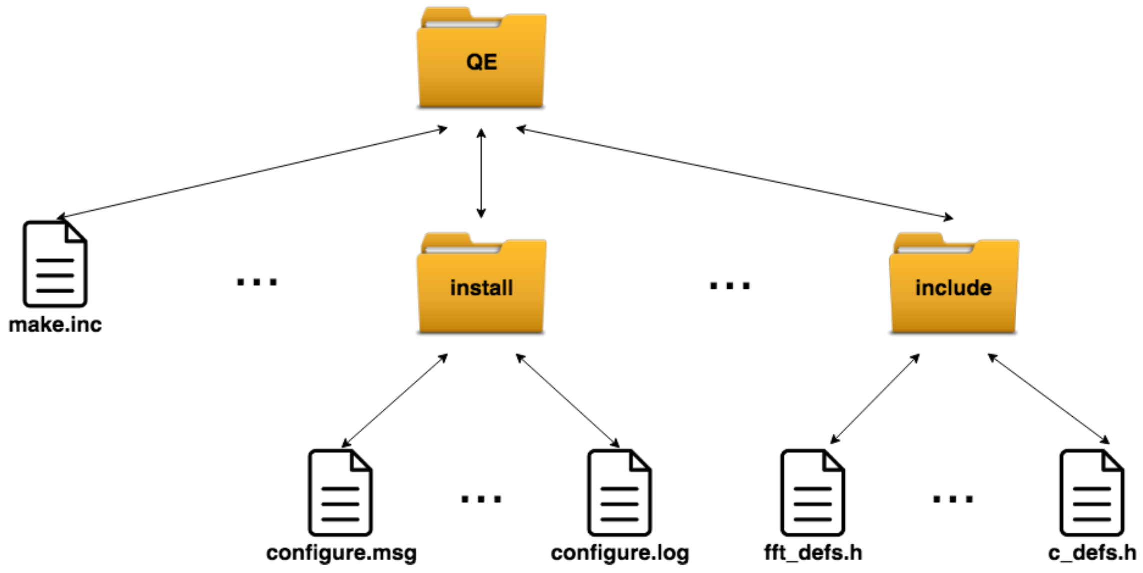


Figure 14.: Quantum ESPRESSO directory and generated files

- make.inc - is the file that contains all compilation rules and flags that will be used in the Makefile[7]; This file can be edited in order to change some performance tuning parameters, resulting in a custom installation, for tuning QE to a specific hardware platform;
- install/configure.msg - is a configuration report, not needed for compilation, but informs the user of all linked libraries so that he can check if everything is in place for installation;
- install/config.log - a detailed log with configuration made for installation[7];
- include/fft defs.h and include/c defs.h - Fortran 90 and C definitions used by Fortran 90 and C files[7].

After running the *configure* command, if the processing device is unknown, the ARCH variable needs to be specified with a set of values, depending on the architecture of the system:

```
./configure ARCH=...
```

| ARCH= | Description |
|---|---|
| IA32 | Intel 32 bits instruction set architecture |
| IA64 | Intel 64 bits instruction set architecture |
| x86_64 | x86 instruction set of 64 bits |
| AIX | A series of Unix operative system developed by IBM |
| Solaris | An Unix operative system developed by SUN Microsystems |
| SPARC | Stands for Scalable Processor Architecture. Is an instruction set with a reduced number of instructions (RISC) |
| Cray XT4 | A massively parallel MIMD supercomputer with distributed memory developed by Cray Inc |
| cygwin | A Unix environment simulator with command line interface used on Windows operative systems |
| mingw32 | A development open-source software used to create 32 bit Windows applications |
| mingw64 | A development open-source software used to create 64 bit Windows applications |
| NEC SX | A SX vector supercomputers made by NEC. Is one of the most advanced vector supercomputers |
| ppc64 | An identifier frequently used on Linux, GNU Compiler Collection (GCC) and LLVM. This identifier is used to refer the target architecture for optimized applications for 64-bit PowerPC and Power Architectures. |
| arm | Extends to Advanced RISC Machine, is a family of RISC architecture for computer processors. |

Table 1.: Available architectures in Quantum ESPRESSO

### 3.3.3 *Optimized libraries*

The Quantum ESPRESSO (QE) installation can be customized to introduce some optimizations and to enable parallelism. The default installation presented before, will automatically enable MPI executable if a parallel architecture platform is detected. An additional switch enables OpenMP (using `--enable-openmp` in the configuration step). The QE package includes the following optimized libraries for compilation:

- BLAS - an internal BLAS library available with the switch `--with-internal-blas`; however, users can define their own versions of BLAS in the configuration step;

- LAPACK - the Fortran 90 optimized routines to solve linear equations using matrix factorization as Cholesky, LU, Schur or SVD;

- FFTW - FFTW libraries for discrete Fourier transform calculation (DFT), using one or multiple dimensions.

Another optimized and relevant library is the Intel *Math Kernel Library (MKL)*. This is not a self-contained library in QE but highly recommended since simulations can benefit significantly with performance gains in Intel's devices. If MKL is available on the system, configure command will use this installation[7] and the same goes for ACML to take advantage of AMD devices. The MKL libraries can also be used in AMD devices with optimized libraries, but with reduced performance levels compared with Intel devices. The *configure* command included in QE can be used to recognize the MKL libraries installed in the system but the switch `--with-openmp` should be specified in the configuration phase, otherwise, a single threaded version of MKL will be linked. The FFTW can be used with MKL to combine the benefits of multithreading and distributed memory execution.

### 3.3.4 *FFTW vs. FFTW3*

Since FFTW3 library is supported in Quantum ESPRESSO (QE), multithreading is available alongside with hybrid parallelism, when combined with MPI. The FFTW3 library supports multi-threading in two modes:

- implicit mode: the routines of this library can be executed with internal multitreading, being called like serial code (requires installation of the library libfftw3_omp);
- explicit mode: this mode uses serial routines that are called by multiple parallel threads; the routines for FFT execution are thread-safe.

The FFTW3 library, with only one routine call, makes calculations of many transforms. This allows performance and flexibility, within multiple platforms.

In the performance test of the threaded FFTW3 library, the FFTW library was chosen for comparison (this library is internally supplied with QE) since it is the only one that supports threading in the hybrid mode. It is also an open-source library and widely available [12].

The experimental measurements in Figure 15 show a performance test of the FFTW library with multiple threads. This test was made on multiple dual quad-core servers, interconnected with Gigabit Ethernet. The code was compiled with icc compiler (v.11.1), with `-O3` flag, and ifortran for compilation of QE [12].

The FFTW3 hybrid library was tested with *PWscf* module of QE. The measured execution times were made by increasing the number of processing devices - 2 and 4 threads per MPI process - and comparing with pure MPI. The total number of computing cores is equal to the number of MPI processes times the number of threads per MPI process.

Figure 15 shows that the FFTW3 (implicit and explicit) with 2 threads presents the fastest execution times when compared with the FFTW internal with 2 threads. The execution times of implicit and explicit FFTW3 with 2 threads are very close to each other and the same happens with 4 threads. However, the FFTW3 with pure MPI displays the best results and for the implicit and explicit FFTW3 with pure MPI the execution times are always faster when compared with FFTW internal.

These results led to the selection of the FFTW3 library on the second installation of QE. Depending on the test case, FFTW3 seems to be a good performance library for pure distributed algorithms using QE.

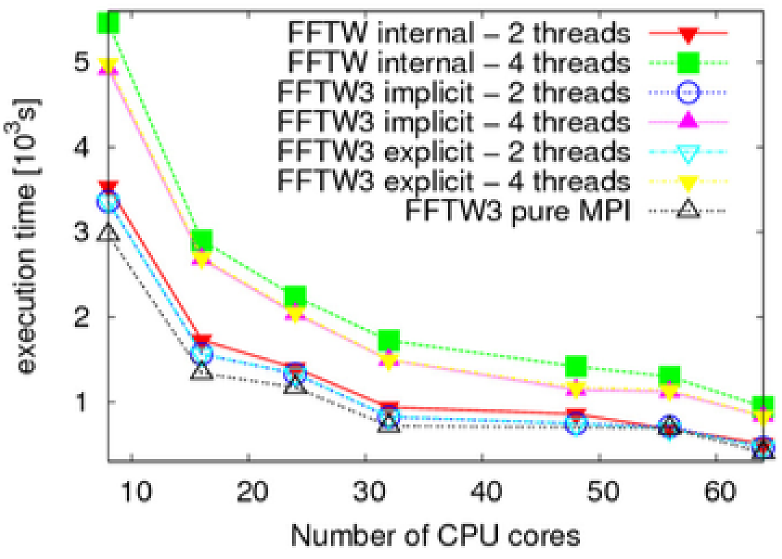


Figure 15.: "PWscf module execution using QE FFTW3 multithreaded version compared with FFTW hybrid implementation and pure MPI FFTW3 implementation. Performance evaluation based on execution times of QE FFTW3 pure MPI, and FFTW3 implicit and explicit version" [12]

There are other approaches to scale up QE simulations, taking the maximum advantage of modern architectures: some of them based on offloading CPU-bound workload to a co-processor with a higher number of cores than a traditional multicore device. The Intel Xeon Phi is an example of these many-core devices, supporting different ways of workload offloading, as presented in the next section.

### 3.3.5 Quantum ESPRESSO *compilation on a co-processor Intel Xeon Phi*

Quantum ESPRESSO (QE) can be installed on a Intel Xeon multicore device with a many-core Xeon Phi co-processor, with a higher number of cores, in order to take advantage of offloading workload. There are three ways of compiling QE on an Intel Xeon Phi.

- *native mode*: the workload is completely offloaded and executed on the co-processor[7];
- *offload* mode: the execution flow starts on the main multicore device but higher groups of workload can be automatically offloaded to the co-processor in a transparent fashion to the programmer[7];
- *symmetric* mode: requires the creation of both binaries (not well explored)[7].

The offload mode requires the libxphi library, which is compiled and dynamically linked to QE. It allows offloading BLAS and the MKL library functions into the Xeon Phi to hide communication latency costs[7]. The switch `-mmic` should be activated during compilation.

### 3.3.6 *A multi-node installation with MPI*

Current computer clusters with multiple shared memory servers run some version of MPI and the QE works with MPI implementations taking advantage of an hybrid environment of shared and distributed memory. The *configure* command will recognize a properly installed parallel environment and prepare the parallel compilation.

In a Linux cluster with MPI, the *configure* command can take multiple behaviours. It tries to locate a parallel compiler in a logical place with a logical name, but if it has a odd name or it is located in a odd location, the programmer has to instruct the configuration script where to find it[7].

On the other hand, the *configure* can try to locate mathematical and parallel libraries in the usual places with usual names, but if they have unknown names or odd locations, the programmer has to rename/move them, or instruct the *configure* where to find them. If MPI libraries are not found, parallel compilation is disabled[7]. The *configure* can also revert to serial compilation if libraries cannot be linked without conflicts and missing symbols.

## 3.4 CHALLENGES TO IMPROVE THE COMPUTATIONAL EFFICIENCY

QUANTUM ESPRESSO (QE) already uses parallelization (MPI and OpenMP) so one of the challenges was to improve its computational efficiency, namely selecting the most adequate parameters for execution, specially with very large data sets. The QE implements several MPI parallelization levels based on the selection of parameters. These parameters combine different group configurations among cores, responsible to process one or more groups of $k$ points. These parameters assume different designations, namely:

- *images*, which divides the number of cores into different "images", where each group of images corresponds to a different self-consistent or linear-response calculation;
- *world*, which establishes one group containing all cores;
- *pools*, each image is placed in pools, where each one is responsible for a group of $k$ points; this was the main parameter used in this study;
- *bands*, where each pool can be placed in a group of bands, each band group taking care of a group of wave-functions;

- *tasks*, which can be organized in multiple groups to process multiple wave-functions in parallel; this organization usually occurs when the number of cores exceeds the number of FFT planes to allow a higher parallelization level[7].

There are multiple flags to control the number of cores in each group:

- `-nimage`, to change the number of images (`-ni`);
- `-npools`, for the number of pools (`-nk`);
- `-nband`, to handle the number of bands (`-nb`);
- `-ntg`, to modify the number of task groups (`-nt`).

These are the main parameters that define the bounds of parallelization. The $k$ points are divided among different pools and mapped into the different cores to be processed in parallel. The $k$ points distribution among cores are made in a transparent way by the *PWscf* package of QE. The $k$ point parallelization is limited to Nk processor pools (Nk, parameter configured with `-nk` switch). The chosen number of pools was 1, 2, or 4. The FFT parallelization should not exceed N3 processors, where N3 is the dimension of the FFT grid along the Z axis. For the simulations performed with `-nk` switch, the number of MPI processes (N) should be $N = Nk \times N3$ MPI processes at most, where N is the value passed to *mpirun* command for the `-np` parameter [8].

The key to obtain good performance values using *PWscf* is to achieve the best compromise between the parameters used in the described control flags. The goal is to achieve a good load balancing among MPI processes. The number of k-point pools should be an integer divisor of Nk (pools). The number of processors for FFT parallelization should be an integer divisor of N3. These parameters can be used together if needed, as shown in the following example:

```
mpirun -np 4096 ./neb.x -ni 8 -nk 2 -nt 4 -i my.input
```

Some of these parameters (i.e. `-ni`) are used for specific modules, such as NEB. The images are properties of the NEB module input.

In the above example, the NEB module of QE, for the input file `my.input`, runs on 4096 cores, divided into 8 images groups, each one with 512 cores, 2 pools of $k$ points with 256 cores each, and 4 task groups, each one with 64 cores. If these parameters are not set, the default value used for `-ni`, `-nk` and `-nt` is 1[7].

8 Source, *PWscf* user guide: http://www.afs.enea.it/software/qu_esp/Doc/pw_6.1.0.pdf

The configuration of these parameters is the main challenge: finding the perfect balance between the possible values is crucial in order to obtain the best parallelization of the QE and therefor the best possible performance.

Another challenge regarding the parallelization is the performance of the QE when all the available cores are used which may lead to a better performance value. If more than one node of the cluster is used with MPI, it is necessary to be aware of the communications overhead between nodes, as well as the best *k* points configuration for a good distribution by the pools.

The size and the data type of the linear algebra system of the case study has an impact to the effectiveness of parallelization, together with the inter-process communication time. The communication between nodes should be performed using Gigabit Ethernet up to 4 or 8 sockets to reduce the communication latency. The wave functions data represented in the algebra system should be kept in cache for as long as possible.

The size of the pool can also affect the performance. Depending on the case study and its size, pools should be divided into multiple *task groups*. This pool distribution should be performed when the number of processors exceeds the number of FFT planes. The advantage of this approach is that each *task group* can simultaneously process multiple wave functions[9].

The next chapter presents a comprehensive performance evaluation between different installations and configurations of QE in order to find out the best use of the hardware resources.

9 Source, *PWscf* user guide: http://www.afs.enea.it/software/qu_esp/Doc/pw_6.1.0.pdf

# 4 EXPERIMENTAL EVALUATION

A set of performance tests with multiple configurations and installations of QE are presented in this chapter, as the experimental work of this dissertation, in order to characterize the fastest execution environment of $WSe_2$ simulation in QE. These tests are based on multithreaded implementations and distribution of processes among the available processing units. The goal is to measure the impact of using faster solver libraries, hybrid installations, $k$ points distribution among processing pools and the impact of other configurations.

In a first phase, the computational platforms used to explore these tests are based on servers with x86 multicore devices. A comprehensive study of these platforms and their potential was made for a better decision process related to the workload distribution configuration for fine tuning the performance.

Once the best environment execution for the simulation using multicore devices is found, the same group of tests was applied on a many-core based server. The first goal for this study was to aid a physicist of University of Minho to improve the time results for his $WSe_2$ simulations in QE. The *reference time (RT)* for the comparative evaluation used the same QE version and was scheduled for 20 processes on a dual 10-core Xeon server. The value of the RT achieved by the physicist was 2.60 minutes. The second goal was to consider the best execution times achieved with the multicore devices under different configurations and measure the effective performance improvements of the many-core server with the simulation of $WSe_2$ in QE.

The study of both computational platforms includes an architecture overview over their cores micro-architecture, available memory and functional units in order to fine tune the workload distribution. In the many-core based server, multiple configuration "modes" are supported, with different core grid organizations and memory access layouts. For the second goal, the impact of these possible configuration modes was taken into account to compare the measured performance value with the measured values on the multicore server.

## 4.1 EXPERIMENTAL SETUP

This chapter presents and discusses the measured execution times, using multiple compute nodes (servers) in the ***S**ervices and **A**dvanced **R**esearch **C**omputing with **H**TC/**H**PC (SeARCH)* cluster - a research computing platform composed by multiple interconnected nodes.

The compute nodes of the cluster are equipped with multiple multicore and many-core architectures. The cluster gives support to the University of Minho R&D projects, in all fields of science and engineering. The current architecture of the SeARCH cluster is equipped with 2 nodes for the front-end, 54 computing nodes with dual-socket Intel Xeon 64-bit devices and one node with a many-core Xeon Phi Knights Landing (KNL). Other 6 nodes provide fast direct access to a SAN with about 40 TiB. The cluster also contains 12 nodes with accelerators based on Nvidia GPUs and 9 nodes with the Intel Xeon Phi co-processor. The inter-node communication is performed by Gb Ethernet and some nodes are also equipped with 10 Gb Myrinet cards. *Linux CentOS* is used as the operating system of the nodes and the cluster management[1] is done by *Rocks Cluster Distribution*.

Two computing nodes were used for the exploratory study, with the same Intel Xeon device generation (Ivy Bridge), but with different number of cores (8 and 10). The following subsections present a full characterization and architecture analysis of these compute nodes, followed by a description of the input data set: a mathematical representation of an inorganic compound tungsten diselenide.

### 4.1.1 *Characterization of the multicore environment*

In a first phase of the experimental work, two different Intel Ivy Bridge servers were used: a dual 8-core server and a dual 10-core server (Figures 16 and 17 respectively), both with the Sandy Bridge architecture. These are the multicore servers that were used to measure the execution times for the different test cases, including the results achieved with the tuned installations.

In both devices, each core has 64 KiB L1 cache - 32 KiB for instructions and 32 KiB for data - and 256 KiB L2 cache. The size of the L3 cache is coherent with Intel's approach, 2.5 MiB per core: 20 MiB in the 8-core device, and 25 MiB in the 10-core device. The devices also have a slight different clock frequency: 2.6 GHz for the 8-core device and 2.5 GHz for the 10-core. The $WSe_2$ crystal dataset is the same for all tests, which makes it possible to carry out a comparative performance study. The dataset, described in section 4.1.4, has about 6.75 GiB in size, so it does not fit entirely in any of the cache levels of the devices used.

1 SeARCH DI Uminho: `http://search6.di.uminho.pt/wordpress/?page_id=43`

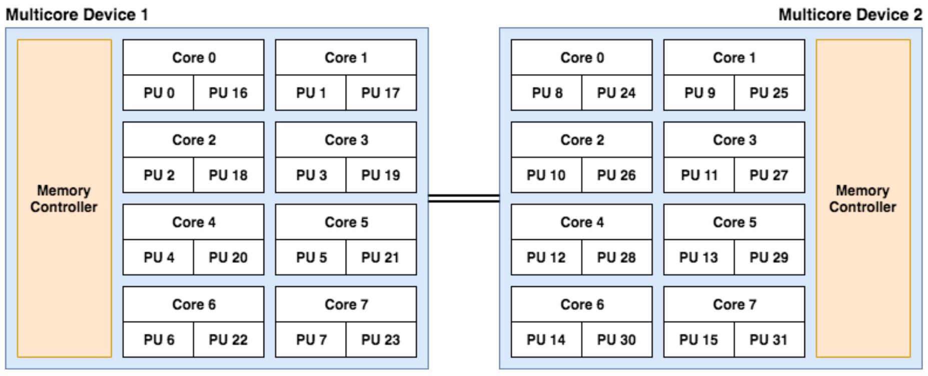


Figure 16.: SeARCH dual 8-core server overview

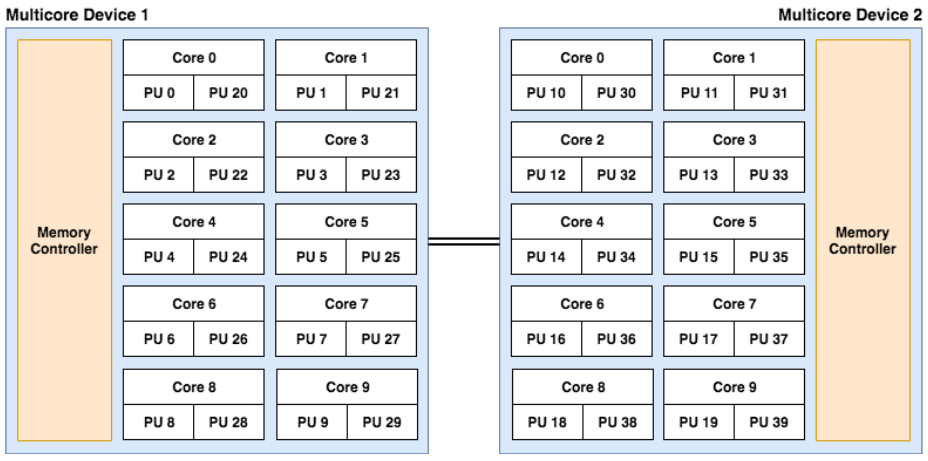


Figure 17.: SeARCH dual 10-core server overview

The table 2 shows the main differences between both servers:

| **Node** | dual 8-core server | dual 10-core server |
|---|---|---|
| **Devices** | Intel Xeon E5-2650 v2 | Intel Xeon E5-2670 v2 |
| **#Sockets** | 2 | 2 |
| **Micro-Architecture** | Sandy Bridge | Sandy Bridge |
| **#Cores per Socket** | 8 | 10 |
| **#Virtual Cores per Socket** | 16 | 20 |
| **Clock Frequency** | 2.6 - 3.4 GHz (Max) | 2.5 - 3.3 GHz (Max) |
| **SIMD** | AVX | AVX |
| **Cache Level 1 (L1)** | 32 + 32 KiB per Core | 32 + 32 KiB per Core |
| **Cache Level 2 (L2)** | 256 KiB per Core | 256 KiB per Core |
| **Cache Level 3 (L3)** | 20 MiB | 25 MiB |
| **#Memory Channels** | 4 | 4 |
| **Memory RAM** | 64 GiB per Node | 64 GiB per Node |
| **Memory Bandwidth** | 59.7 GiB/s | 59.7 GiB/s |

Table 2.: Specifications for the dual 8-core and dual 10-core servers on SeARCH

A set of multiple performance tests were made for both nodes with Ivy Bridge using the same dataset. All tests were made running the simulation software only with a minimum of intrusion level as possible. As stated in Table 2, the compute nodes used for this study have a variable CPU frequency from a 2.6 GHz to 3.4 GHz (dual 8-core server) and 2.5 GHz to 3.3 GHz (dual 10-core server). To prevent execution time variations (which could lead to inaccurate results), the clock frequency was fixed to 2.6 GHz and 2.5 GHz in the dual 8-core server and dual 10-core server respectively. Each of the walltimes documented in section 4.3 is the weighted result of the 5-best executions.

### 4.1.2 *Characterization of the many-core environment*

Over the last decades, processor manufacturers released processors with a higher number of cores and lower clock frequency as opposed to the increase of the internal clock frequency. The many-core architectures have emerged in the last ten years with a considerable higher core density in the chip compared to multicore architectures.

The first group of tests running $WSe_2$ simulation was made in the compute nodes using a multicore architecture. The second group of tests was made in a many-core architecture, available in the SeARCH cluster, to study the possibility of getting an higher level of parallel work and better performance results compared with the results already achieved in the multicore tests. To perform these tests, an Intel Xeon Phi Knights Landing (KNL) many-core device was used, replicating the same test cases, for the same case study.

The goal with these tests is to find out if many-core devices such as, the Intel Xeon Phi KNL can be considered a good target devices for performance running this type of simulations using QE. The Intel Xeon KNL used for the second group of tests is similar to the processor showed in Figure 6 of section 3.1.

The many-core device in the SeARCH compute node, equipped with an Intel Xeon KNL, has 32 dual-core tiles (64 cores) interconnected by a 2D mesh of bi-directional rings. Each tile has 2 cores, each with 2 VPU, and 1 MiB of L2 cache. This device has no on-chip L3 cache. Regarding RAM memory, KNL has 8x 2 GiB MCDRAM on-package (3D stacked chips), which can be configured as L3 cache or RAM. An high bandwidth of 8 DDR4 channels can connect up to 384 GiB of RAM. In terms of I/O, this device has 36 lanes of PCIe Gen3 and 4 lanes of DMI per chipset [10].

In terms of performance, for scalar operations, this chip is theoretically 3x faster than the Xeon Phi Knights Corner. For vector operations each core in this chip has two vector units each with 512 bits as mentioned before, with the potential to perform 32 single precision operations per clock cycle and 16 double precision operations per clock cycle (it supports Fused Multiply-Accumulate, FMA), at each core.

The hardware specifications for this chip (Table 3) also shows that Knights Landing introduces two AVX-512 units (SIMD technology for vector instructions). These vector units are specially suitable for compilers with good vectorization heuristics, such as the compilers for *High-level language (HLL)* (C/C++ and Fortran) [10]. Since QE is written in these languages, the installations made for KNL tests were made using appropriate flags for vectorization during the package compilation.

| **Device** | Intel Xeon Phi 7210 |
|---|---|
| **#Sockets** | 1 |
| **Micro-Architecture** | Knights Landing |
| **#Tiles** | 32 |
| **#Cores** | 64 (2 per Tile) |
| **#Virtual Cores** | 256 |
| **Clock Frequency** | 1.3 - 1.5 GHz (Max) |
| **SIMD** | 2 x AVX-512 per core |
| **Cache Level 1 (L1)** | 32 + 32 KiB per Core |
| **Cache Level 2 (L2)** | 1 MiB per Tile |
| **Memory RAM** | 384 GiB |
| **Embedded RAM** | 8 x 2 GiB |
| **#Memory Channels** | 8 |

Table 3.: Intel KNL node specifications

This many-core device, installed on the SeARCH cluster, offers the possibility to configure the cluster modes (all to all, SNC4 and quadrant mode) as well as the HBM modes (flat and cache mode). These multiple modes configuration were previously described in sections 3.1.1 and 3.1.2. For a dataset below 16 GiB, the Flat mode revealed to be an important HBM mode to tune the performance of $WSe_2$ simulation in QE. This tuning was achieved through memory allocation manipulations using `numactl`.

The multiple possibilities of memory allocations suggest a comparative performance test between different HBM configurations. The performance results using these configurations (cluster and HBM modes) will be explored in section 4.4.

### 4.1.3 *Software and libraries versions*

Table 4 shows the version of the software packages (QE, compilers, libraries and solvers) that were used on the experimental work of this dissertation. The versions of QE and ELPA were the most recent versions at the moment of the study. When the several installations were made for the experimental work, the versions of the compilers were also the most recent (installed in the SeARCH cluster) at that time.

| | Versions | | | |
|---|---|---|---|---|
| | Without ELPA | | With ELPA | |
| **Description** | **Multicore** | **Many-core** | **Multicore** | **Many-core** |
| Quantum ESPRESSO | 6.1 | | | |
| Intel Compilers and Libraries | 2013.1.117 | 2017.1.132 | 2017.4.196 | 2017.3.191 |
| Intel OpenMPI MX | 1.8.2 | N/A | | |
| ELPA Solver | N/A | | 2016.11.001 | |

Table 4.: Software packages versions

### 4.1.4 *Input characterization*

The input files for *PWscf* module are described as a structured list of Fortran 90 constructors called NAMELISTS and specific QE codes named INPUT_CARDS. Among the several NAMELISTS available, there are three mandatory to use in *PWscf* (also included in the input file of the present case study): &CONTROL, &SYSTEM and &ELECTRONS.

The &CONTROL namelist defines a group of variables responsible for flux control of the computation. For example, the specification of the task to be performed (SCF defined by default), a title to define the input, verbosity level, input and output files directories, among others.

The &SYSTEM namelist specifies the properties related to the material under study, which can be the number of atoms in the unit cell, the number of electronic states to be computed, the kinetic energy cutoff for wavefunctions, or the total charge of the system, among others.

The control variables used in the algorithms to achieve self-consistent solutions are defined in the &ELECTRONS namelist. This last namelist is specially important to define the variables to achieve the solutions of Kohn–Sham equations for the electrons (already explained in chapter 2). These variables includes the maximum number of SCF step iterations, the diagonalization approach (Davidson approach or conjugate-gradient).

Regarding the QE NAMELISTS, *PWscf* can process other additional NAMELISTS that defines moving properties. The &IONS namelist can be used as an input variable that defines the movement of ions in molecular dynamics, as long as &CELL namelist that defines cell-shape evolution whenever a cell moves. The &EE namelist can be used to perform charge corrections when the problems have boundary conditions.

These movement simulation namelists and &EE are not used in this thesis since the study of $WSe_2$, with the input file used, does not have any charge corrections. To define the relative position of each atom in the cell, INPUT_CARDS are used. This code also allows to specify the name, the type and the mass of multiple species of atoms. The existing INPUT_CARDS are:

- CELL_PARAMETERS: contains the vectors that defines the cell;
- ATOMIC_SPECIES: defines the mass, name and pseudo-potential used for each atomic species in the input file;
- ATOMIC_POSITIONS: defines the type and the relative position of each atom in the cell (represented with three dimensional coordinates);
- K_POINTS: defines the coordinates and weights of the $k$ points.

The input case study of $WSe_2$ uses mostly NAMELISTS and INPUT_CARDS, as described before (except for &IONS, &CELL and &EE). The complete $WSe_2$ definition input file is in appendix B. Table 5 shows the most relevant crystal properties defined in the input file, namely the number of defined $k$ points (144, $12 \times 12 \times 1$, where only 78 were used after symmetries removal), the kinetic energy cutoff in the wave functions (*ecutwfc*), the number of electronic states (*nbnd*) and the number of G-Vectors.

| **Properties** | **Value** |
|---|---|
| K Points | 78 |
| Electrons | 18 |
| ecutwfc | 50 eV |
| nbnd | 23 |
| G-Vectors | 75113 |

Table 5.: Most relevant properties of the case study ($WSe_2$)

The memory consumption directly depends on the number of G-Vectors, the number of electrons and the number of $k$ points. These $k$ points can be divided into a set of pools (represented by QE `-nk` parameter). A pool is the designation used in QE for a group of processing cores by which the $k$ points are distributed.

To run a simulation using a dataset, an input file is used with self consistent field configuration values for these parameters. The dataset size, used in memory, can be computed by the following expression, where the size of a complex number is 2 doubles: one for the real part and the other for the imaginary part:

$$(\#GVectors \times Electrons \times 4 \times kpoints)^2 \times SizeOfComplexNumber \qquad (2)$$

The case study considered 18 electrons, 75,113 G-Vectors, and 78 $k$ points, which led to a dataset size of approximately 6.75 GiB. The size of the output file is also in this order of magnitude. The key performance metric for all tests is the wall time of QE simulation [2].

2 Source, *PWscf* user guide: `http://www.afs.enea.it/software/qu_esp/Doc/pw_6.1.0.pdf`

## 4.2 SELECTING ADEQUATE INSTALLATIONS

One of the major advantages of QE is its integrated suite with support for exploring electronic structure materials in different parallel paradigms. Different paradigms suggest different strategies to split data between processing units, or even different compute nodes to increase the scalability. QE can be installed to explore electronic structure materials for codes using message passing interface (MPI) (mainly for wave-functions distribution across multiple computes nodes), OpenMP for shared memory environment, or hybrid environment combining the advantages of shared memory parallelism and data distribution across multiple nodes. By default, the parallelism in QE is explored using performance libraries as SCALAPACK.

The efficiency of the computations in performance libraries such as SCALAPACK is granted by a block cyclic data distribution, or block-partitioned algorithms for matrix data reuse. When programmers make a standard installation of QE (as presented in section 3.3.2), SCALAPACK is installed by default. However, as it will be demonstrated in this study, the adoption of alternative libraries allows to push up toward the performance in cases such as eigensystem calculations, using solvers to overcome potential computational bottlenecks. Considering this, the present study starts by exploring the default installation that uses SCALAPACK as the default performance library in multiple parallel paradigms.

The ELPA is used as an alternative direct solver to understand the influence of the solver in terms of performance. This library is mentioned in some studies, like quantum chemistry, biological networks and materials science, as one of the best libraries for massively parallel executions [8]. The installations in Intel KNL allows to verify the impact of using ELPA in an alternative many-core architecture.

### 4.2.1 *Hybrid installation*

The installation process of QUANTUM ESPRESSO (QE) supports multiple configurations depending on the computing platform used to run the simulations. These can run in shared memory environments (SMP or NUMA servers) as well as in multiple nodes in distributed memory environments, with high bandwidth interconnects. When no parameters for a specific architecture are configured in the installation process, QE is installed with a basic multiparadigm support. This type of installation cannot take full advantage of partitioning input strategies or data distribution across nodes, limiting the performance.

*Quantum ESPRESSO installation on SeARCH*

Nowadays, is common to see Hybrid applications in HPC infrastructures to improve performance. These improvements can be achieved to combining OpenMP and MPI implementations. The main advantage of these hybrid environments is the reduction of memory required for the application in general. The QE provides an hybrid package for hybrid simulations that was used in this study.

```
# Load modules
module load intel/2013.1.117
module load intel/openmpi_mx/1.8.2

# Command configure
./configure --enable-openmp

# Installation
make pw
```

Listing 4.1: Default installation on a multicore node

This first installation was used as a reference to be compared with other installations that take advantage of performance tuning configurations for the compute node and KNL (addressed in section 4.2.1).

```
# Load modules
module load intel/2013.1.117
module load intel/openmpi_mx/1.8.2

# Command configure
./configure --enable-openmp --enable-parallel

# Changes on make.inc
DFLAGS = -D__OPENMP -D__INTEL -D__DFTI -D__MPI -D__SCALAPACK -D__FFTW3
LAPACK_LIBS = -lmkl_blacs_intelmpi_lp64

# Installation
make pw
```

Listing 4.2: Second installation on the SeARCH cluster

In this installation, the hybrid execution counts with a clean package installation using FFTW3 library through direct linkage without any additional improvements. As said before, in section 3.3.4, this library should be considered when multithreaded or MPI simulations are used to process multiple transforms in parallel.

This installation includes the addiction of LAPACK library with direct linkage enabled by Intel MPI module for a architecture optimized installation. The LAPACK was chosen as the performance library instead of ATLAS, but it is not a fully replacement as described in QE manual[7]. As in the previous installation, the same modules for ICC and MPI were used.

*Quantum Expresso installation on a many-core device*

Since the first installations, all the tests were made using multicore nodes. In this section there is a new approach of testing Quantum ESPRESSO in a computer architecture equipped with a many-core device: the Intel Xeon Phi Knights Landing.

This architecture provides a next level scaling performance from QE taking advantage of their high core count and hardware threading architecture. As opposed to Intel Xeon processors, KNL has a higher number of cores (64 cores), 4 SMT threads (Simultaneous multithreading), and wider AVX instructions (two units of AVX-512 bit). Regarding the memory architecture, the KNL embedded in-package RAM chips have three different memory modes: flat, cache and hybrid. These systems level configurations of the physical addressable space, allows for a better use of the systems MCDRAM chips, to reduce memory accesses.

The following listing shows the configurations made on the installation process of QE in the compute node of the SeARCH cluster that contains this many-core processor.

```
# Load modules
source /opt/intel/compilers_and_libraries_2017.1.132/linux/bin/compilervars.sh
    intel64
source /opt/intel/mkl/bin/mklvars.sh intel64
source /opt/intel/impi/2017.1.132/bin64/mpivars.sh

# Command configure
./configure CC=mpiicc MPIF90=mpiifort F90=mpiifort F77=mpiifort CPP="icc -E"
FC=mpiifort LIBDIRS="/share/apps/fftw/3.4.0/include/fftw3" --enable-openmp --
    enable-parallel

# Changes on make.ink
DFLAGS = -D__OPENMP -D__INTEL -D__PARA -D__MPI -D__SCALAPACK -D_FFTW3
LAPACK_LIBS = -lmkl_blacs_intelmpi_lp64
SCALAPACK_LIBS = -lmkl_scalapack_lp64 -lmkl_blacs_intelmpi_lp64

# Installation
make pw
```

Listing 4.3: First Installation on KNL

The configuration of the MKL, ICC and MPI compilers were made in the node as the starting point. After that, the makefile of *PWscf* module was adjusted with some minor changes to ensure the use of the 64-bit LAPACK and SCALAPACK libraries. This installation tests differ from the tests done so far, since they are made directly on the node without a batching process and neither a frontend to schedule the tests.

Until now, the main key to improve performance was the adoption of performance libraries such as SCALAPACK and FFTW3. These libraries were compiled to ensure a tuned installation of QE in Intel Xeon based compute nodes and in KNL. The core values of SCALAPACK include efficiency (run the algorithms as fast as possible), scalability (improving performance when increasing the problem size and number of used cores) and portability, since SCALAPACK is compatible with the main parallel processor architectures.

Besides the usage of previous performance libraries, many studies also recommend other approaches for performance tuning, namely the use of switches that enable AVX instructions as well as the ELPA library. These approaches were also explored in this study to measure their performance impact. In the following section the ELPA library is presented.

### 4.2.2 *Using ELPA solver for performance tuning*

The acronym ELPA stands for Eigenvalue soLvers for Petaflop Applications. It is a parallel library for scalable solutions involving multidimensional matrices and vectors. In this study the ELPA library is used to solve a set of equations, which is the highest expensive operation in QE. In electronic structure theory, as well as in other computational sciences, the processing weight of these operations increases as the size of the problem increases.

The ELPA library has proven to be a good Eigenvalue solver replacement over SCALAPACK. The solver can be compared with the SCALAPACK library but uses their own subroutines in the parallel solution steps. The library can be configured for MKL, ICC and Intel MPI used in QE and provides an efficient algebraic solution for the symmetric Eigenvalue and Hermitian problems of dense matrices that have real and complex values. The key performance of the ELPA solver is its subroutines, that are a constituint part of its parallel solution steps. The solver can even outperform the SCALAPACK library that implements the interface for SCALAPACK, such as the known MKL from Intel. The critical zone in terms of performance is the matrix reduction to a tridiagonal form and their back-transformation of eigenvectors. The solver provides two different tri-diagonalization methods: the first one, only uses Householder transformations and the second one is the two-step transformation which is more efficient for larger matrices and for a higher number of processing units.

The library is based on MPI and hybrid MPI/OpenMP implementation that provides a singular step for tri-diagonalization and two different steps for matrix transformations. If the size of the matrices is high enough, as the number of CPU's increases, the efficiency of these steps also tends to increase [8].

The task of finding eigenvectors and eigenvalues for matrices with higher dimensions is a common problem in computer science, specially in electronic structure theory. Remembering Kohn-Sham theory presented in 1, the electron problem and the self-consistent solution can be represented in a matrix format. The equation system is solved using algebraic solutions as in LAPACK or SCALAPACK, integration solutions or iterative processes. ELPA has its own functions of linear algebra for critical areas in terms of performance.

### 4.2.3 Quantum ESPRESSO *installation with ELPA*

The next step in this study was to explore the impact of the ELPA solver on two different compute nodes of the SeARCH cluster: a dual multicore Xeon node and a single many-core Xeon Phi node. This section presents the several installations that were made on both servers, in order to accomplish a better tuning to the conventional installation of QE.

In the previous sections, the study was centered in tuning the installation of QE considering the system's architecture as the basis. On previous approaches, no changes were made to the default kernels used in order to make the necessary computations. From now on, all the presented installations use ELPA for different paradigms: shared and distributed memory.

ELPA can be installed using configuration wrapper scripts[3]. Fortunately, ELPA has a bunch of different wrappers scripts available for different paradigms and architectures, including KNL. The ELPA solver and QE configuration can be made using the tutorial suggested by Intel[4], which was used for the first installation of QE using ELPA.

Throughout this study, four different sets of installations[5] were made for each device, multicore and many-core (as stated in appendix A): a sequential installation, distributed and shared memory installations and a hybrid installation which combines shared and distributed memory. From these installations, different tests were made on the SeARCH with Intel Xeon based devices and KNL. These tests allows to understand the impact of the ELPA solver using two different devices categories: multicore and many-core. FFTW3 and SCALAPACK libraries were used, as the installations presented so far.

3 http://elpa.mpcdf.mpg.de/elpa-tar-archive
4 https://software.intel.com/en-us/articles/quantum-espresso-for-the-intel-xeon-phi-processor
5 https://gitlab.mpcdf.mpg.de/elpa/elpa/blob/6bb59888ab6a5c5558d1200083cb47ee765b37ab/INSTALL.md

## 4.3 PERFORMANCE EVALUATION ON MULTICORE DEVICES

This section presents the results of the experimental component of this work. An analysis of the achieved performance values using all the custom installations, faster solver libraries and parameter tuning used for the $WSe_2$ case study is presented. Even though the used parameters may apply to other case studies in a general way, only the $WSe_2$ case study was used to validate the results. The experimental results are shown in a comparative perspective to an usual (basic) installation, without using FFTW3 and MKL libraries or the ELPA solver.

The main goal of the tests is to measure the possible performance gains of using these libraries. The multiple test variants explores the best execution environment by changing the number of threads and processes evolved in the simulation, adopting FFTW3 library, faster solvers, using multiple compute nodes for parallel computation, or even changing the mapping strategy between threads/processes and processing units.

The first step measuring the performance evaluation of *PWscf* was to get a control flow graph that allows to measure the impact of the heaviest routines (and subroutines) in terms of number of calls and spent execution times. Figure 18 shows the critical path in terms of performance when executing *PWscf* running the $WSe_2$ case study. This *callgraph* was extracted and exploited using *callgrind* and the full *callgraph* tree is available in appendix C for future reference. This graphic representations helps to identify the heaviest routines in $WSe_2$ simulation, *PWscf* own routines or routines of their linked/external libraries that are called during the code execution. The heaviest routines have an high impact on the whole *PWscf* performance and are the ones that require performance improvements.

After collecting profiling data and drawing the callgraph multiple times, the critical path suggests that the heaviest routines in the execution are the ones of the MKL library. However, the MKL library is widely known for being extremely efficient since their routines are based on BLAS, LAPACK and SACLAPACK.

Since the performance of the *PWscf* directly depends on the performance of these libraries, getting a better performance out of the *PWscf* cannot be restricted to code inspection for detecting targets to parallelize the workload.

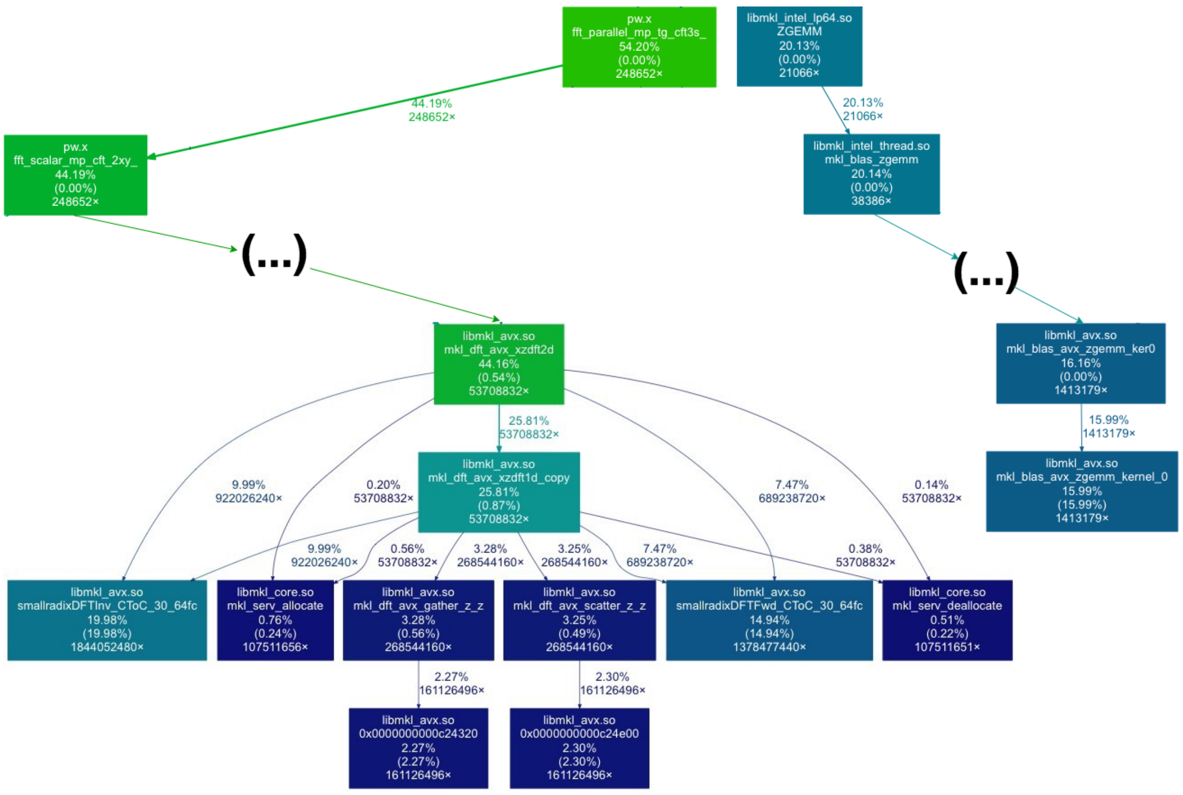


Figure 18.: Heaviest routines of the *PWscf*

The next steps in this study were redirected to understand how much the libraries configuration, *PWscf* configurations itself and the execution environment parameters at runtime can affect the performance of the whole simulation.

This section exploits a performance evaluation of $WSe_2$ simulations using *PWscf* only for multicore devices. The goal of these results are to measure the performance at the following levels:

1. The impact of multithread and multiprocess implementations.
2. An alternative installation adopting FFTW3 and LAPACK libraries explicitly linked, comparing the performance of the installation with the default linkage with no optimizations in the configuration phase of *PWscf*. This group of tests will also measure the impact of process affinity comparing the results achieved using two different mapping strategies: mapping processes by core and mapping processes by socket.
3. The impact of multinode computation.
4. Using parameterized executions for different parallel computing paradigms installations with ELPA: shared, distributed and hybrid memory.

5. The impact of adopting ELPA as the main eigensolver to solve the linear equation system in the simulation since *PWscf* supports the configuration of different direct solvers.

The following tests were performed using the 5-best metrics regarding the execution times. Each test was performed with a full compute node reservation and with a minimum number of processes running to decrease as much as possible the intrusion effect.

### *Impact of multithread and multiprocess*

Figure 19 shows the results achieved with a variable number of processes and threads in a 10-core node of the SeARCH.

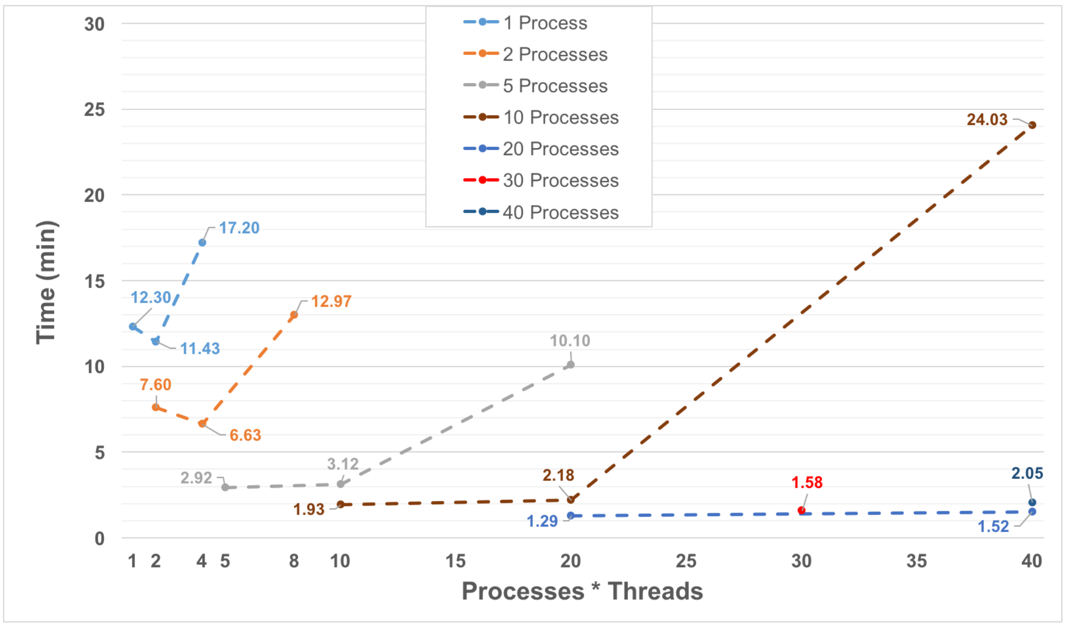
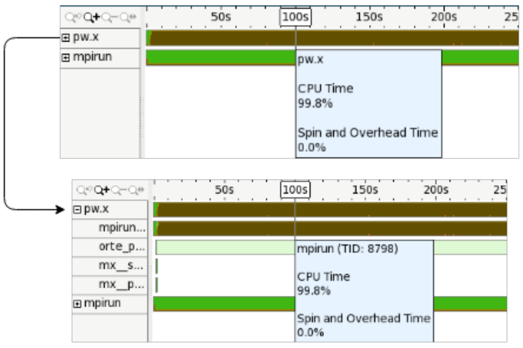


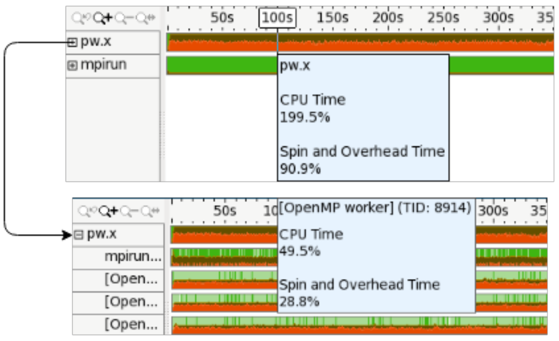
Figure 19.: Execution times to measure the impact of multithreading on multicore devices (dual 10-core server). **Reference time: 2.60 minutes**

Figure 20 shows the CPU time and the spin time for a single-process/single-thread and single-process/four-threads. The performance considerably decreases for more than one thread per core which indicates an handicap of *PWscf* taking advantage of multithread in multicore architectures.

After profiling the simulation for a single thread per core, VTune shows an almost 100% core usage. The same profiling applied to more than one thread per core shows that the wait time increases for all threads as long as their spinning time. This can explain why all the tests with more than one thread per core have worse results. This behaviour also



(a) *PWscf* execution with 1 Process 1 Thread



(b) *PWscf* execution with 1 Process 4 Threads

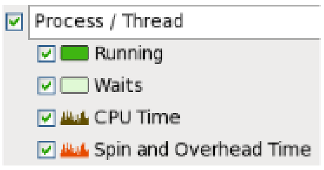


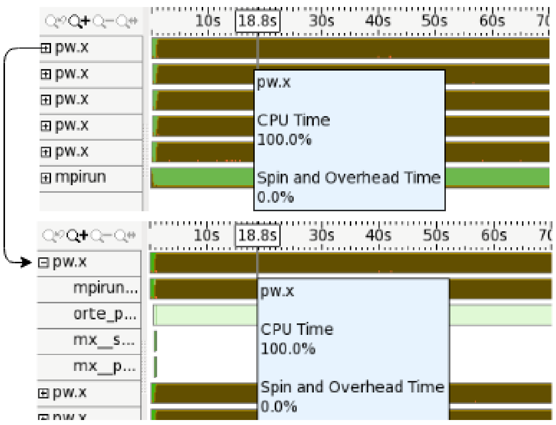
Figure 20.: VTune profiling diagrams for *PWscf* execution with 1 process

occurs in tests with more than one process using multiple threads which indicates that the workload is not being correctly balanced among all the threads. The threads are then in a waiting state and spinning most of the time as stated in the right side of the Figure 20b, where almost 30% of the time corresponds to spinning time.

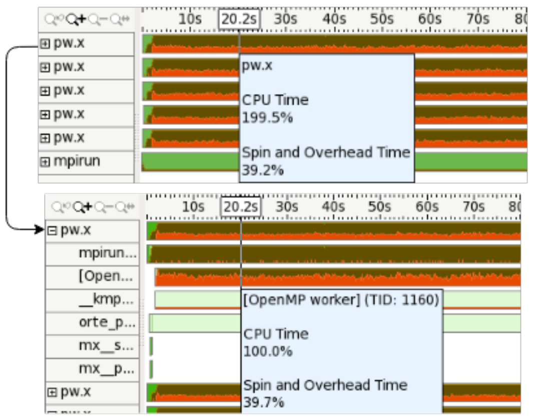
After making the same profiling for a higher number of processes, the effect of an increased spinning and overhead time is aggravated. As stated in Figure 21, the brown region which corresponds to an active state of the thread, is prominent for one thread per process, but the spinning time overlaps the next tests when the number of threads per process increases. As already showed in Figure 19, the adoption of a higher number of processes using a single thread contributes to lower execution times. On the other hand, when the number of threads per process is increased, the spinning and overhead time increases leading to higher execution times which consequently compromises the performance.

For the first group of tests, the best time achieved was 1.28 min using 20 processes and a single thread in a compute node with 20 physical cores. For future simulations, using a default installation of *PWscf*, this execution environment should be considered as having the best process/thread distribution. These first group of tests revealed that future versions of QE should improve the parallelism at a thread level, since the simulations performance decreased and the spinning time increased for more than one thread.

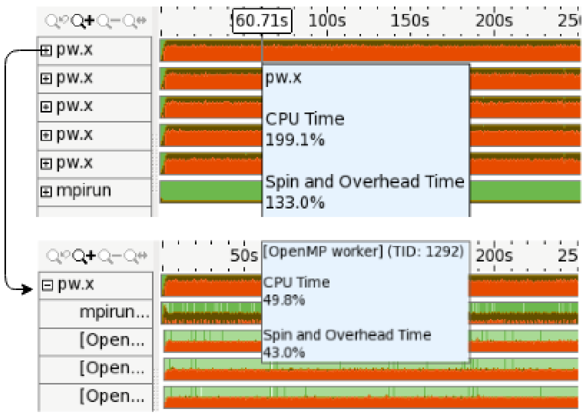
A better load balancing mechanism, routines and data structures should be improved to take advantage of *simultaneous multithreading (SMT)* capabilities of the hardware. Taking in consideration the reference time (2.60 minutes), the result achieved in this group of tests (1.28 minutes for 20 processes and 1 thread) has revealed a performance gain of about 2x.



(a) *PWscf* execution with 5 Process 1 Thread



(b) *PWscf* execution with 5 Process 2 Threads



(c) *PWscf* execution with 4 Threads

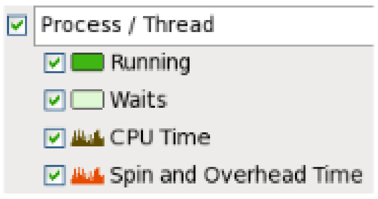


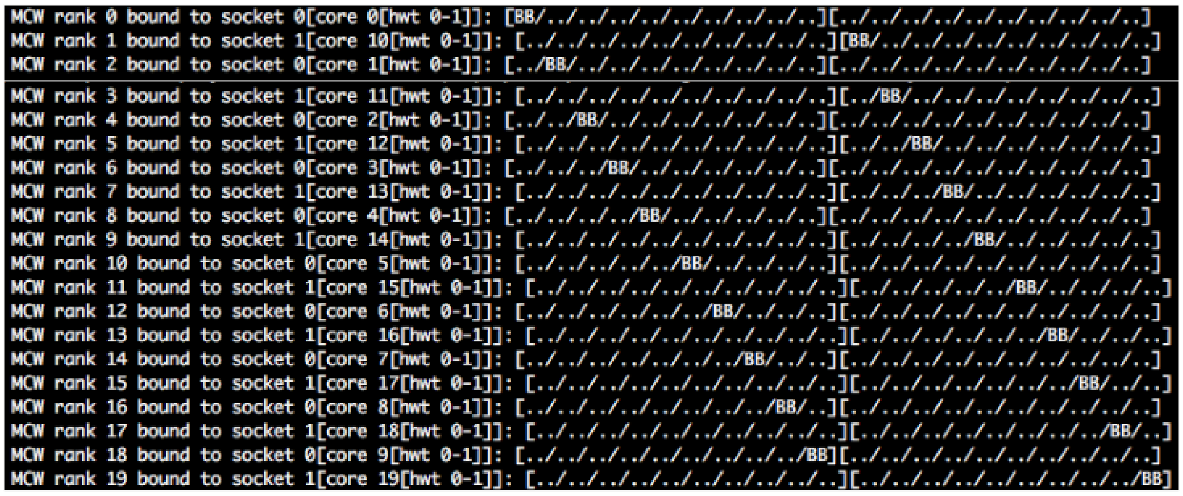
Figure 21.: VTune profiling diagrams for *PWscf* execution with 5 processes

### *Impact of FFTW3 and LAPACK libraries with explicit linkage and process mapping strategies*

The next group of tests explores the impact of the process affinity at the socket and core levels. The affinity concept is a popular technique in HPC that allows the process and threads distribution among physical cores and sockets in order to increase the amount of parallel workload. An wisely affinity strategy can significantly improve the workload throughput per second by distributing and pinning the processes to specific cores or sockets. This distribution can also improve the throughput in main memory accesses of NUMA architectures, by spreading processes among available cores and sockets, so that they can concurrently access data with less contentions.

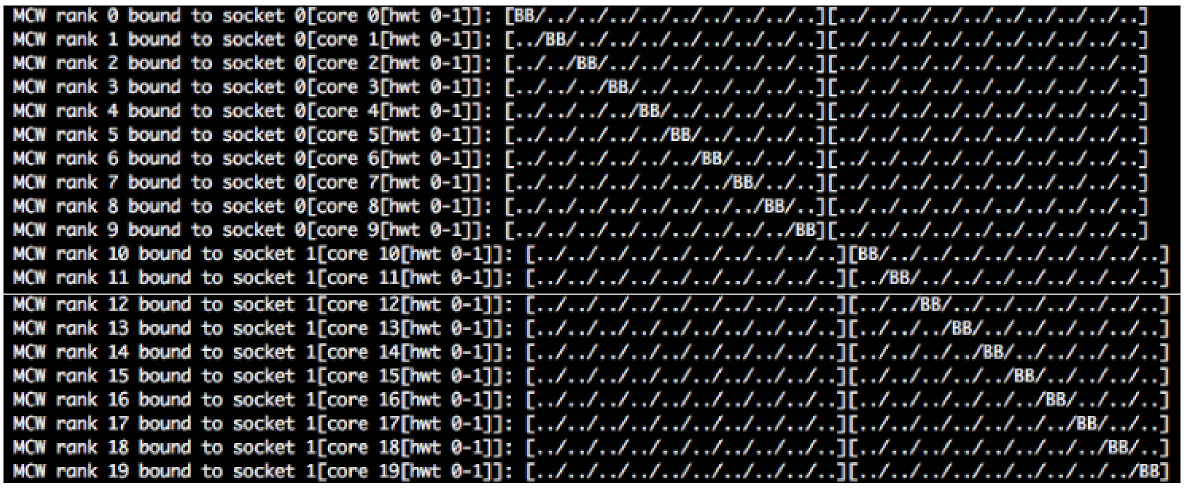
The "map" or "binding" designations are commonly used for affinity. They are used to describe this technique of distributing processes and threads among cores or sockets. There are multiple strategies available to perform these mappings. It is possible to distribute cores and threads in an interleaved fashion among cores in sockets, attribute more than one process or thread to the same physical core, distribute firstly to the cores of a certain socket and then to the other sockets, and many others.

In this study, a group of tests were made to measure the impact of two different mapping strategies provided by `mpirun`: `--map-by-socket` and `--map-by-core`. The `--map-by-socket` strategy uses an interleaved way of mapping among available sockets, as shown in Figure 22, while the `--map-by-core` first distributes the processes among the available cores of the first device and then among the cores of the other device (Figure 23).



```
MCW rank 0 bound to socket 0[core 0[hwt 0-1]]: [BB/../../../../../../../../..][../../../../../../../../../..]
MCW rank 1 bound to socket 1[core 10[hwt 0-1]]: [../../../../../../../../../..][BB/../../../../../../../../..]
MCW rank 2 bound to socket 0[core 1[hwt 0-1]]: [../BB/../../../../../../../..][../../../../../../../../../..]
MCW rank 3 bound to socket 1[core 11[hwt 0-1]]: [../../../../../../../../../..][../BB/../../../../../../../..]
MCW rank 4 bound to socket 0[core 2[hwt 0-1]]: [../../BB/../../../../../../..][../../../../../../../../../..]
MCW rank 5 bound to socket 1[core 12[hwt 0-1]]: [../../../../../../../../../..][../../BB/../../../../../../..]
MCW rank 6 bound to socket 0[core 3[hwt 0-1]]: [../../../BB/../../../../../..][../../../../../../../../../..]
MCW rank 7 bound to socket 1[core 13[hwt 0-1]]: [../../../../../../../../../..][../../../BB/../../../../../..]
MCW rank 8 bound to socket 0[core 4[hwt 0-1]]: [../../../../BB/../../../../..][../../../../../../../../../..]
MCW rank 9 bound to socket 1[core 14[hwt 0-1]]: [../../../../../../../../../..][../../../../BB/../../../../..]
MCW rank 10 bound to socket 0[core 5[hwt 0-1]]: [../../../../../BB/../../../..][../../../../../../../../../..]
MCW rank 11 bound to socket 1[core 15[hwt 0-1]]: [../../../../../../../../../..][../../../../../BB/../../../..]
MCW rank 12 bound to socket 0[core 6[hwt 0-1]]: [../../../../../../BB/../../..][../../../../../../../../../..]
MCW rank 13 bound to socket 1[core 16[hwt 0-1]]: [../../../../../../../../../..][../../../../../../BB/../../..]
MCW rank 14 bound to socket 0[core 7[hwt 0-1]]: [../../../../../../../BB/../..][../../../../../../../../../..]
MCW rank 15 bound to socket 1[core 17[hwt 0-1]]: [../../../../../../../../../..][../../../../../../../BB/../..]
MCW rank 16 bound to socket 0[core 8[hwt 0-1]]: [../../../../../../../../BB/..][../../../../../../../../../..]
MCW rank 17 bound to socket 1[core 18[hwt 0-1]]: [../../../../../../../../../..][../../../../../../../../BB/..]
MCW rank 18 bound to socket 0[core 9[hwt 0-1]]: [../../../../../../../../../BB][../../../../../../../../../..]
MCW rank 19 bound to socket 1[core 19[hwt 0-1]]: [../../../../../../../../../..][../../../../../../../../../BB]
```

Figure 22.: Diagram of the processes mapped by socket



```
MCW rank 0 bound to socket 0[core 0[hwt 0-1]]: [BB/../../../../../../../../..][../../../../../../../../../..]
MCW rank 1 bound to socket 0[core 1[hwt 0-1]]: [../BB/../../../../../../../..][../../../../../../../../../..]
MCW rank 2 bound to socket 0[core 2[hwt 0-1]]: [../../BB/../../../../../../..][../../../../../../../../../..]
MCW rank 3 bound to socket 0[core 3[hwt 0-1]]: [../../../BB/../../../../../..][../../../../../../../../../..]
MCW rank 4 bound to socket 0[core 4[hwt 0-1]]: [../../../../BB/../../../../..][../../../../../../../../../..]
MCW rank 5 bound to socket 0[core 5[hwt 0-1]]: [../../../../../BB/../../../..][../../../../../../../../../..]
MCW rank 6 bound to socket 0[core 6[hwt 0-1]]: [../../../../../../BB/../../..][../../../../../../../../../..]
MCW rank 7 bound to socket 0[core 7[hwt 0-1]]: [../../../../../../../BB/../..][../../../../../../../../../..]
MCW rank 8 bound to socket 0[core 8[hwt 0-1]]: [../../../../../../../../BB/..][../../../../../../../../../..]
MCW rank 9 bound to socket 0[core 9[hwt 0-1]]: [../../../../../../../../../BB][../../../../../../../../../..]
MCW rank 10 bound to socket 1[core 10[hwt 0-1]]: [../../../../../../../../../..][BB/../../../../../../../../..]
MCW rank 11 bound to socket 1[core 11[hwt 0-1]]: [../../../../../../../../../..][../BB/../../../../../../../..]
MCW rank 12 bound to socket 1[core 12[hwt 0-1]]: [../../../../../../../../../..][../../BB/../../../../../../..]
MCW rank 13 bound to socket 1[core 13[hwt 0-1]]: [../../../../../../../../../..][../../../BB/../../../../../..]
MCW rank 14 bound to socket 1[core 14[hwt 0-1]]: [../../../../../../../../../..][../../../../BB/../../../../..]
MCW rank 15 bound to socket 1[core 15[hwt 0-1]]: [../../../../../../../../../..][../../../../../BB/../../../..]
MCW rank 16 bound to socket 1[core 16[hwt 0-1]]: [../../../../../../../../../..][../../../../../../BB/../../..]
MCW rank 17 bound to socket 1[core 17[hwt 0-1]]: [../../../../../../../../../..][../../../../../../../BB/../..]
MCW rank 18 bound to socket 1[core 18[hwt 0-1]]: [../../../../../../../../../..][../../../../../../../../BB/..]
MCW rank 19 bound to socket 1[core 19[hwt 0-1]]: [../../../../../../../../../..][../../../../../../../../../BB]
```

Figure 23.: Diagram of the processes mapped by core

Depending on the algorithm used, the pattern of memory accesses, main memory allocations and the amount of independent workload available, it is not possible to state that a specific mapping strategy is better than another.

The plot in Figure 24 shows the execution times of the first two installations in order to evaluate the results of the core mapping, socket mapping and direct linked libraries.

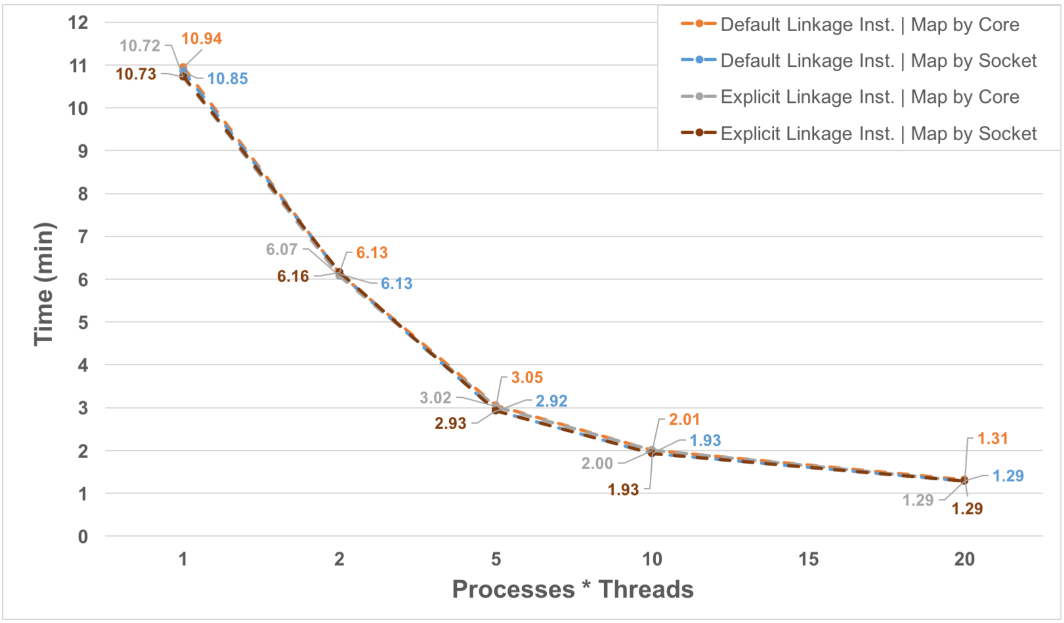


Figure 24.: Impact of explicit linkage libraries and process mapping on a multicore device (10-core server) with only 1 thread per process. **Reference time: 2.60 minutes**

In conclusion, the mapping by socket strategy and the installation with explicit linkage of FFTW3 and LAPACK libraries, did not had a significant impact compared to the installation used with default linkage. These four tests are a good example to demonstrate that the decrease execution time is mainly caused by the increase of the number of processes and not directly by the mapping strategy adopted. For this group of tests the performance gain over the reference time was 2x as the previous group of tests.

### *Impact of multinode computation*

The plot in Figure 25 shows the results achieved by running the application with a single and multiple compute nodes in parallel on the SeARCH. The test demonstrates that QE scales considerably well for this case study when multiple nodes are used. The best execution time achieved was with 32 processes, distributed to the cores among four nodes. Given that the SeARCH has a limitation of eight MPI processes per node, no other test with a higher number of processes was possible to develop.

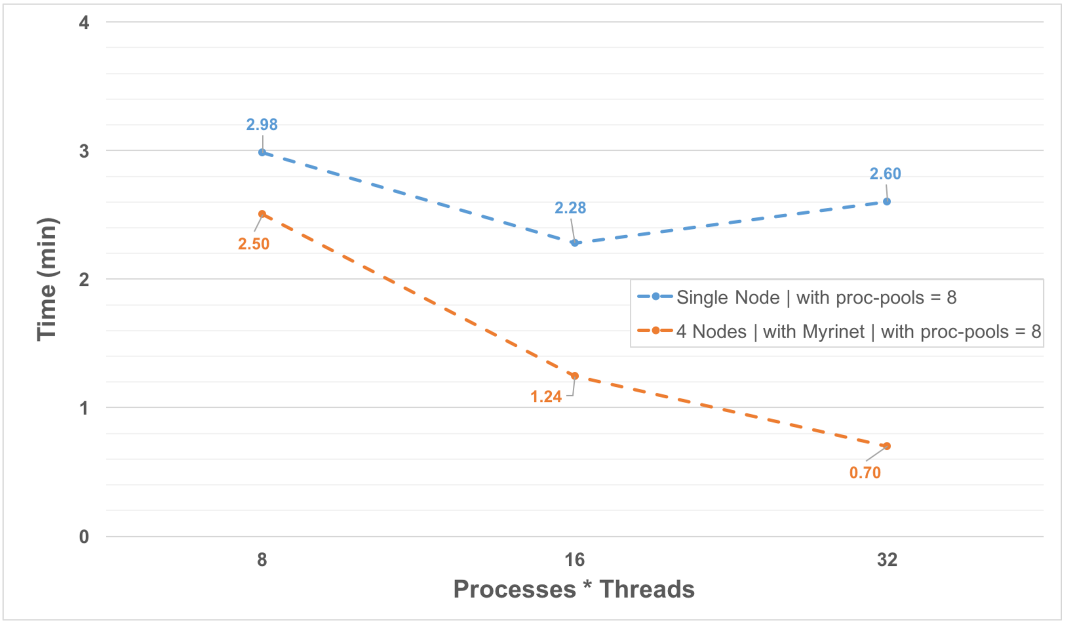


Figure 25.: Execution times to measure the node scalability for QE on multicore devices (dual 8-core server). **Reference time: 2.60 minutes**

Both tests were made using the same number of proc-pools, which corresponds to the $-nk$ parameter of QE. This parameter defines the the number of sets of cores to be used in the simulation. For example, with 32 cores choosing 8 proc-pools (sets), QE will divide those 32 cores by 8 sets resulting in 4 cores per set, so the number of $k$ points will be divided among these sets. After several tests, using a broad range of proc-pool numbers, the results using 8 proc-pools revealed to be the ones with the best performance configuration for the compute node used.

The configuration of the multiple node tests, required the development of a *hostfile* (4.4), which is invoked in the job script, with information of the compute nodes to be used in the test, using `mpirun` (as stated on listing 4.5) and the number of cores. In the listing bellow, the number of slots represents the number of cores to be used in each node.

```
compute-641-10 slots=8
compute-641-12 slots=8
compute-641-13 slots=8
compute-641-14 slots=8
```

Listing 4.4: Example of the hostfile

The tests revealed a decrease in execution times, using 4 nodes, as the number of processes used increases. This decrease is less accentuated after 16 processes are used. The single node tests showed that the execution times decreases when using up to 16 processes, which is the number of physical cores available in the node, at which point the usage of more processes leads to a gradually increase in the execution times.

```
mpirun -np $ppn --hostfile hosfile_name --mca mtl mx --mca pml cm ./
    excutable_name
```

Listing 4.5: Example of the mpirun command line used

The number of cores chosen should be equal to the total number of available cores in the node in order for the entire node to be reserved exclusively for the tests execution. The listing 4.6 shows how to properly request the entire resources of each compute node.

```
#PBS -l nodes=compute-641-10:ppn=32+compute-641-12:ppn=32+compute-641-13:ppn=32+
    compute-641-14:ppn=32
```

Listing 4.6: Example of nodes request line on PBS job

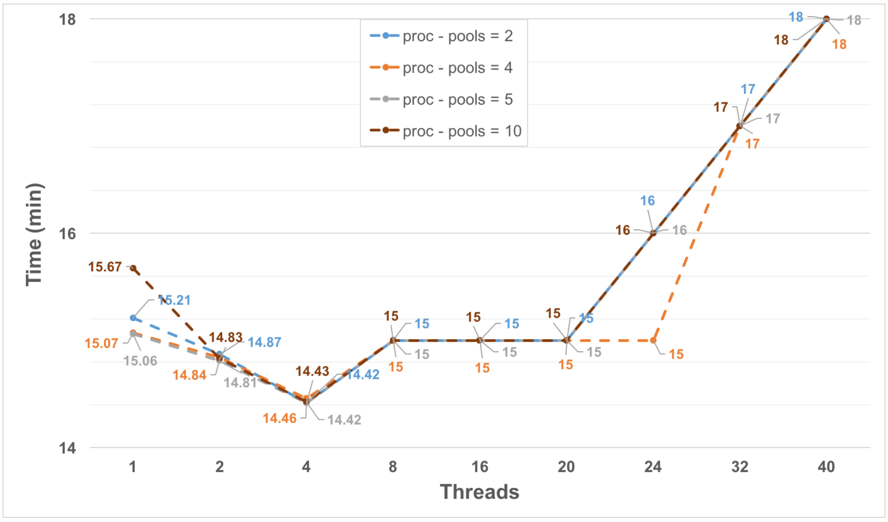
Even though only four nodes were used, the number of nodes can be increased, leading to a better expected performance. However an higher number of requested nodes implies an higher waiting time. In order to streamline the testing process, the maximum number of nodes that was possible to use in this study, were four compute nodes.

To summarize, the simulation scaled well in terms of performance, when more than one compute node was used with myrinet inter-node communication. Comparing the results between single node and 4 nodes, the best speedup was about 3.7 times. Taking in account the reference time, the performance gain was also about 3.7 times. The application scales well running the simulation with four compute nodes. However it was not possible to determine the critical point in execution time (the point where the application stops scaling) because it was not possible to test the simulation with more than four compute nodes due to limitations in the availability of nodes.

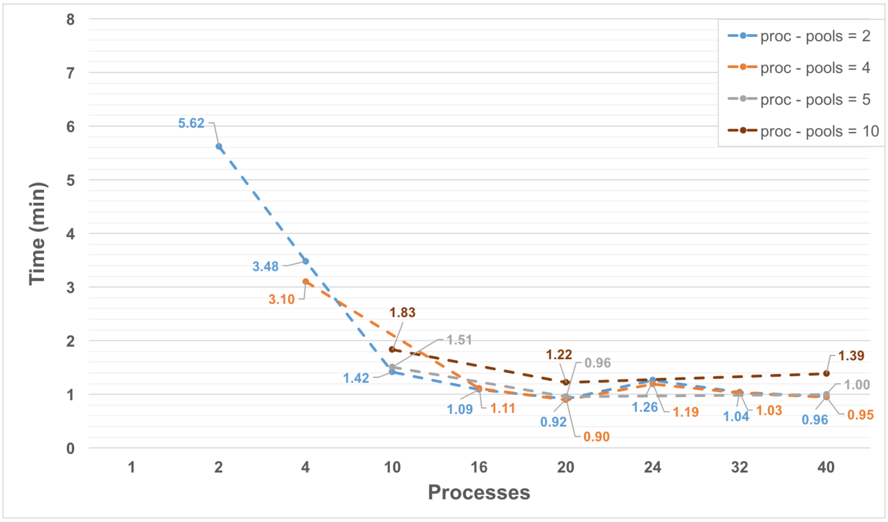
### *Parameterized executions for different parallel paradigms installations with ELPA*

The next group of tests are referred to the installations made with ELPA solver. The installation and the configuration process of the QE with this solver for different computing paradigms (shared, distributed and hybrid memory) are described in appendix A.

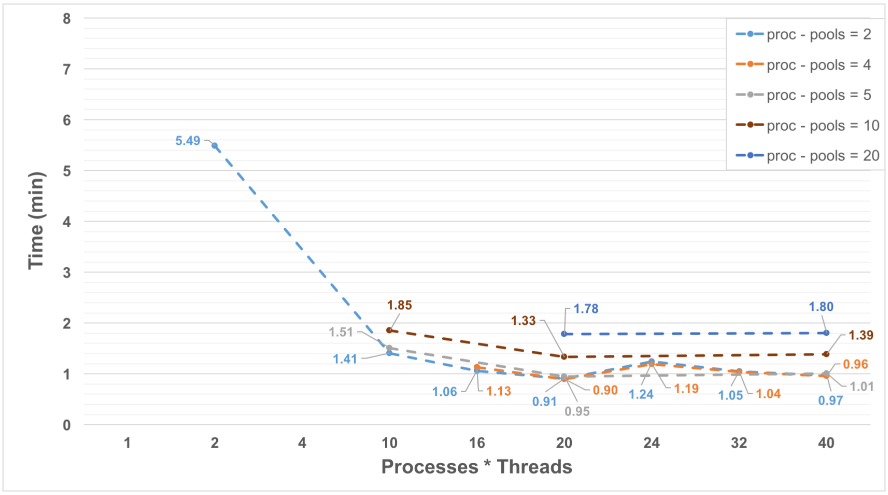
The first installation using ELPA solver was a sequential version of QE and ELPA. This sequential test allowed to assert that the $WSe_2$ simulation time was about 15.22 minutes in a dual 10-core server with 4 proc-pools. To achieve a better execution time, three groups of tests were made for multiple installation paradigms of the QE using ELPA, as shown in Figure 26.



(a) Shared memory installation



(b) Distributed memory installation



(c) Distributed and shared memory (hybrid) installation

Figure 26.: Execution times for shared, distributed and hybrid memory installations of QE using ELPA on a dual 10-core server. **Reference time: 2.60 minutes**

These results were obtained using shared, distributed and hybrid memory implementations for different numbers of proc-pools. These results were also the best results of all single-node multicore tests. The execution time with these implementations decreased from 15.22 minutes to 0.90 minutes in distributed and hybrid memory implementations, which reflects a gain of almost 17 times.

Only one thread per process was used for the distributed and hybrid memory implementations. However, these two installations are actually different since the hybrid installation can support multiple threads per process, while the distributed installation can not. Due to the fact that the use of more than one thread per process revealed to have the worst performance in previous tests (Figure 19), in both installations was used a single thread per process.

The best results of QE are usually achieved with distributed memory implementations. The obtained performance figures using distributed memory and shared memory reveals that parallelization made in shared memory implementations are worst than MPI implementations for distributed memory. Shared memory achieved the best results using 2 and 5 proc-pools distributed among 4 threads. The best results for distributed and hybrid memory implementations were obtained using 4 proc-pools distributed among 20 processes. The shared memory installation cannot compete with the distributed and hybrid installations and this is without taking into account that the hybrid installation did not use multiple threads per process, since previous tests showed that this is a worst solution, as mentioned before. Overall the performance gain of this group of tests over the reference time was about 2.8 times.

#### *Impact of using ELPA solver with QE*

To summarize the tests on multicore devices, the plot in Figure 27 shows the best results without using ELPA solver and the impact of using it. The use of the ELPA solver improved the performance of the simulation in about 10%. Both tests were performed using hybrid installations and the same number of proc-pools.

At this point in the study and in general, the installations using ELPA revealed a better performance compared with all previous groups of tests stated in this study so far. In the next section a performance evaluation will be explored in detail for tests in a many-core devices, taking advantage of hardware design re-configurations, ELPA solver and multi-paradigm installations.

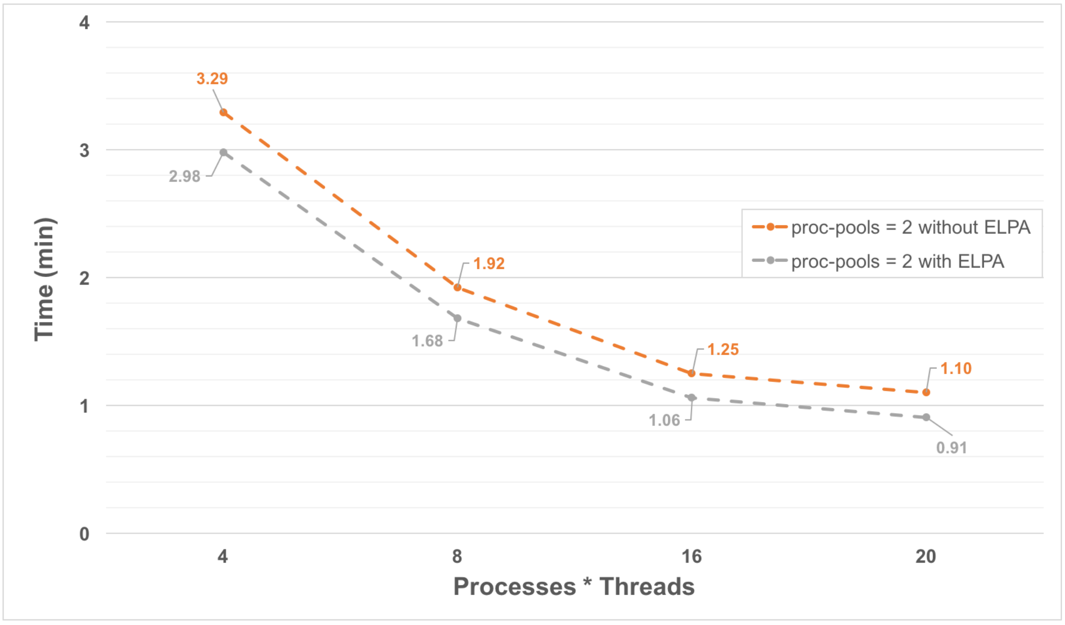


Figure 27.: Impact of using ELPA solver with QE on multicore devices (dual 10-core server)

## 4.4 PERFORMANCE EVALUATION ON MANY-CORE DEVICES

Until this section, all $WSe_2$ simulation tests were performed on a multicore environment in order to explore different computing paradigms and specific implementations.

In this section, the QE simulation is done in a many-core device to be able to explore different clustering and memory modes, already explained in Section 3.1. All tests in this section were performed on a single compute node with a many-core device, for multiple computing paradigm installations, variable number of processes and threads, and multiple parameter configurations. The goal of these results are to measure the performance at the following levels:

1. Impact of the cluster and memory modes with the use of proc-pools parameters.
2. Multi-paradigm installations with the `-numactl` switch.
3. Impact of ELPA solver on many-core devices with the use of proc-pools parameters.

Since the plots for multicore revealed that it was possible to increase the performance in about 10% using ELPA, all the tests for many-core devices were made with this solver.

### *Impact of the cluster and memory modes*

The bellow plot shows the execution times for the different node configurations . The values in the plot of Figure 28 are the fastest times for each configuration test, which combines two different cluster modes (all-to-all and quadrant), two memory modes (flat and cache mode), and a fixed value of 16 proc-pools.

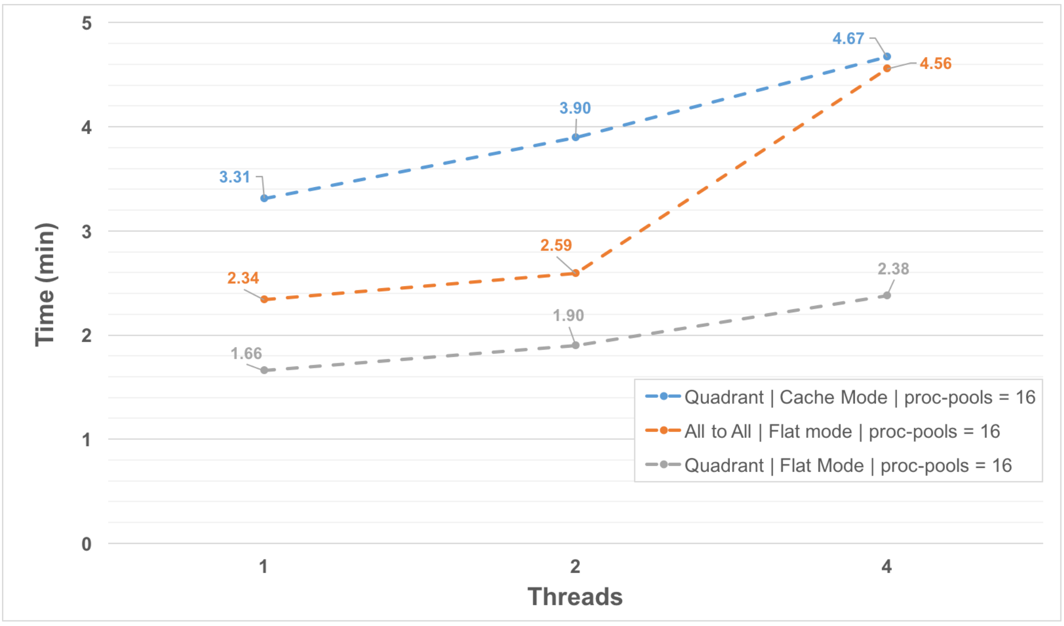


Figure 28.: Execution times for different node configurations for 64 processes. **Reference time: 2.60 minutes**

The results show that the best execution environment in terms of performance was achieved with the quadrant mode with flat memory access and using 16 proc-pools, which results in 16 sets of 4 cores, each responsible for a group of $k$ points. In addition to the use of ELPA with the many-core device, the quadrant and flat modes were fixed for the next group of tests. The result for quadrant and flat modes overtook the performance results of the reference time, with a gain of ~1.5 times, using 64 single-threaded processes.

### *Multi-paradigm installation tests*

Until now tests explored different hardware configurations, parameter values, number of processes and threads and mapping modes. However, none of the tests explicitly addressed memory allocations.

The next plots show the way memory allocations are performed in the many-core device and compares the results in multiple paradigms. All tests with `numactl` were made in flat mode, using the Unix tool `numactl` to instruct the compiler to place the data in the embedded or external RAM, using the following switches:

- `--preferred=1`: to give preference to MCDRAM, before allocating in external DRAM.
- `--membind=1`: to force memory allocations only in MCDRAM.
- `--membind=0`: to force memory allocations only in the external RAM.

Since tests on a shared memory environment with multicore devices showed that using more than 1 thread leads to a performance degradation, the same outcome was obtained on the many-core server.

Figure 29 shows that the performance of the distributed memory installation with `numactl --preferred=1` is better than hybrid installations with any other `numactl` configuration. This is true because accessing the MCDRAM is faster than accessing the external DRAM.

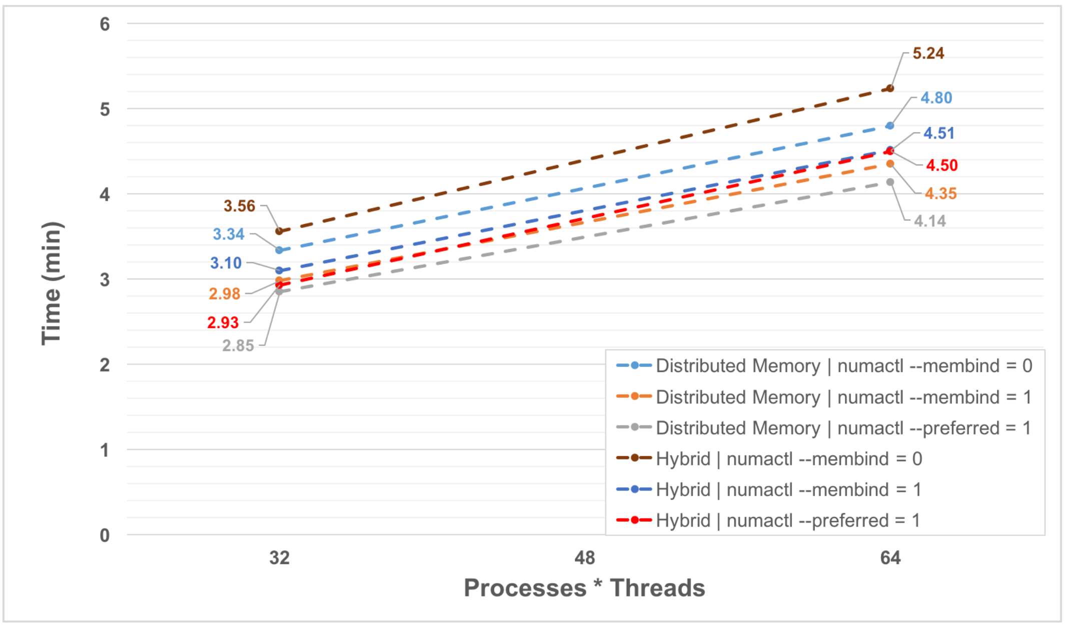


Figure 29.: Execution times for distributed memory and hybrid installation with `-numactl` switch. **Reference time: 2.60 minutes**

None of these tests overtook the performance of the reference time. However, the appropriate use of the parameters can actually improve these timings, as shown in the next group of tests.

### *Impact of ELPA solver on a many-core device*

Figure 30 illustrates the results of using ELPA solver in the simulation process, as was already done for multicore devices. For many-core devices it is possible to draw the same conclusions about the impact of this solver. The use of the solver has provided a small improvement in the overall performance of the simulation. The higher gain obtained in this group of tests reflects a performance gain in execution time of 17% for 16 proc-pools.

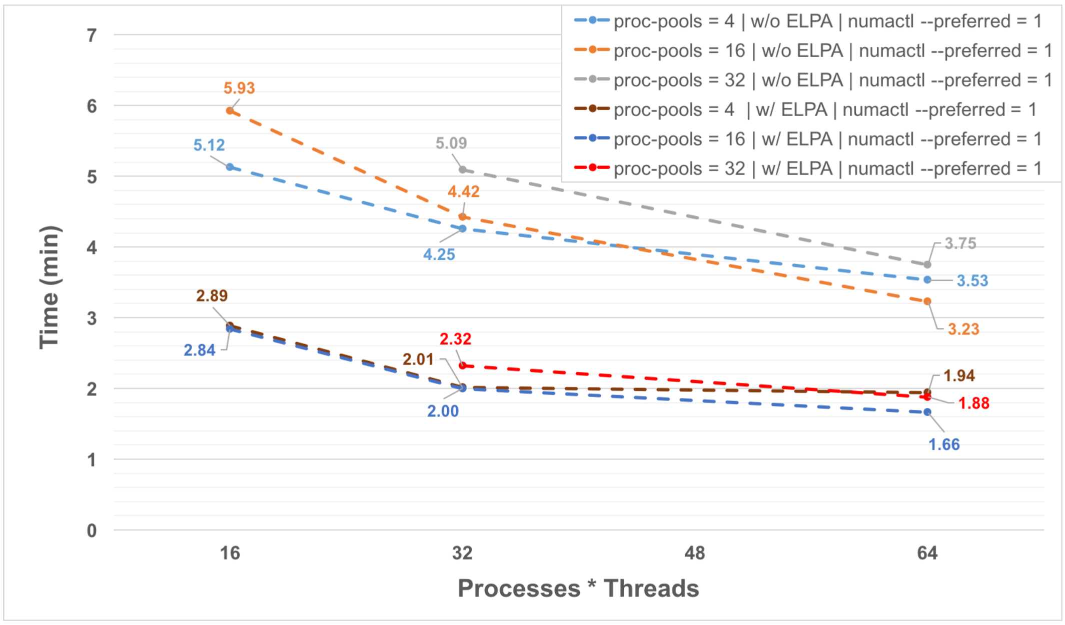


Figure 30.: Impact of ELPA in QE using proc-pools parameter with variable nº of processes and 1 thread per each process. **Reference time: 2.60 minutes**

Comparing the usage of ELPA between multicore and many-core tests, these final results shows that the impact of ELPA is more expressive in many-core devices. The installation of ELPA is then recommended for this type of simulations and the QE installation should preceed the installation of ELPA so that the library can be linked to QE. The three tests without the usage of ELPA resulted in a performance loss so they can be ignored. For both implementations with and without ELPA, the usage of 16 proc-pools revealed to be the best parameter configuration for performance. Contrary to the previous tests (where the proc-pools parameter was not used), the execution times from 32 processes to 64 processes actually decreased.

To guarantee a fair comparison between tests, all were made using the memory allocation on MCDRAM whenever possible, otherwise the memory allocation will be made in DRAM. This was ensured using `numactl -preferred=1` since the tests in Figure 29 revealed that this was the best data allocation strategy.

The usage of proc-pools parameters, showed in Figure 30, had a positive impact on performance. When this parameter is used with a value of 16, the total number of available cores in the hardware (64 cores) is organized in 16 sets (4 cores per set). The number of *k* points are internally divided between these 16 sets. Probably these 16 sets are evenly divided by the 4 quadrants (as described before, this test uses a quadrant mode configuration). When the proc-pools parameter is not used, there is only 1 set with all the cores, and the distribution of *k* points across them may be uneven (worst case scenario, one core may receive all the k points to process while the other cores idle).

The main advantage of the quadrant mode is the logical division of all tiles and each quadrant is assigned to the closest MCDRAM. The tiles of each quadrant privileges the memory access to MCDRAM of its own quadrant, so the distance incurred in a memory access by a tile in a quadrant is shorter, as long as the latency penalty.

The results shows that this presumed combination between proc-pool parameters and quadrant mode, leads to a performance improvement, specially when compared to the tests/configurations without the usage of proc-pools. Compared to the referenced time, the performance gain of using 64 processes, 1 thread and 16 proc-pools was about 1.5 times.

# 5 CONCLUSIONS

## 5.1 SUMMARY

The main goal of the developed work in this dissertation was to aid physicists on quantum mechanical modeling problems, namely the usage of density functional theory and QUANTUM ESPRESSO, to improve the execution times of their software application in different parallel computing platforms, supporting both shared and distributed memory environments. Although the QE tool already supports both parallelism paradigms, the multiple available versions and configuration options have a significant impact on the application performance. This required a detailed analysis and experimental work to determine the performance outcomes using different software installations and configurations, different multi-thread and multi-process implementations, different parallel computing platforms, memory organizations and hardware configurations, namely on many-core devices such as the Intel Xeon Phi KNL.

The case study was the tungsten diselenide simulation process using the *PWscf* module from QUANTUM ESPRESSO, while the testbed used was the SeARCH cluster environment. This study allowed to gauge a tuned installation and execution environment in terms of performance running *PWscf*. The obtained outcomes can serve as a starting point to enable the physics community to be able to speedup application codes similar to the $WSe_2$ simulation in QE, using an installation with better performance and a better execution environment for different computing paradigms (shared, distributed and hybrid).

As a starting point, a profiling study was made using the Intel VTune 2016 in order to inspect the critical path of the whole simulation process and discover which routines were more computationally intensive. The results lead to the conclusion that most of the simulation execution time was spent performing numerical computations with the Intel MKL library, which contains routines that already have a very high level of computational efficiency. With this conclusion, the focus on this work was in a performance analysis to measure the impact of multi-threading and multi-processing using multicore and many-core based servers.

The reference time to be improved was the better execution time a physics researcher managed to reach with a standard installation and configuration of the Quantum ESPRESSO package. To improve this time, the first step was to address the simulation execution environment in QE: changes were made in the installation process, in the simulation parameters configuration, testing each variation's performance against the result achieved with a standard installation. The reference execution time was 2.60 min, achieved with 20 processes on a dual 10-core server.

The developed tests are organized in two major groups: the tests made on multicore servers (dual 10-core and dual 8-core servers), and on a server with a many-core device (the Intel Xeon Phi KNL).

The multicore group of tests acheived better execution times when the simulation used an ELPA solver and was performed on a dual 10-core server, with 20 single-threaded processes and a number of proc-pools equal to 4, in a distributed memory environment. For these parameters, the simulation execution time was ~2.8 times faster than the reference time.

The hybrid installation lead to the same gain as in the distributed memory installation. This same gain is taking into account that it was not possible to take full advantage of the hybrid installation since using more than one thread immediately lead to a performance degradation. With the ELPA solver, in multicore devices, the gains were 10% more compared with the same configuration test but without this solver. Linking FFTW3 and LAPACK libraries did not lead to any significant improvement. The shared memory approach registered the worst time results and this paradigm should be discarded when exploring performance on this device with the Quantum ESPRESSO package.

Using multiple nodes - 4 dual 8-core servers interconnected with a 10 Gb Myrinet and 8 proc-pools - the performance gain was even higher: 3.7 times faster than the reference time.

The second group of tests, using the many-core device, concluded that the quadrant flat mode was the best KNL node configuration to run the QE simulation of $WSe_2$. The best execution time was 1.5x faster than the reference time, using 16 proc-pools and 64 processes with a single thread.

Using this device the tests also showed that multi-paradigm installations did not provide a significant performance improvement. However, the distributed memory installation managed to be the best paradigm option using `numactl -preferred=1`, compared with the hybrid and shared memory installations, as happened in the first group of multicore tests. The shared memory tests led to the same conclusion as in multicore devices: the worst results and this negative impact was even larger than on multicore tests.

The simulation using the ELPA solver led to a 17% gain over the simulation times without this solver, compared with the 10% gain measured in the multicore server. The usage of parameters in ELPA test revealed a significant execution time reduction for a number of proc-pools equal to 16. However, the overall performance gain with the ELPA solver was lower than in the multicore server: only 1.5 times the reference time.

Comparing multicore and many-core results, the best one were achieved in the multicore device. This can be explained by the fact that the simulation is memory-bound and despite of the high memory bandwidth of the embedded RAM in the many-core KNL package, if the number of memory accesses in the simulation is significantly high, the simulation performance is drastically affected by the lack of a L3 cache.

To summarize, the best execution time for the simulation of $WSe_2$ in QE was achieved on a multi-node configuration (3.7 times faster than the reference time on 4 dual 8-core servers), while the best configuration on a single server was on a dual 10-core server with a distributed memory environment using ELPA solver (2.8 times faster than the reference time). The number of processes considered in the execution environment should be equal to the number of physical cores using 4 proc-pools. An installation tutorial can be accessed in the Appendix A.1.3.

## 5.2 FUTURE WORK

After all the tests made searching for the best execution environment to run $WSe_2$ simulations in QE, some considerations and suggestions can be made in order to expand this study. All experimental tests aimed to tune the execution environment of a software package for the different computing paradigms, without performing any changes to the code itself.

All the improvements in execution times for the simulation were achieved by changing the way the software operates on data, how that data is mapped in the memory hierarchy and the heuristics used to distribute the workload in runtime. Since the QE algorithm is memory bound, some possible changes in the code can be considerd in order to improve the way memory data is stored - for example, by changing the memory allocator. When the HBM modes were studied in section 3.1.2, the pointer to the developer's guide[3] suggests the usage of the Memkind library to take advantage of the memkind and the high bandwidth memory allocator (hbwmalloc).

As showed in the results of the shared memory tests, using more than one thread always leads to a performance degradation. This was the main limitation found in QE for multi-threaded tests. Therefore, the second suggestion for future work is to improve the routines parallelization, and the workload division among cores for this paradigm. This suggestion should be seriously considered because the benefits of improving the shared memory paradigm can lead to a positive impact in the hybrid memory simulation environments.

The best results achieved were when the computation of the simulation was distributed among multiple independent nodes. This led to the conclusion that QE positively scales the performance in a distributed memory environment. However, due to limitations in the number of nodes and the number of processes per node, it was only possible to use 4 compute nodes and 8 processes per each. As a consequence, the inflection point for this scalability was not possible to determine. If more than 4 nodes were used, maybe it would have been possible to improve the execution time even more. The third suggestion refers to the determination of this inflection point, where the simulation stops scaling in order to understand the limits of distributed memory simulations for this case study.

The last suggestion refers to the exploration of the simulation performance in heterogeneous environments, using GPUs to offload heavy workloads, since QE offers this support. This type of computing accelerators were not explored in this study since the goal was to explore the expected performance gains, using this kind of simulations in a KNL server. This server was the most recent many-core based server installed in SeARCH cluster when this dissertation work started.

# BIBLIOGRAPHY

[1] R. Coehoorn, C. Haas, J. Dijkstra, C. J. F. Flipse, R. A. de Groot, and A. Wold. Electronic structure of mose$_2$, mos$_2$, and wse$_2$. i. band-structure calculations and photoelectron spectroscopy. *Physical review. B, Condensed matter*, 35, 1987.

[2] M. d'Avezac, M. Calandra, and F. Mauri. Density functional theory description of hole-trapping in Sio$_2$: A self-interaction-corrected approach. *Physical review. B, Condensed matter and materials physics*, 71, 2005.

[3] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, and et al. Quantum ESPRESSO: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21, 2009.

[4] P. Giannozzi, O. Andreussi, T. Brumme, O. Bunau, M. B. Nardelli, M. Calandra, R. Car, and et al. Advanced capabilities for materials modelling with Quantum ESPRESSO. *Journal of Physics: Condensed Matter*, 29, 2017.

[5] J. Hafner, C. Wolverton, and G. Ceder. Toward computational materials design: The impact of density functional theory on materials research. *MRS Bulletin*, 31, 2006.

[6] J. Kohanoff. *Electronic Structure Calculations for Solids and Molecules: Theory and Computational Methods*. Cambridge University Press, 1st edition, 2006.

[7] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *The Physical review*, 140, 1965.

[8] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H-J Bungartz, and H. Lederer. The elpa library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter*, 26, 2014.

[9] J. Paier, R. Hirschl, M. Marsman, and G. Kresse. The perdew–burke–ernzerhof exchange-correlation functional applied to the g2-1 test set using a plane-wave basis set. *The Journal of Chemical Physics*, 122, 2005.

[10] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Chen Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36, 2016.

[11] J. M. Soler, E. Artacho, J. D. Gale, A. García, J. Junquera, P. Ordejón, and D. Sánchez-Portal. The siesta method for ab initio order-n materials simulation. *Journal of Physics: Condensed Matter*, 14, 2002.

[12] D. Stanković, P. Jovanović, A. Jović, V. Slavnić, D. Vudragović, and A. Balaž. Implementation and benchmarking of new fft libraries in quantum espresso. In *High-Performance Computing Infrastructure for South East Europe's Research Communities*. Springer, 2014.

# A

# TUTORIAL FOR QUANTUM ESPRESSO INSTALLATIONS WITH ELPA

## A.1 QUANTUM ESPRESSO INSTALLATION WITH ELPA ON MULTICORE AND MANY-CORE

### A.1.1 *Sequential installation*

1. Download the ELPA configure wrapper scripts (XCONFIGURE reference).
2. Download an ELPA release and unarchive:

```
$ wget http://elpa.mpcdf.mpg.de/html/Releases/2016.11.001.pre/elpa
    -2016.11.001.pre.tar.gz
$ tar xvf elpa-2016.11.001.pre.tar.gz
$ cd elpa-2016.11.001.pre
```

3. Copy the respective configure wrapper scripts (according to your architecture) into ELPA root folder.

   For Sandy Bridge architecture (multicore):

```
$ cp /path/to/xconfigure/elpa/configure-elpa-snb.sh .
```

   For KNL architecture (many-core):

```
$ cp /path/to/xconfigure/elpa/configure-elpa-knl.sh .
```

4. Load the compiler modules.

   On multicore:

```
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/bin/
  compilervars.sh intel64
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/mkl/
  bin/mklvars.sh intel64
```

On many-core:

```
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/bin/
  compilervars.sh intel64
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/mkl/bin/
  mklvars.sh intel64
```

5. Run the configuration wrapper script for ELPA.

   On multicore node:

```
$ ./configure-elpa-snb.sh <directory_seq_installation_elpa> --with-mpi=0
```

   On many-core node:

```
$ ./configure-elpa-knl.sh <directory_seq_installation_elpa> --with-mpi=0
```

   *<directory_seq_installation_elpa>* is the directory where sequential version of ELPA will be installed and `--with-mpi=0` will disable the MPI.

6. Build and install ELPA:

```
$ make -j
$ make install
```

7. Download QE and unarchive:

```
$ wget https://github.com/QEF/q-e/archive/qe-6.1.0.tar.gz
$ tar xvf qe-6.1.tar.gz
$ cd qe-6.1
```

8. Copy the respective configure wrapper scripts (according to your architecture) into QE root folder.

   For Sandy Bridge architecture (multicore):

```
$ cp /path/to/xconfigure/qe/configure-qe-snb.sh .
```

   For KNL architecture (many-core):

```
$ cp /path/to/xconfigure/qe/configure-qe-knl.sh .
```

9. Load the compiler modules.

   On multicore:

```
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/bin/
   compilervars.sh intel64
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/mkl/
   bin/mklvars.sh intel64
```

   On many-core:

```
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/bin/
   compilervars.sh intel64
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/mkl/bin/
   mklvars.sh intel64
```

10. Run the configuration wrapper script for QE.

    On many-core:

```
$ ./configure-qe-snb.sh <directory_seq_installation_qe> --disable-
   parallel
```

On multicore:

```
$ ./configure-qe-knl.sh <directory_seq_installation_qe> --disable-
    parallel
```

*<directory_seq_installation_qe>* is the directory where sequential QE will be installed (the name of this directory need to be the same of the *<directory_seq_installation_elpa>* and need to be in the same directory of ELPA folder). `--disable-parallel` will disable the MPI.

11. Add the `-D__FFTW3` flag on DFLAGS in make.inc

```
DFLAGS = ... -D__FFTW3
```

12. Build the QE application that you need (e.g., "pw", "cp", or "all"):

```
$ make pw -j
```

### A.1.2 *Shared Memory Installation*

1. Download the ELPA configure wrapper scripts (XCONFIGURE reference).
2. Download an ELPA release and unarchive:

```
$ wget http://elpa.mpcdf.mpg.de/html/Releases/2016.11.001.pre/elpa
    -2016.11.001.pre.tar.gz
$ tar xvf elpa-2016.11.001.pre.tar.gz
$ cd elpa-2016.11.001.pre
```

3. Copy the respective configure wrapper scripts (according to your architecture) into ELPA root folder.

   For Sandy Bridge architecture (multicore):

```
$ cp /path/to/xconfigure/elpa/configure-elpa-snb-omp.sh .
```

For KNL architecture (many-core):

```
$ cp /path/to/xconfigure/elpa/configure-elpa-knl-omp.sh .
```

4. Load the compiler modules.

   On multicore:

```
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/bin/
  compilervars.sh intel64
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/mkl/
  bin/mklvars.sh intel64
```

   On many-core:

```
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/bin/
  compilervars.sh intel64
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/mkl/bin/
  mklvars.sh intel64
```

5. Run the configuration wrapper script for ELPA.

   On multicore node:

```
$ ./configure-elpa-snb-omp.sh <directory_omp_installation_elpa> --with-
  mpi=0
```

   On many-core node:

```
$ ./configure-elpa-knl-omp.sh <directory_omp_installation_elpa> --with-
  mpi=0
```

   <*directory_omp_installation_elpa*> is the directory where shared memory version of ELPA will be installed and `--with-mpi=0` will disable the MPI.

6. Build and install ELPA:

```
$ make -j
$ make install
```

7. Download QE and unarchive:

```
$ wget https://github.com/QEF/q-e/archive/qe-6.1.0.tar.gz
$ tar xvf qe-6.1.tar.gz
$ cd qe-6.1
```

8. Copy the respective configure wrapper scripts (according to your architecture) into QE root folder.

   For Sandy Bridge architecture (multicore):

```
$ cp /path/to/xconfigure/qe/configure-qe-snb-omp.sh .
```

   For KNL architecture (many-core):

```
$ cp /path/to/xconfigure/qe/configure-qe-knl-omp.sh .
```

9. Load the compiler modules.

   On multicore:

```
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/bin/
    compilervars.sh intel64
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/mkl/
    bin/mklvars.sh intel64
```

   On many-core:

```
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/bin/
    compilervars.sh intel64
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/mkl/bin/
    mklvars.sh intel64
```

10. Run the configuration wrapper script for QE. On many-core:

```
$ ./configure-qe-snb-omp.sh <directory_omp_installation_qe> --disable-
    parallel
```

On multicore:

```
$ ./configure-qe-knl-omp.sh <directory_omp_installation_qe> --disable-
    parallel
```

<*directory_omp_installation_qe*> is the directory where shared memory version of QE will be installed (this directory need to be in the same directory of ELPA folder, the name of this directory need to be the same of the <*directory_omp_installation_elpa*>). `--disable-parallel` will disable the MPI.

11. Add the `-D__FFTW3` flag on DFLAGS in make.inc

```
DFLAGS = ... -D__FFTW3
```

12. Build the QE application that you need (e.g., "pw", "cp", or "all"):

```
$ make pw -j
```

### A.1.3 *Distributed Memory Installation*

1. Download the ELPA configure wrapper scripts (XCONFIGURE reference).
2. Download an ELPA release and unarchive:

```
$ wget http://elpa.mpcdf.mpg.de/html/Releases/2016.11.001.pre/elpa
    -2016.11.001.pre.tar.gz
$ tar xvf elpa-2016.11.001.pre.tar.gz
$ cd elpa-2016.11.001.pre
```

3. Copy the respective configure wrapper scripts (according to your architecture) into ELPA root folder.

   For Sandy Bridge architecture (multicore):

   ```
   $ cp /path/to/xconfigure/elpa/configure-elpa-snb.sh .
   ```

   For KNL architecture (many-core):

   ```
   $ cp /path/to/xconfigure/elpa/configure-elpa-knl.sh .
   ```

4. Load the compiler modules.

   On multicore:

   ```
   $ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/bin/
     compilervars.sh intel64
   $ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/mkl/
     bin/mklvars.sh intel64
   ```

   On many-core:

   ```
   $ source /opt/intel/compilers_and_libraries_2017.3.191/linux/bin/
     compilervars.sh intel64
   $ source /opt/intel/compilers_and_libraries_2017.3.191/linux/mkl/bin/
     mklvars.sh intel64
   ```

5. Run the configuration wrapper script for ELPA.

   On multicore node:

   ```
   $ ./configure-elpa-snb.sh <directory_mpi_installation_elpa>
   ```

On many-core node:

```
$ ./configure-elpa-knl.sh <directory_mpi_installation_elpa>
```

*<directory_mpi_installation_elpa>* is the directory where distributed memory version of ELPA will be installed.

6. Build and install ELPA:

```
$ make -j
$ make install
```

7. Download QE and unarchive:

```
$ wget https://github.com/QEF/q-e/archive/qe-6.1.0.tar.gz
$ tar xvf qe-6.1.tar.gz
$ cd qe-6.1
```

8. Copy the respective configure wrapper scripts (according to your architecture) into QE root folder.

   For Sandy Bridge architecture (multicore):

```
$ cp /path/to/xconfigure/qe/configure-qe-snb.sh .
```

   For KNL architecture (many-core):

```
$ cp /path/to/xconfigure/qe/configure-qe-knl.sh .
```

9. Load the compiler modules.

   On multicore:

```
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/bin/
    compilervars.sh intel64
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/mkl/
    bin/mklvars.sh intel64
```

On many-core:

```
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/bin/
  compilervars.sh intel64
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/mkl/bin/
  mklvars.sh intel64
```

10. Run the configuration wrapper script for QE. On many-core:

```
$ ./configure-qe-snb.sh <directory_mpi_installation_qe>
```

On multicore:

```
$ ./configure-qe-knl.sh <directory_mpi_installation_qe>
```

<*directory_mpi_installation_qe*> is the directory where distributed memory version of QE will be installed (the name of this directory need to be the same of the <*directory_mpi_installation_elpa*> and need to be in the same directory of ELPA folder).

11. Add the `-D__FFTW3` flag on DFLAGS in make.inc

```
DFLAGS = ... -D__FFTW3
```

12. Build the QE application that you need (e.g., "pw", "cp", or "all"):

```
$ make pw -j
```

### A.1.4 *Hybrid Installation*

1. Download the ELPA configure wrapper scripts (XCONFIGURE reference).

2. Download an ELPA release and unarchive:

```
$ wget http://elpa.mpcdf.mpg.de/html/Releases/2016.11.001.pre/elpa
  -2016.11.001.pre.tar.gz
$ tar xvf elpa-2016.11.001.pre.tar.gz
$ cd elpa-2016.11.001.pre
```

3. Copy the respective configure wrapper scripts (according to your architecture) into ELPA root folder.

   For Sandy Bridge architecture (multicore):

```
$ cp /path/to/xconfigure/elpa/configure-elpa-snb-omp.sh .
```

   For KNL architecture (many-core):

```
$ cp /path/to/xconfigure/elpa/configure-elpa-knl-omp.sh .
```

4. Load the compiler modules.

   On multicore:

```
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/bin/
  compilervars.sh intel64
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/mkl/
  bin/mklvars.sh intel64
```

   On many-core:

```
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/bin/
  compilervars.sh intel64
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/mkl/bin/
  mklvars.sh intel64
```

5. Run the configuration wrapper script for ELPA.

   On multicore node:

```
$ ./configure-elpa-snb-omp.sh <directory_hybrid_installation_elpa>
```

On many-core node:

```
$ ./configure-elpa-knl-omp.sh <directory_hybrid_installation_elpa>
```

*<directory_hybrid_installation_elpa>* is the directory where hybrid version of ELPA will be installed.

6. Build and install ELPA:

```
$ make -j
$ make install
```

7. Download QE and unarchive:

```
$ wget https://github.com/QEF/q-e/archive/qe-6.1.0.tar.gz
$ tar xvf qe-6.1.tar.gz
$ cd qe-6.1
```

8. Copy the respective configure wrapper scripts (according to your architecture) into QE root folder.

   For Sandy Bridge architecture (multicore):

```
$ cp /path/to/xconfigure/qe/configure-qe-snb-omp.sh .
```

   For KNL architecture (many-core):

```
$ cp /path/to/xconfigure/qe/configure-qe-knl-omp.sh .
```

9. Load the compiler modules.

   On multicore:

```
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/bin/
   compilervars.sh intel64
$ source /share/apps/intel/compilers_and_libraries_2017.4.196/linux/mkl/
   bin/mklvars.sh intel64
```

On many-core:

```
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/bin/
   compilervars.sh intel64
$ source /opt/intel/compilers_and_libraries_2017.3.191/linux/mkl/bin/
   mklvars.sh intel64
```

10. Run the configuration wrapper script for QE. On many-core:

```
$ ./configure-qe-snb-omp.sh <hybrid_installation_directory>
```

On multicore:

```
$ ./configure-qe-knl-omp.sh <hybrid_installation_directory>
```

<*directory_hybrid_installation_qe*> is the directory where hybrid version of QE will be installed (this directory need to be in the same directory of ELPA folder and the name of this directory need to be the same of the <*directory_hybrid_installation_elpa*>).

11. Add the `-D__FFTW3` flag on DFLAGS in make.inc

```
DFLAGS = ... -D__FFTW3
```

12. Build the QE application that you need (e.g., "pw", "cp", or "all"):

```
$ make pw -j
```

# B

# THE $WSE_2$ INPUT DEFINITION FILE

```
 &CONTROL
                      title = 'MX2' ,
                calculation = 'scf' ,
               restart_mode = 'from_scratch' ,
                     outdir = './out/' ,
                 pseudo_dir = './' ,
                     prefix = 'MX2' ,
 /
 &SYSTEM
                      ibrav = 0,
                  celldm(1) = 1,
                        nat = 3,
                       ntyp = 2,
                    ecutwfc = 50.0 ,
                      nosym = .false. ,
                       nbnd = 23,
                occupations = 'smearing' ,
                    degauss = 0.01 ,
                   smearing = 'gaussian' ,
   starting_magnetization(1) = 0.00001,
   starting_magnetization(2) = 0.00001,
                   noncolin = .true. ,
                  angle1(1) = 0,
                  angle1(2) = 0,
                  angle2(1) = 45,
                  angle2(2) = 0,
                   lspinorb = .true. ,
                     london = .true. ,
 /
 &ELECTRONS
 /
CELL_PARAMETERS hexagonal
     5.500000000    3.175426480    0.000000000
     5.500000000   -3.175426480    0.000000000
     0.000000000    0.000000000   45.000000000
ATOMIC_SPECIES
    W  183.84000  W.RMR-pbe-TM.UPF
   Se   78.96000  Se.rel-pbe-n-nc.UPF
ATOMIC_POSITIONS crystal
Se       0.333333333   0.333333333  -0.070606112
Se       0.333333333   0.333333333   0.070606112
W        0.000000000   0.000000000   0.000000000
K_POINTS automatic
  12 12 1   1 1 0
```

# C

# CALLGRAPH (ALL ROUTINES)