Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Optimization of admission of patients to a hospital by using parallel algorithms on Intel Xeon Phi

Jan Kůrka

Supervisor: Ing. Libor Bukata
Supervisor–specialist: Ing. Přemysl Šůcha, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence
May 2016

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Jan Kůrka**

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Optimization of admission of patients to a hospital by using parallel algorithms on Intel Xeon Phi**

Guidelines:

1. Familiarize yourself with the Intel Xeon Phi architecture.
2. Setup a PC system for using Intel Xeon Phi cards and write a document summarizing installation procedure.
3. Explore existing works related to parallel algorithms for the optimization in health care. Based on your findings decide which algorithm is suitable for our purposes.
4. Implement the CPU version of the algorithm and evaluate the effectiveness and quality of the solutions on exiting datasets.
5. Implement the version for Intel Xeon Phi and carry out the benchmarks. Compare the implementation with the CPU version.
6. Compare your work with other existing works.
7. Summarize your work (survey, system configuration, proposed algorithms,...) and write it up in your diploma thesis.

Bibliography/Sources:

Peter Demeester, Wouter Souffriau, Patrick De Causmaecker, Greet Vanden Berghe, A hybrid tabu search algorithm for automatically assigning patients to beds, Artificial Intelligence in Medicine, Volume 48, Issue 1, January 2010, Pages 61-70, ISSN 0933-3657, http://dx.doi.org/10.1016/j.artmed.2009.09.001.

Sara Ceschia, Andrea Schaerf, Local search and lower bounds for the patient admission scheduling problem, Computers & Operations Research, Volume 38, Issue 10, October 2011, Pages 1452-1463, ISSN 0305-0548, http://dx.doi.org/10.1016/j.cor.2011.01.007.

Jim Jeffers, James Reinders, Intel Xeon Phi Coprocessor High Performance Programming, edited by Jim Jeffers, James Reinders, Morgan Kaufmann, Boston, 2013, ISBN 978-0-12-410414-3.

Diploma Thesis Supervisor: Ing. Libor Bukata

Valid until the end of the winter semester of academic year 2017/2018

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
L.S.
-4-

prof. Dr. Michal Pěchouček, MSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, April 21, 2016

Acknowledgements

I would like to thank my supervisors Ing. Libor Bukata and Ing. Přemysl Šůcha, Ph.D, who lead me through this project with great patience and encouragement. I am also grateful for the support of my family and friends.

Declaration

I declare that this thesis is my own work and that I have listed all the literature and publications used in accordance with *Metodický pokyn č. 1/2009 - O dodržování etických principů při přípravě vysokoškolských závěrečných prací.*

V Praze, 27. May 2016

Abstract

We have implemented the state-of-the art sequential algorithm for the Patient Admission Scheduling (PAS) problem and proposed two versions of parallel algorithms in this thesis. All algorithms are based on the local search metaheuristic called Simulated Annealing (SA). Two approaches to parallelization are compared. The first version runs multiple instances of SA which share solutions via a solution pool. The second version runs one SA with the moves evaluated in parallel. This version proved to be better than the first one because it produced more consistent results with average speedup 2.9 on CPU. The experiments indicate that the Intel Xeon Phi is not suitable for this problem since we were not able to utilize vectorization efficiently.

Keywords: scheduling, hospital, patient, admission, parallel algorithm, Xeon Phi, local search, simulated annealing

Supervisor: Ing. Libor Bukata

Abstrakt

Tato práce se zabývá problémem optimalizace přijímání pacientů do nemocnice známým jako Patient Admission Scheduling (PAS) problem. Implementovali jsme sekvenční algoritmus a navrhli dva paralelní algoritmy na jeho řešení. Základem je metaheuristika na lokální prohledávání nazývaná simulované žíhání. Porovnali jsme dva přístupy k paralelizaci. Jeden využíval paralelního běhu několika simulovaných žíhání, která si vyměňovala řešení přes sdílenou paměť. Druhý přístup používá jeden průběh simulovaného žíhání, kde vyhodnocuje více změn paralelně. Tato varianta se ukázala jako vhodnější, protože podávala konzistentní výsledky a na procesoru dosáhla průměrného zrychlení 2.9 oproti sekvenční verzi. Experimenty ukázaly, že Intel Xeon Phi není vhodný pro tento problém, protože jsme nebyli schopni využít efektivně vektorizaci.

Klíčová slova: plánování, nemocnice, pacient, přijímání pacientů, paralelní algoritmy, Xeon Phi, lokalní prohledávání, simulované žíhání

Překlad názvu: Optimalizace přijímání pacientů do nemocnice pomocí paralelních algoritmů na Intel Xeon Phi

Contents

Figures

Tables

Chapter 1

Introduction

Efficient planning is an important part of a high-quality health-care system. Better planning means lower costs for the hospital and better services for the patient. With increased efficiency, a hospital is able to serve more patients and patients can more likely get into rooms that suit their needs. That can lead to a greater comfort for the patients as well as higher income for the hospital for special rooms.

Available statistical data from the years 1995–2013 shows that health expenditures per capita in the Organisation for Economic Co-operation and Development (OECD) countries are growing [1] and the growth will continue with increasing life expectancy [2]. That suggests that hospitals will have to accommodate to higher demand for their services.

Figure 1.1: Patient (source: [3])

1.1 Patient Admission Scheduling (PAS)

An admission of patients into a hospital is a promising area which can become more effective using automated scheduling. Demeester et al. [4] were aware of that and introduced the PAS problem. After an extensive discussion with hospital staff and other relevant people, they formulated this problem as an assignment of patients to the beds in hospital with respect to the patients' needs and preferences. The needs and preferences are formulated as soft constraints and the objective is to optimize overall assignment of the patients. Each patient has a fixed admission day and discharge day, between these days they must be assigned a bed. The scheduling is performed for the entire hospital consisting of multiple departments with different specialisms and across all rooms with various capacity, specialisms and properties. This definition became a standard in this branch of research and most of the following literature uses or extends this model. It will also be used in this thesis.

Demeester et al. [4] tried to solve this problem by integer programming approach which proved to be too slow since this problem is NP complete, as Vancroonenburg et al. [5] proved. Using Tabu Search hybridized with token-ring and variable neighborhood descend they have been able to solve this problem in a reasonable time. They also created artificial instances based on realistic hospital situations, since the real-world data was hard to obtain due to privacy issues. These are available on the PAS website [6].

Bilgin et al. [7] followed up and used high-level hyper-heuristic approach to tackle two health care timetabling problems, the PAS and the nurse rostering. They compared various hyper-heuristics and concluded that the best one significantly outperforms previously mentioned tabu search [4] when focusing on PAS.

Ceschia and Schaerf [8] proposed a multi-neighborhood local search algorithm to solve this problem. Two metaheuristics were considered, the Simulated Annealing and Tabu Search. In the preliminary comparison, the Simulated Annealing performed better, it was therefore used and examined in more detail. Results were compared to the lower bounds computed using ILOG CPLEX [9] software and Integer Linear Programming (ILP) model with relaxed some integrality constraint(s) for variables. Comparison showed that the obtained results are quite close to the lower bound. It was observed that due to the nature of the problem and the instances, the best solutions have no transfers between beds. This allowed to reduce the search space to solutions with zero transfers.

Kifah and Abdullah [10] tried to use a different metaheuristic called adaptive non-linear Great Deluge (GD) algorithm for solving PAS problem. The algorithm is based on local search and accepts the worse solution if it is less than or equal to the water level. The standard GD uses linear decay rate which determines the decline of the water level. Kifah and Abdullah [10] employed adaptive changes to the water level and decay rate and reported positive effect of this improvement. From the four neighborhood moves they used, two are the same random moves as Ceschia and Schaerf [8] used and the other two aim for more precise focus on satisfying the soft constraints. However, the results are not as good as in Ceschia and Schaerf [8].

Range et al.[11] committed themselves to produce new best-known solutions. That was quite a challenge since the results in Ceschia and Schaerf [8], with which they are comparing, were close to the lower bound. The column generation approach was proposed for this task and fine-tuned so that for most of the small instances gives the best results, however the solver took more time to get the result. There is no dominant parameter setting that would yield strictly better results than the other settings, which limits its real-world application. To get competitive results for the larger instances this approach would need the further improvement.

Based on the original PAS problem Ceschia and Schaerf [12] proposed a new formulation called Dynamic Patient Admission Scheduling under Uncertainty (PASU) which introduced several real-world features including registration day, uncertainty of the length of stay and delaying the admission of the (non-urgent) patients. Registration day is the day the patient became known to the system, primarily their admission and discharge day. The actual admission day can be delayed but no more than given maximum for a patient dependent on her/his medical condition. For emergency patients, the registration day is equal to the admission day and delay is not possible. The overstay risk expresses the possibility that the patient will need an extra day to stay in the hospital and there is a corresponding penalty for that. Artificial instance generator of this problem was implemented and published as well as the benchmark instances [13] used when comparing ILP and local search approach.

1.2 Intel Xeon Phi

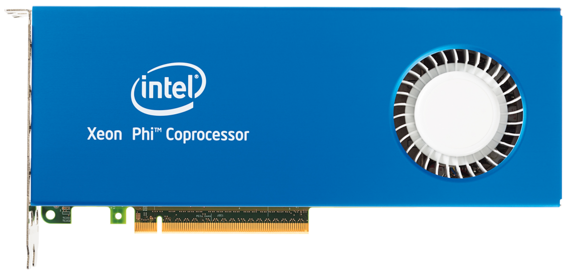

Figure 1.2: Intel Xeon Phi coprocessor (source: [14])

Intel Many Integrated Core (MIC) Architecture used in Intel Xeon Phi coprocessors (shown in Figure 1.2) is a relatively new and promising approach to the parallelization and high-performance computing. It directly competes with GPGPU approach, mainly with the NVIDIA CUDA which utilizes graphics cards for general-purpose computing. Intel MIC is a x86 compatible multiprocessor architecture that allows the same code to run on a coprocessor as runs on the CPU. That results in added comfort for programmer and fair comparison of the CPU and accelerated version, because optimizations for one piece of hardware will also reflect on the other.

There is almost no published research dealing with Intel Xeon Phi usage for solving combinatorial problems. One exception is the work of Melab et al. [15] who proposed a new data structure called Integer–Vector–Matrix (IVM) for solving large permutation problems using a parallel Branch-and-Bound algorithm. They showed that IVM is better suited for large number of threads compared to the traditional linked list, and is thus suited for Intel MIC architecture. The experiments performed on Intel Xeon Phi showed that Branch-and-Bound algorithm with IVM is about 10 times faster than with linked list.

1.3 Outline

This chapter presented a brief introduction, motivation and the literature relevant to this thesis. Chapter 2 elaborates the problem discussed in this thesis in more detail and shows a mathematical model. Chapter 3 contains description of the properties and architecture of Intel Xeon Phi coprocessor, its setup guide for Gentoo Linux and a micro-benchmark. After that, three algorithms are proposed. The sequential algorithm is in Chapter 4, Parallel I and Parallel II are in Chapter 5 and 6 respectively. The efficiency of the algorithms is compared on the Central Processing Unit (CPU) and Intel Xeon Phi in Chapter 7. The thesis is concluded in Chapter 8.

There are several appendices. The contents of the attached CD is described in Appendix A. After that, there is a list of acronyms in Appendix B and the used *notation is summarized in Appendix C.* The bibliography is at the end of the thesis.

Chapter 2

Problem Definition

In this chapter, we present detailed description of the PAS problem starting with required terminology and notation, followed by an overview of constraints and mathematical model. We employ a similar notation to Ceschia and Schaerf [8]. The PAS problem consists of the following features:

- **Day** $d \in \mathcal{D}$ is a unit of time and one day is considered one time-slot. The $\mathcal{D}$ denotes all days in planning horizon.
- **Planning horizon** is a number of consecutive days in the schedule.
- **Patient** $p \in \mathcal{P}$ is a person who needs a medical attention and have to spend time in hospital. $\mathcal{P}$ denotes the set of all patients. Each patient has the following features:
  - **Admission day** $\mathrm{AD}(p)$ is a day when the patient is admitted to the hospital (first day).
  - **Discharge day** $\mathrm{DD}(p)$ is a day when the patient is discharged from the hospital (next to the last day). $\mathcal{D}_p(p)$ denotes all days that patient p stays in hospital.
  - **Gender** according to which we can divide the patients into groups of women $\mathcal{P}_F$ and men $\mathcal{P}_M$.
  - **Age** $p_{age}(p)$
  - **Mandatory room property (MRP)** $p_{\mathrm{MRP}}(p, r_q)$ is a room property that the patient really needs.
  - **Preferred room property (PRP)** $p_{\mathrm{PRP}}(p, r_q)$ is a room property that the patient should have for his/her comfort.
  - **Specialism** $p_S(p)$ which corresponds with the medical treatment the patient requires.
  - **Room preference** $p_{\mathrm{RP}}(p)$ is a capacity of the room in which the patient wish to stay. Smaller rooms are usually charged extra in hospitals.

- **Room** $r \in \mathcal{R}$ where $\mathcal{R}$ denotes set of all available rooms for planning. Each room is assigned to exactly one department. They have the following features:
  - **Capacity** $r_c(r)$ which expresses how many beds are in it.
  - **Department** $r_z(r)$ to which the room r belongs to.
  - **Gender type** which specifies which gender should be accommodated in the room. There are 4 types:
    - F type means that only female patients are allowed.
    - M type means that only male patients are allowed.
    - D type (most common) means that only one (arbitrary) gender is allowed.
    - N type is a mixed gender room, both genders are allowed at the same time.
  - **Specialism** $\mathcal{S}_r(r)$ of the room
  - **Properties** $r_q \in \mathcal{R}_Q$ present in the room, where $\mathcal{R}_Q$ is set of all properties that can be in the rooms.
- **Beds**
- **Department** z is qualified for treatment in several specialisms $\mathcal{S}(z)$.
- **Transfer** is each change of bed during the patient's stay in hospital. It is undesirable and thus penalized.

The goal is to assign patients to the beds with respect to hard constraints:

- There can be at most one patient per bed per day.
- Each patient has to have assigned a bed for each day of his/her stay.

And violate the following soft constraints as little as possible:

- Room gender type restrictions should be fulfilled.
- Patients preferred and mandatory room properties should be present in the room.
- Department age restrictions should be fulfilled.
- Department and room specialism should correspond with the patients'.
- Transfers should be minimized.

2.1 Mathematical Model

2.1.1 Preprocessing Steps

As Ceschia and Schaerf [8] pointed out, the beds in each room are equivalent in terms of features and constraints therefore we can use patient-room assignments to simplify the problem. Resulting patient-room assignment must be then post-processed into the patient-bed assignment. It is important to avoid moving patient from bed to another bed inside a room and thus causing transfer. There always exists patient-bed assignment that never transfers patient from one bed to another in the same room if the patients are processed according to their first day in the room [8].

There are several patient-room constraints that can be precomputed into a joint matrix $C(p, r)$. It contains following penalties which are independent of the day and other patients:

- Static room gender (SRG) $f_{\text{SRG}}(p, r)$
- Mandatory room property (MRP) $f_{\text{MRP}}(p, r)$
- Preferred room property (PRP) $f_{\text{PRP}}(p, r)$
- Age constraint $f_{age}(p, r)$
- Room preference (RP) $f_{\text{RP}}(p, r)$
- Department specialism (DS) $f_{\text{DS}}(p, r)$
- Room specialism (RS) $f_{RS}(p, r)$

These constraints are described in more detail below. Weights of all soft constraints are specified in the Table 2.1.

$$\begin{aligned} C(p, r) = f_{\text{SRG}}(p, r) + f_{\text{MRP}}(p, r) + f_{\text{PRP}}(p, r) + f_{age}(p, r) + \\ + f_{\text{RP}}(p, r) + f_{\text{DS}}(p, r) + f_{RS}(p, r) \end{aligned} \tag{2.1}$$

Static room gender (SRG) constraint is applied only for room types F or M which means that the particular room can be occupied only by females or males.

$$f_{\text{SRG}}(p, r) = \begin{cases} w_{\text{SRG}}, & \text{iff room } r \text{ type does not correspond with patient gender,} \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

Mandatory room property (MRP) is denoted $r_{\text{MRP}}(r, r_q) \in \{0, 1\}$ where 1 (0) means that the room property r_q is present (not present) in the room r. Mandatory room property from point of view of a patient p is $p_{\text{MRP}}(p, r_q) \in \{0, 1\}$ where 1 (0) means the room property r_q is needed (not needed) by the patient p.

$$f_{\text{MRP}}(p, r) = w_{\text{MRP}} \cdot \sum_{r_q \in \mathcal{R}_Q} [(1 - r_{\text{MRP}}(r, r_q)) \cdot p_{\text{MRP}}(p, r_q)] \tag{2.3}$$

Description	Weight	Penalty
Patients in the rooms of type F and M do not have appropriate gender	w_{SRG}	5
Room is not equipped with mandatory room properties	w_{MRP}	5
Room is not equipped with preferred room properties	w_{PRP}	2
Age of the patients does not correspond to the department age limits	w_{age}	10
Patient room preference is not fulfilled	w_{RP}	0.8
Department specialism does not correspond to the patient needs	w_{DS}	1
Room specialism does not correspond to the patient needs	w_{RS}	1
All patients in the room of type D do not have the same gender	w_{RG}	5
Transfer of a patient	w_{Tr}	11

Table 2.1: Weights of soft constraints

Preferred room property (PRP) $r_{\text{PRP}}(r, r_q) \in \{0, 1\}$ is similar to MRP, however when MRP penalty for a particular room, patient and room property is issued, the PRP penalty is not applied there.

$$f_{\text{PRP}}(p, r) = w_{\text{PRP}} \cdot \sum_{r_q \in \mathcal{R}_Q} \left[(1 - r_{\text{MRP}}(r, r_q)) \cdot (1 - r_{\text{PRP}}(r, r_q)) \cdot p_{\text{PRP}}(p, r_q)\right], \tag{2.4}$$

Departments and by extension the rooms may have age constraints - lower bound $\text{LB}_{age}(r)$ and upper bound $\text{UB}_{age}(r)$. The penalty for not meeting the age constraint is following:

$$f_{age}(p, r) = \begin{cases} w_{age} & \text{iff } p_{age}(p) < \text{LB}_{age}(r) \text{ OR } p_{age}(p) > \text{UB}_{age}(r), \\ 0, & \text{otherwise.} \end{cases} \tag{2.5}$$

Room preference (RP) $p_{\text{RP}}(p)$ of a patient p is a type of a room according to its capacity $r_c(r)$ which the patient desire (there are single, double rooms etc.). The penalty is issued only if patient gets a room with greater capacity than she/he wants.

$$f_{\text{RP}}(p, r) = \begin{cases} w_{\text{RP}} & \text{iff } p_{\text{RP}}(p) < r_c(r), \\ 0, & \text{otherwise.} \end{cases} \tag{2.6}$$

Each room r belongs to a department $z = r_z(r)$ which is qualified for treatment in several specialisms $s \in \mathcal{S}(z)$. $\mathcal{S}(z)$ is set of specialisms of department z. The degree of specialism $d_S(z)$ of given department characterize how well it is prepared to treat patients with given specialism $p_S(p)$. The major specialism has degree 1, the minor has 2. For every patient that is not treated in department with his major specialism, the penalty proportional to degree-specialism difference $\Delta_{d_S}(z, s)$ is issued.

$$f_{\text{DS}}(p, r) = \Delta_{d_S}(r_z(r), p_S(p)) \cdot w_{\text{DS}} \tag{2.7}$$

$$\Delta_{d_S}(z,s) = \begin{cases} d_S(z) - 1, & \text{iff specialism } s \in \mathcal{S}(z), , \\ 2, & \text{otherwise.} \end{cases} \tag{2.8}$$

Similarly to the departments, each room is equipped for the certain specialisms $s \in \mathcal{S}_r(r)$. The degree of specialism is not applied here.

$$f_{RS}(p,r) = \begin{cases} 0, & \text{iff specialism } p_S(p) \in \mathcal{S}_r(r), \\ w_{\mathrm{RS}}, & \text{otherwise.} \end{cases} \tag{2.9}$$

2.1.2 Mathematical Model

The mathematical model we present in this section helps to demonstrate the PAS problem in detail. Finding optimal solution even for the smallest instance using ILP model and CPLEX proved to be very time-consuming. P. Demeester provided the optimal result for the first instance and it took over 60 hours of CPLEX computation [16]. We have also implemented our mathematical model in ILOG CPLEX [17], verified it and used it to compare results with our local search approach in the section 7.2.1.

The decision variables are following:

$$x(p,r,d) = \begin{cases} 1, & \text{iff patient } p \text{ is assigned to room } r \text{ on day } d, \\ 0, & \text{otherwise.} \end{cases} \tag{2.10}$$

$$t(p,r,d)) = \begin{cases} 1, & \text{iff patient } p \text{ is transferred from room } r \text{ on day } d, \\ 0, & \text{otherwise.} \end{cases} \tag{2.11}$$

$$f(r,d) \qquad \text{number of female patients in room } r \text{ on day } d \tag{2.12}$$

$$m(r,d) \qquad \text{number of male patients in room } r \text{ on day } d \tag{2.13}$$

Thanks to the preprocessing step described above, there are only three components in the objective function. Patient-room cost (PRC) F_{PRC} which contains cost using $C(p,r)$ matrix whose elements describe how is a room r generally suitable for a patient p. Room gender (RG) cost F_{RG} which covers room gender constraint of D type room (those which should be occupied by arbitrary one gender) and transfer cost F_{Tr} which includes the transfers of patients between rooms. Set of all D type rooms is denoted $\mathcal{R}_D$

$$F_{PRC} = \sum_{p\in\mathcal{P}, r\in\mathcal{R}, d\in\mathcal{D}_p} C_{p,r} \cdot x_{p,r,d} \tag{2.14}$$

$$F_{RG} = \sum_{r\in\mathcal{R}_D, d\in\mathcal{D}} w_{\mathrm{RG}} \cdot \min\left(f(r,d), m(r,d)\right) \tag{2.15}$$

$$F_{Tr} = \sum_{p\in\mathcal{P}, r\in\mathcal{R}, d\in\mathcal{D}} w_{\mathrm{Tr}} \cdot t_{p,r,d} \tag{2.16}$$

The objective is to minimize the function (2.17) subject to the following constraints: constraint (2.18) ensures that every patient is assigned to exactly one room for every day of his/her stay; constraint (2.19) makes sure that no room is overcrowded; constraints (2.20, 2.21) computes numbers of female $f(r,d)$ and male $m(r,d)$ patients for room gender cost; and finally constraint (2.22) computes number of transfers $t(p,r,d)$ for computing transfer cost.

$$\min \left(F_{\text{PRC}} + F_{\text{RG}} + F_{\text{Tr}}\right) \tag{2.17}$$

s.t.

$$\sum_{r \in \mathcal{R}} x(p,r,d) = 1, \quad \forall p \in \mathcal{P}, d \in \mathcal{D}_p(p) \tag{2.18}$$

$$\sum_{p \in \mathcal{P}} x(p,r,d) \leq r_c(r), \quad \forall r \in \mathcal{R}, d \in \mathcal{D} \tag{2.19}$$

$$f(r,d) = \sum_{p \in \mathcal{P}_F} x(p,r,d), \quad \forall r \in \mathcal{R}, d \in \mathcal{D} \tag{2.20}$$

$$m(r,d) = \sum_{p \in \mathcal{P}_M} x(p,r,d), \quad \forall r \in \mathcal{R}, d \in \mathcal{D} \tag{2.21}$$

$$t(p,r,d) \geq x(p,r,d) - x(p,r,d+1), \quad \begin{array}{l} \forall p \in \mathcal{P}, r \in \mathcal{R}, \\ d \in \{\text{AD}(p) \ldots \text{DD}(p) - 2\} \end{array} \tag{2.22}$$

Chapter 3

Intel Xeon Phi

In this chapter, the Intel Xeon Phi coprocessor is discussed. First its parameters, architecture and properties, then setup guide for Gentoo Linux. At the end of this section, the synthetic micro-benchmark is presented and used to compare compilers in terms of code optimizations and to compare CPU with Intel Xeon Phi.

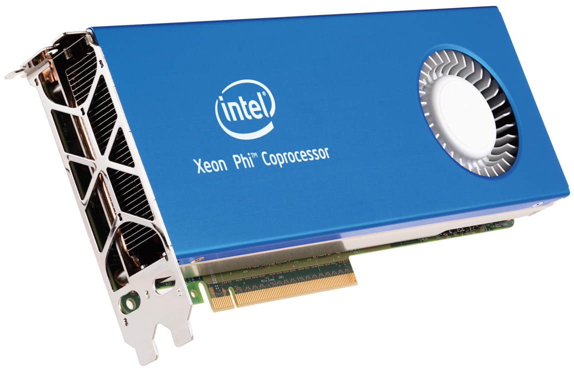

Figure 3.1: Intel Xeon Phi card (source: [14])

Number of cores	57–61
Core clock speed	1.1–1.238 GHz
Number of hardware threads per core	4
L1 Data Cache	32 kB
L1 Instruction Cache	32 kB
L2 Cache	512 kB
Memory	6–16 GB

Table 3.1: Intel Xeon Phi specifications (Knights Corner product line)[18]

Intel Xeon Phi is a coprocessor for high-performance computing based on Intel Many Integrated Core (MIC) architecture. The Knights Corner product line is available, which is the first generation of Intel's commercial MIC product. Summary of the basic specifications is shown in Table 3.1. It has over 50 cores connected by on-die bidirectional ring shown in Figure 3.3 and is connected via PCIe bus to a Intel Xeon processor, which is usually referred as the host. Communication on PCIe bus is performed via virtual TCP/IP connection allowing it to have its own IP address, be connected to the network and act as high-performance compute node. The coprocessor has its own Linux operating system and can run native Xeon Phi applications as well as heterogeneous ones where part executes on the host and part is offloaded to the coprocessor.

Intel MIC architecture is an x86 compatible multiprocessor architecture allowing us to run the code designated for CPU on Xeon Phi and vice versa. The core architecture shown in Figure 3.2 is based on Intel Pentium processor family which uses in-order instruction execution therefore there is no dynamic scheduling of instructions common in modern CPUs. There are also some new features and improvements, mainly Vector Processing Unit (VPU) supporting 512-bit SIMD instruction set called Intel Initial Many Core Instructions (IMCI). The VPU accommodates Extended Math Unit (EMU) which helps to vectorize the transcendental operations, such as log, square root and reciprocal function [19, 20]. Each core has 2 pipelines (U-pipe and V-pipe), therefore it can execute 2 instructions per cycle, however the pipelines are not equal. Not all instructions can be executed on the V-pipe, e.g. more complex vector instructions can be executed only by the U-pipe. Each core can run 4 hardware threads.

The second generation of Intel Xeon Phi coprocessors with codename Knights Landing should be available during the year 2016[21]. The architecture should be based on Intel Silvermont (Atom) architecture.

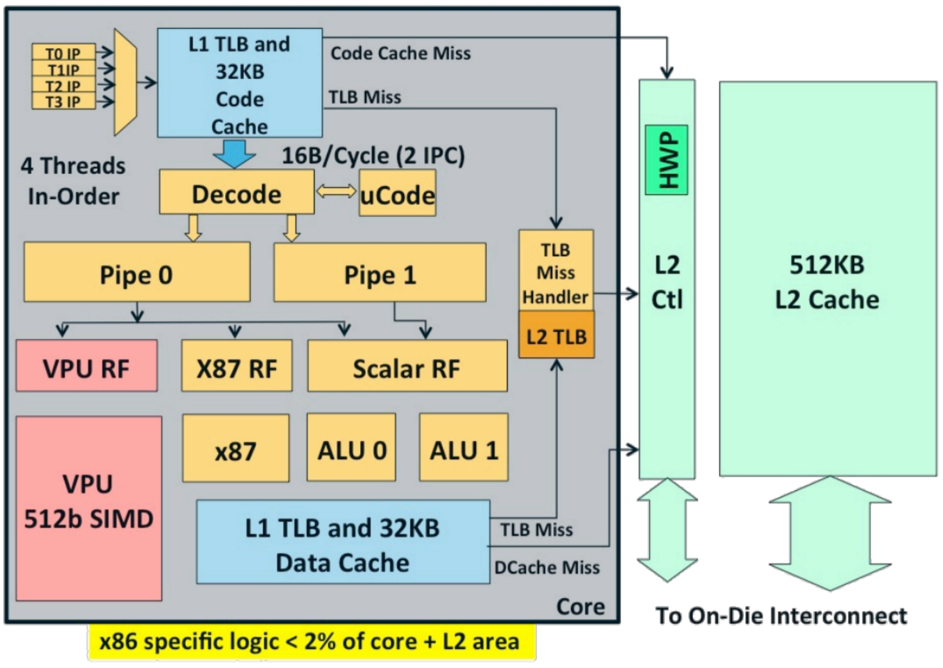

Figure 3.2: Intel Xeon Phi core architecture (source:[22])

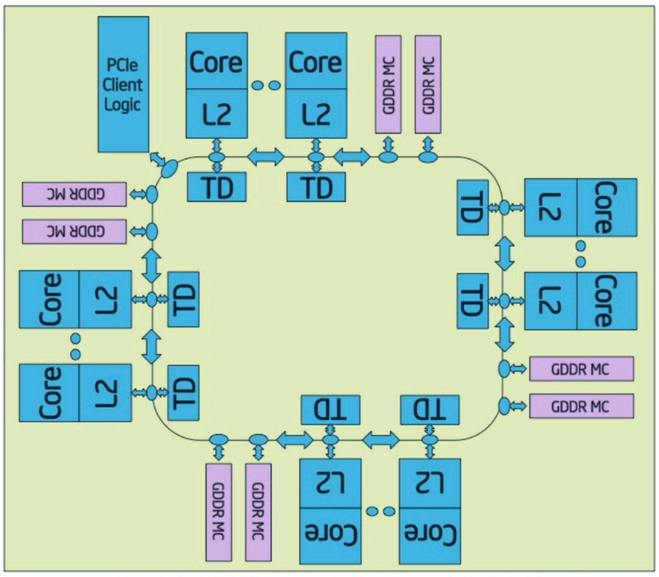

Figure 3.3: Intel Xeon Phi bidirectional ring interconnect (source: [22])

3.1 Installation Instructions for Gentoo Linux

Official support of Linux distributions is limited to only two distributions: SUSE Linux Enterprise Server (SLES) and Red Hat Enterprise Linux (RHEL). The installation procedure on other distributions is not always straightforward, therefore we are going to show how to install Xeon Phi on Gentoo Linux. We used Gentoo Linux with kernel version 3.12.49, Manycore Platform Software Stack (MPSS) 3.5.2 [23] and two Xeon Phi 31S1P coprocessors.

We have built upon the work of Anselm Busse who managed to run the Xeon Phi on Gentoo Linux. He has presented overlay on his GitHub [24] as well as brief tutorial [25]. We have extended his overlay by adding new ebuilds, updating it to newer MPSS 3.5.2, fixing some dependencies and embedding Python 2.7 dependency into the ebuilds so that they behaved correctly even when multiple Python versions are installed. New ebuilds mainly consist of packages supporting offloading. Since Intel supports only SLES and RHEL with a very limited number of kernel versions 2.6, 3.0, 3.10, 3.12; a few patches had to be written in order to make the ebuilds compilable. The changes usually deal with OS distribution check or a slightly different kernel API.

This installation guide has the following structure. First we show how to setup the host system and coprocessor, then how to install Intel Parallel Studio XE, after that you should be able to compile code using Intel compiler and use offloading mentioned in the third chapter. Finally we present a compiler comparison of GNU compiler, Intel compiler CPU version and Intel compiler offloaded version on synthetic micro-benchmark.

3.1.1 Setup System and Xeon Phi Coprocessor

Packages to Install

The following MPSS packages need to be installed in order to setup the host system to run and administrate the coprocessor.

- *sys-apps/mpss-daemon* - Daemon for starting/stopping Xeon Phi coprocessors + micctrl control utility
- *sys-apps/mpss-micmgmt* - Various tools to manage Intel Xeon Phi coprocessors (e.g. miccheck, micinfo, micflash)
- *sys-firmware/mpss-flash* - Bootloader and firmware images for flashing Intel Xeon Phi coprocessors
- *sys-kernel/mic-image* - Boot image for Xeon Phi card
- *sys-kernel/mic-rasmm-kernel* - RASMM kernel for Intel Xeon Phi card
- *sys-kernel/mpss-modules* - Kernel modules for a host
- *sys-libs/libmicmgmt* - C-library to access and update Intel Xeon Phi coprocessor parameters
- *sys-libs/libscif* - SCIF library for Intel MIC coprocessors
- *sys-libs/mpss-headers* - Header files for MIC Architecture
- *sys-devel/mpss-sdk-k1om* - SDK for Intel Xeon Phi
- *dev-util/gen-symver-map* - Utility for generating maps of symbols (System.map)

The following MPSS packages provide offloading support.

- *sys-libs/mpss-coi* - Library for offloading support for Intel Xeon Phi coprocessor
- *sys-libs/mpss-myo* - Shared memory library for MPSS stack

The ebuilds for the packages mentioned above are on the enclosed CD.

After Installation

When all the necessary packages are installed, we can load the kernel module `mic`. There can also be `mic_host` module present in the kernel, however we cannot use it. Make sure that `mic_host` is not loaded to avoid possible collisions. The kernel module can be loaded by the following command:

```
modprobe mic
```

It is useful to add `mic` module to /etc/conf.d/modules in order to load the module automatically during the booting process. The snippet of this file can look like this:

```
modules="nvidia-uvm msr nf_conntrack_ftp mic"
```

In the next sections we will use the `micctrl` control utility that allows us to control and administer the coprocessors. Note that the last argument of this tool is a list of Xeon Phi cards which allows us to select only some of the cards installed in the system. This argument is optional and when not specified, it applies the command to all available Xeon Phi cards. For example the

```
micctrl -s mic0 mic1
```

will check coprocessor 0 and 1 status. The coprocessors are hereinafter denoted "micX" where X stands for coprocessor number (e.g. mic0, they are numbered beginning from zero).

We can generate default configuration files in /var/mpss

```
micctrl --initdefaults
```

and start the daemon.

```
/etc/init.d/mpss start
```

The `micctrl` utility let us control and configure the coprocessor. First we use

```
micctrl -s
```

to check the coprocessor status. The card should be online. If it indicates the ready state, try to boot it.

```
micctrl -b
```

Note that booting takes a moment because Xeon Phi runs on Linux microkernel that has to be loaded to the card from the host system.

After that, it is needed to add users, setup network and possibly update the coprocessor flash as shown in the next sections.

Adding Users

Users can be added to /etc/passwd and /etc/shadow files on the coprocessor file system by `micctrl --useradd` command. The syntax is following

```
micctrl --useradd=<username> --uid=<uid> \
--gid=<gid> [--home=<dir>] [--comment=<string>] \
[--app=<exec>] [--sshkeys=<keyloc>] [MIC list]
```

It is necessary to specify a correct user and group IDs. These can be obtained by `id` command. Users should have valid RSA keys on the host system in their .ssh directory in order to be able to establish SSH connection with the coprocessor. To generate SSH key use `ssh-keygen`.

If the SSH keys are not added automatically or they are stored at different location, you can use `--sshkeys=<keyloc>` switch to explicitly specify the path.

Network Configuration

The mic kernel module has to be loaded before the network setup, see section After Installation 3.1.1. The communication between host and coprocessor is done via virtual TCP/IP network over PCIe bus. The Static Pair topology is the simplest configuration usually used in single host installations:

- Top two quads have default value "172.31".
- Third quad indicates coprocessor number (0, 1,...).
- Last quad: coprocessor gets "1", host "254".

More complex configurations are also possible. For example, it is possible to connect the card to the Internet. For more information see System Administration for the Intel Xeon Phi Coprocessor [26] and Configuring Intel Xeon Phi coprocessor inside a cluster [27] guides.

Each coprocessor is assigned to a separate subnet. The first three quads must match, they define the subnet of the particular coprocessor. Example of IP addresses for a host number X:

- Assigned address to the host for communication with coprocessor number X: 172.31.X.254
- Assigned address to the coprocessor number X for the communication with the host: 172.31.X.1

Running `micctrl --initdefaults` does not correctly initialize micX network interfaces on Gentoo, it has to be done manually. The host side of the coprocessors is defined in /etc/conf.d/net (note that we don't use RHEL configuration file /etc/sysconfig/network-scripts/ifcfg-micX). In /etc/conf.d/net you should define network interface micX for each installed coprocessor. A snippet of the file which defines the network interface for coprocessor number X is as follows.

```
config_micX=null #ensure that a network manager is not used
config_micX="172.31.X.254␣netmask␣255.255.255.0"
mtu_micX="64512"
```

After that, create a symbolic link from net.lo to net.micX.

```
cd /etc/init.d
ln -s net.lo net.micX
```

Start the network services. For OpenRC and default runlevel that means to execute:

```
rc-update add  net.micX default
```

The coprocessor configuration is located in file /etc/mpss/micX.conf. There can be assigned hostname to coprocessor and set the network configuration. The part of the file that we are interested in could look like this:

```
# Hostname to assign to MIC card
HostName your.domain.com

Network class=StaticPair micip=172.31.X.1 hostip=172.31.X.254 \
mtu=64512 netbits=24 modhost=yes modcard=yes
```

After editing this file run `micctrl --resetconfig` to instantiate the changes in the configuration files.

Check with `ifconfig` that all host network interfaces are correctly configured. There is snippet of what you might get:

```
micX: flags=67<UP,BROADCAST,RUNNING>  mtu 64512
inet 172.31.X.254  netmask 255.255.255.0  broadcast 172.31.X.255
inet6 fe80::4e79:baff:fe1c:1b4f  prefixlen 64  scopeid 0x20<link>
ether 4c:79:ba:1c:1b:4f  txqueuelen 1000  (Ethernet)
RX packets 6  bytes 468 (468.0 B)
RX errors 0  dropped 0  overruns 0  frame 0
TX packets 8  bytes 648 (648.0 B)
TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Update Flash

MPSS has specific firmware requirements which are stated in MPSS readme [28]. The `micinfo` utility provides detailed information about the coprocessor hardware and software and allows us to check whether the flash, SMC firmware and bootloader versions correspond to required ones. For MPSS 3.5.2 [23] these versions are required:

- Flash Version : 2.1.02.0391
- SMC Firmware Version : 1.17.6900
- SMC Boot Loader Version : 1.8.4326

Before flashing, it is very important to see Flash Issues & Remedies [29] and readme for MPSS [28]. There are described critical combinations that do not allow the standard way of flashing. In rare cases, these combinations can even lead to a card becoming stuck in a non-operational state.

Flash image files (with the suffix .rom.smc) are stored in the default location /usr/share/mpss/flash/. If they are elsewhere, you need to pass the path as an argument to the flash tool. There are few preconditions that should be satisfied before flashing.

- Current running version of Flash must be >=375, otherwise see Flash Issues & Remedies [28].
- Coprocessors must be in the ready state. You can achieve it by running `micctrl -rw`.

The `micflash` utility can also check the compatibility of the image file or save the current flash image into a file.

Use this command to update device X:

```
micflash -update -device X
```

When SMC boot loader update is necessary use this command to update flash and SMC of device:

```
micflash -update -device X -smcbootloader
```

After the flashing is complete perform a cold boot.

3.1.2 Installation of Intel Parallel Studio

Intel Parallel Studio XE [30] is available for students, educators, academic researchers, and open source contributors free of charge. After download, the install script install.sh will guide you through the process. For more details, see Intel Parallel Studio XE Installation guide. It depends on Python 2.7, you can select the correct Python version and run the installer by using the following command.

```
EPYTHON=python2.7 ./install.sh
```

The installer will warn you that the operating system is unsupported, however the installation should proceed successfully and Intel Compiler should work. So far we have not noticed any problems with Intel Compiler.

To initialize the environment variables for Intel Parallel Studio XE tools in a current shell run

```
source <install-dir>/bin/compilervars.sh intel64
```

You can add this command to the `~/.bashrc` file to make it permanent for a current user.

For more information see Intel Parallel Studio XE Installation Guide [31].

3.1.3 Offloading

Offload programming model refers to running main program on a host and offloading work to the coprocessor. This model fits well in the program where there are sections in which coprocessor parallel performance and vectorization can be utilized, as well as sections which benefit from CPU performance and architecture (e.g. out-of-order execution, bigger cache). However there are significant overhead costs for initialization and transferring data to a coprocessor that may incur a performance bottleneck.

On each coprocessor, there is an automatically created special user called micuser who executes work offloaded to the coprocessor.

For more information, see Offload Compiler Runtime for the Intel Xeon Phi Coprocessor [32] and Native and Offload Programming Models [33].

3.2 Benchmark

We have created synthetic micro-benchmark to test our system and to make some comparisons. Since version 2.22, the glibc contains libmvec [34], the vector math library which contains vector variants of scalar math functions implemented by using Single Instruction, Multiple Data (SIMD) instructions (e.g. SSE or AVX on x86 64 bit platform). Therefore, we were able to compare the performance with and without vectorization using GNU Compiler Collection (GCC) [35]. The second objective was to compare the offloaded and CPU versions using Intel C++ Compiler (ICPC) [36]. Third objective was to compare ICPC and GCC in terms of vectorization and code optimization only for the CPU.

The benchmark randomly initializes the array a of the length H and does specified number of attempts G to compute $\sin^2 a_i + \cos^2 a_i$; $i = 0...H - 1$. The input is H - the size of the array and G - the number of trials (number of repetitions of computation). The snippet of the benchmark source code follows, the whole source code is available on the enclosed CD.

```
#pragma omp parallel
...
#pragma omp for
for (uint32_t j = 0; j < G; ++j){
    #pragma omp simd aligned(a,b:AVX_ALIGNMENT)
    for (uint32_t i = 0; i < H; ++i) {
        // note: suffix 'f' to function names for floats
        //       to enable vectorization
        b[i] = powf(sinf(a[i]), 2.0f);
        b[i] += powf(cosf(a[i]), 2.0f);
    }
}
```

The following optimization flag were used:

```
g++ -march=native -ffast-math -fopenmp -O3 -std=c++11
icpc -march=native -openmp -O3 -std=c++11
```

The number of attempts G distinguished small and heavy workloads. The time in Table 3.2 is average of 10 runs. Hardware configuration was following: two Intel Xeon E5-2620 CPUs and two Xeon Phi 31S1P coprocessors with 64 GB of RAM.

	G	H	Average time
GCC 4.9.3 without vectorization	2^{20}	2^{10}	2.310 s
GCC 4.9.3 with vectorization	2^{20}	2^{10}	0.602 s
ICPC 16.0.0 CPU	2^{20}	2^{10}	0.381 s
ICPC 16.0.0 offloaded	2^{20}	2^{10}	0.686 s
ICPC 16.0.0 CPU	2^{30}	2^{10}	355 s
ICPC 16.0.0 offloaded	2^{30}	2^{10}	39.6 s

Table 3.2: Benchmark results

From Table 3.2, it is obvious that for a small workload, the offloading overhead significantly reduces the speedup, resulting in a worse time than pure CPU version compiled by ICPC. However, for the heavy workload the offloaded version is almost 9 times faster than ICPC CPU version. Comparing ICPC with GCC using vectorization, the ICPC is around 1.6 times faster on this benchmark. When focusing on GCC, the vectorization improved performance 3.8 times.

Chapter 4

Sequential Algorithm

The sequential algorithm we used is an implementation of M_0 solver proposed by Ceschia and Schaerf [8]. It is a multi-neighborhood local search algorithm based on SA.

4.1 Simulated Annealing

SA is a metaheuristic emulating a physical process used in metallurgy where the metals are slowly cooled from high-energy (high temperature) to the low-energy (low temperature) state in order to decrease defects. Gelatt et al. [37] and Černý[38] introduced it to find near-optimal solutions of the hard combinatorial problems such as traveling salesman problem. In each iteration of SA, a new neighbor solution is generated using moves and evaluated. All improving and some of the non-improving moves are accepted, the temperature is used to control the acceptance probability of the non-improving moves. During the run of the algorithm, the gradually decreasing temperature also decreases the probability that the non-improving move is accepted. The non-improving moves help to escape the local minima and allow more extensive search for the solution.

We used SA with geometric cooling, probabilistic acceptance, and two random moves which are described more thoroughly in the next section 4.2. The pseudocode in the Algorithm 0 illustrates the algorithm in greater detail. Cooling rate is denoted β and it is used to decrease the previous temperature T' into the next value T such that

$$T = \beta \cdot T'. \tag{4.1}$$

The probabilistic acceptance threshold is defined as

$$e^{-\frac{10 \cdot \Delta F}{T}} \tag{4.2}$$

where

$$\Delta F = F(S \circ m) - F(S) \tag{4.3}$$

is a difference between the value of objective function of a solution S with applied move m and value of solution S. The $\circ$ operator in $S \circ m$ means that the move m is applied at the solution S. The hard constraints on the capacity of rooms are converted into soft constraints with very high weight.

The number 10 in the expression 4.2 is motivated by the benefits of using integer algebra that are described later in the section 4.3. For better efficiency, the weights are multiplied by ten at the start of the algorithm and the resulting objective function divided by ten at the end to obtain correct objective function value. Note that this value relates to the original values of weights as stated in Table 2.1.

There are two loops in the algorithm - the inner loop performs N_{max} number of iterations on one value of temperature, the outer one decreases the temperature by multiplicating it by the cooling rate β. Each iteration consists of choosing random move (function RandomMove(S) in Algorithm 0) with random parameters. Then the move is evaluated, which yields ΔF. The improving move is applied every time, the non-improving is applied only with the probability defined by the acceptance criteria in Equation 4.2. At the end of the iteration, the best solution S_{best} is updated if needed.

There are four parameters that this algorithm needs:

- T_{min} is a stopping temperature of the SA.
- T_{init} is an initial temperature.
- β is a cooling rate.
- N_{max} is a number of iterations performed on one value of temperature (controls inner loop of the SA algorithm).

4.2 Moves

There are two moves used in this algorithm: Change Room (CR) and Swap Patients (SP). As mentioned above, the algorithm chooses in each iteration one of them randomly, but with a different probability. Ceschia and Schaerf [8] found out that the best probability for SP is 38% and for CR it is the remaining 62%.

Change Room (CR)

CR move picks one random patient and assigns him/her a new random room for the full stay. The image in Figure 4.1 illustrates an example of this move.

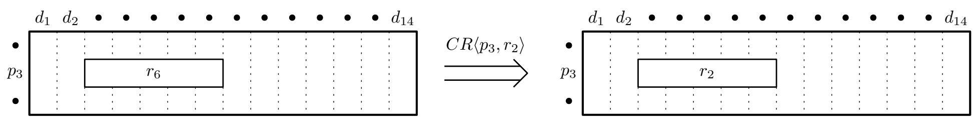

Figure 4.1: Change Room (CR) move example

Algorithm 0 Sequential algorithm

```
1:  $T \leftarrow T_{init}$;
2:  $S \leftarrow$ RandomState( );
3:  $S_{best} \leftarrow S$;
4:  while $T \geq T_{min}$ do
5:      $N \leftarrow 0$;
6:      while $N < N_{max}$ do
7:          $m \leftarrow$ RandomMove($S$);
8:          $\Delta F \leftarrow F(S \circ m) - F(S)$;          ▷ $S \circ m$ means $m$ applied at $S$
9:          if $\Delta F \leq 0$ then
10:             $S \leftarrow S \circ m$;
11:         else
12:             $r \leftarrow$ RandomNumber$(0, 1)$;        ▷ Random number from [0; 1)
13:             if $r \leq e^{-\frac{10 \cdot \Delta F}{T}}$ then
14:                 $S \leftarrow S \circ m$;
15:             end if
16:         end if
17:         if $F(S) < F(S_{best})$ then                ▷ Update the best solution
18:             $S_{best} \leftarrow S$;
19:         end if
20:         $N \leftarrow N + 1$;
21:     end while
22:     $T \leftarrow \beta \cdot T$;
23: end while
24: $return\ S_{best}$;
```

Swap Patients (SP)

SP move picks two random patients such that their stay overlap in at least one day. Than the patients exchange their rooms,the first patient gets the room of the second patient and vice versa. The Figure 4.2 illustrates an example of this move.

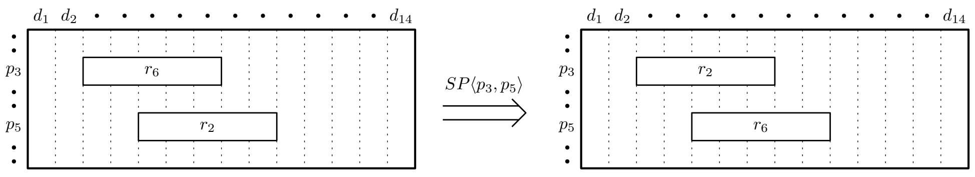

Figure 4.2: Swap Patients (SP) move example

4.3 Optimizations

There are several important optimizations that we used to create an efficient implementation. The most important one is a delta evaluation of the objective function. The changes to the solution made by certain move are analyzed and the ΔF difference in the objective function is computed without directly evaluating the whole schedule. That means it is not necessary to compute the full objective function when move is evaluated and since the moves are quite simple, the delta evaluation is significantly faster than the full evaluation of the schedule.

The vectorization was used to precompute values of exponential function and random numbers. To do it efficiently for the exponential function, the weights of the constraints presented in Table 2.1 were multiplied by 10 during the run of the solver. The objective function (as well as ΔF) than always has integer value for the weights in Table 2.1. That allows to simplify the indexing of the precomputed values by ΔF. At the end of the algorithm, the objective function is divided by 10 to obtain the correct value. The values of exponential function were precomputed for limited number of ΔF every time the temperature dropped (at the beginning of the outer loop in the Algorithm 0). This approach benefits from the high number of iterations per one temperature value. The values beyond the precomputed range can be computed directly or the closest precomputed value can be used instead of the exact value in case that there are small differences. In the case of the random numbers, the buffer for them is created and filled with random numbers. When all the numbers are used, it is refilled.

For the vectorization, the specialized library can be utilized. In this case the Intel Math Kernel Library (MKL) [39] was useful because it contains optimized and vectorized math and statistics functions including the vectorized random number generation and exponential function computation.

Chapter 5

Parallel Algorithm I

We propose two parallel algorithms for solving PAS problem. The first version is presented in this chapter, the second in the next (Chapter 6). Both versions are based on the sequential version presented in Chapter 4 and use Simulated Annealing (SA). There are more parameters than in the sequential version:

- T_{min} is a stopping temperature of the SA.
- T_{reop} is a SA initial temperature threshold. The β_{reop} rate is used below this threshold instead of β.
- T_{pool} is an initial temperature of random solutions in the solution pool.
- β is a cooling rate of the SA.
- β_{reop} is a cooling rate of the reoptimizations that have initial temperature below T_{reop}.
- N_{max} is a number of iterations performed on one temperature (controls the inner loop of the SA algorithm).
- $itersPerCheck$ defines the number of iterations between the checks of the solution pool. The checks are performed in order to load the better solution from solution pool to the SA.
- u_{min} is a threshold for temperature T_{min} which defines how big the relative difference between SA best solution S_{best} and the solution in pool must be to load the better solution from pool to SA and continue from that new solution. It is used to compute parameters of exponential function 5.1 as shown in Figure 5.2.
- u_{max} is a threshold similar to u_{min} but for temperature T_{pool}.
- μ is a mean value of the normal distribution of a random number generator.
- σ^2 is a variance of the normal distribution of a random number generator.
- $poolSize$ is the size of the solution pool.

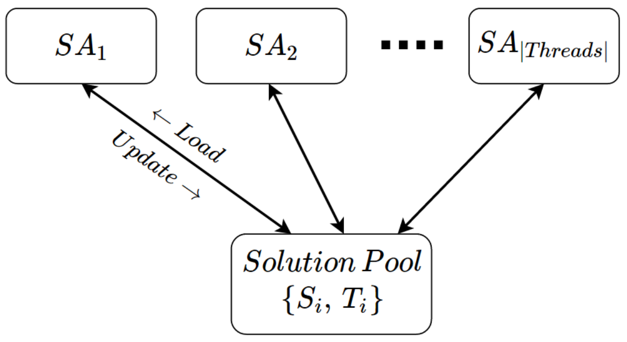

Figure 5.1: High-level scheme of the Parallel Algorithm I

This first approach utilizes multiple SA runs in parallel as depicted in Figure 5.1 or described in more detail in Algorithm 1. The threads share solutions via solution pool of the the size $poolSize$. Each thread repeatedly chooses a random solution out of the solution pool and run the SA algorithm (function RUNSA in Algorithm 2) on them until one of the stopping criteria is met, either timeout or the requested quality is achieved. The quality criteria was introduced for the purpose of comparison with the existing algorithms. When applying to unknown instances, there is of course no knowledge of what quality is achievable; in this case, the timeout has to suffice.

Each solution in the solution pool is assigned a temperature at which the solution was uploaded to the pool. Meaning that when SA updates the solution in the pool, it saves its temperature as well. The SA algorithm needs an initial temperature which is based on temperature $T[i]$ of loaded solution $S[i]$ in the pool, and then raised by an absolute value of random number from normal distribution with mean μ and variance σ (see line 9 in Algorithm 1).

There are some modifications of the SA algorithm compared to the sequential version, mainly with regards to the solution pool. The pseudocode of it is shown in Algorithm 2. When the solution S_{best} in SA achieves better quality than the one saved in the pool, the algorithm tries to update the solution pool (function UPDATE in Algorithm 3). There, it checks whether the solution in the pool is still worse than the one found by SA. If it is, the pool is updated.

The SA can load a better solution from the solution pool during its run (function LOAD in Algorithm 4). The interval of checking is controlled by $itersPerCheck$ parameter that defines number of iterations between the checks. The load happens only if the difference of the SA best solution S_{best} and solution from pool is higher than a threshold given by exponential function (depicted in Figure 5.2)

$$K \cdot e^{A \cdot T} \tag{5.1}$$

where coefficients A and K are computed from parameters u_{min} and u_{max} in the following way:

$$A = \log\left(u_{max}/u_{min}\right)/(T_{pool} - T_{min}) \tag{5.2}$$

$$K = u_{min}/e^{A \cdot T_{min}} \tag{5.3}$$

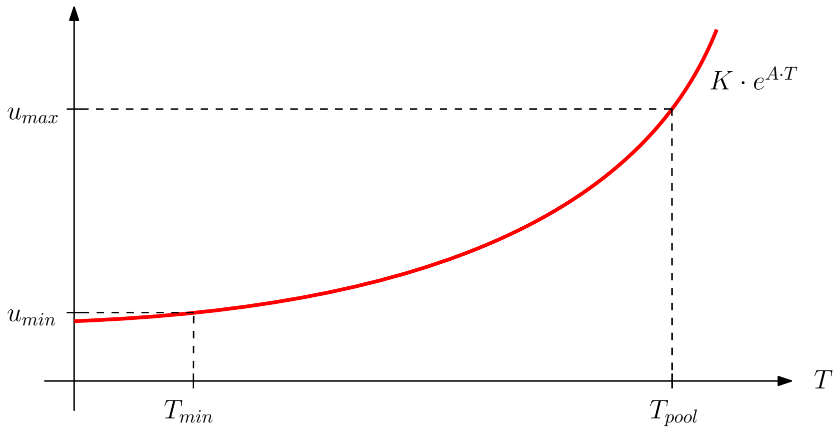

Figure 5.2: Exponential function

Algorithm 1 Version I of the parallel algorithm

```
1:  $A \leftarrow \log(u_{max}/u_{min})/(T_{pool} - T_{min})$;
2:  $K \leftarrow u_{min}/e^{A \cdot T_{min}}$
3:  for $i \leftarrow 0 \ldots poolSize - 1$ do                ▷ Initialize the solution pool
4:      $S[i] \leftarrow$ RandomState( );
5:      $T[i] \leftarrow T_{pool}$;
6:  end for
7:  while (!$timeout$ || !$qualityAchieved$) do                    ▷ Parallel section
8:      $idx \leftarrow$ RandomInteger($0, poolSize$);
9:      $T_{init} \leftarrow T[idx] + |$RandomFrom($N(\mu, \sigma^2)$)$|$;
10:     if $T_{init} < T_{reop}$ then
11:         $S_{out} \leftarrow$ RunSA($T_{init}, T_{min}, \beta_{reop}, idx, itersPerCheck$);
12:     else
13:         $S_{out} \leftarrow$ RunSA($T_{init}, T_{min}, \beta, idx, itersPerCheck$);
14:     end if
15:     Update($idx, S_{out}, T_{min}$)
16: end while
17: return BestSolutionFromPool( );
```

Algorithm 2 RunSA function

```
function RunSA($T_{init}, T_{min}, \beta, idx, itersPerCheck$)
    $S \leftarrow S[idx]$;
    $T \leftarrow T_{init}$;
    $S_{best} \leftarrow S$;
    while $T \geq T_{min}$ do
        $N \leftarrow 0$;
        while $N < N_{max}$ do
            $m \leftarrow$ RandomMove($S$);
            $\Delta F \leftarrow F(S \circ m) - F(S)$;        ▷ $S \circ m$ means $m$ applied at $S$
            if $\Delta F \leq 0$ then
                $S \leftarrow S \circ m$;
            else
                $r \leftarrow$ RandomNumber$(0, 1)$;
                if $r \leq e^{-\frac{10 \cdot \Delta F}{T}}$ then
                    $S \leftarrow S \circ m$;
                end if
            end if
            if $F(S) < F(S_{best})$ then        ▷ Update the best solution
                $S_{best} \leftarrow S$;
            end if
            if $F(S) < F(S[idx])$ then
                Update($idx, S_{best}, T$);
            end if
            if $(N \bmod itersPerCheck) == 0$ then
                $S \leftarrow$ Load($idx, S_{best}, T$);
            end if
            $N \leftarrow N + 1$;
        end while
        $T \leftarrow \beta \cdot T$;
    end while
    return $S_{best}$;
end function
```

Algorithm 3 Update function

```
function Update($idx, S_{best}, T$)
    if $S_{best} < S[idx]$ then
        lock($S[idx]$);
        $S[idx] \leftarrow S_{best}$;
        $T[idx] \leftarrow T$;
        unlock($S[idx]$);
    end if
end function
```

Algorithm 4 Load function

```
1: function LOAD($idx, S_{best}, T$)
2:     if $((F(S_{best}) - F(S[idx]))/F(S[idx])) > K \cdot e^{A \cdot T}$ then
3:         if $trylock(S[idx])$ then
4:             $S_{out} \leftarrow S[idx]$;
5:             $unlock(S[idx])$;
6:         end if
7:     end if
8:     $return\, S_{out}$;
9: end function
```

Chapter 6

Parallel Algorithm II

The second version of the parallel algorithm runs only one instance of simulated annealing, however, the moves are evaluated in parallel. It still preserves the form of the SA algorithm used in sequential version, which is apparent from simplified diagram in Figure 6.1. The pseudocode in the Algorithm 5 and 6 covers it in a greater detail. The parallelization is based on the efficient representation of the solution and conflict detection. The conflict occurs when one thread changes the part of the solution that another thread is working on.

The solution S consists of the set of room-assignments R_i where patients are assigned to the particular room number i. At the start, the solution is randomly initialized and each thread sets its temperature to the initial value. That is followed by iterating through inner and outer loops as in sequential algorithm in chapter 4. Inside the inner loop, the algorithm starts by choosing a random move (using function RandomMove). Each move operates only on two room-assignments from the solution: R_j and R_k, which limits the conflicts between threads. There is a counter of changes for each room-assignment ($changesCounter[R_i]$) that keeps track of modifications in the rooms and helps to identify the conflicts between threads. After the move and its parameters are chosen, the values of the counter of changes are stored and the move is evaluated. The criteria for accepting the moves are the same as in sequential version, however, the application of the move onto the solution is more complicated, as shown in function ApplyMove in Algorithm 6.

When applying a move, both room-assignments R_j and R_k have to be locked. If the locking is not successful, the current move is thrown away and new move is evaluated. In the case that the locking is successful, the counter of changes is compared with the saved values to check whether another thread did not changed the room-assignments intended for update. When the counter of changes is equal to the saved values, there is no conflict, the move is applied and the counter of changes is incremented. When a conflict occurs, the move cannot be applied in order to keep the solution consistent. In this case, the move is discarded and the thread resume with the next iteration. Note that since the moves are simple and the delta evaluation is efficient, it is better to continue to work rather than blocking the thread or resolving the conflicts.

Algorithm 5 Version II algorithm

```
 1: $S \leftarrow$ RandomSolution( );
 2: $T \leftarrow T_{init}$;                                      ▷ Parallel section
 3: while $T \geq T_{min}$ do
 4:     $N \leftarrow 0$;
 5:     while $N < N_{max}$ do
 6:         $m, R_j, R_k \leftarrow$ RandomMove($S$);
 7:         $c_j \leftarrow changesCounter[R_j]$;                 ▷ Atomic read
 8:         $c_k \leftarrow changesCounter[R_k]$;                 ▷ Atomic read
 9:         $\Delta F \leftarrow F(S \circ m) - F(S)$;   ▷ $S \circ m$ means $m$ applied at $S$
10:         if $\Delta F \leq 0$ then
11:             ApplyMove($m, S, R_j, R_k, c_j, c_k$);
12:         else
13:             $r \leftarrow$ RandomNumber($0, 1$);   ▷ Random number from [0; 1)
14:             if $r \leq e^{-\frac{10 \cdot \Delta F}{T}}$ then
15:                 ApplyMove($m, S, R_j, R_k, c_j, c_k$);
16:             end if
17:         end if
18:         $N \leftarrow N + 1$;
19:     end while
20:     $T \leftarrow \beta \cdot T$;
21: end while
22: $return\, S$;
```

Algorithm 6 ApplyMove function

```
 1: function ApplyMove($m, S, R_j, R_k, c_j, c_k$ )
 2:     if $!trylock(R_j)$ then
 3:         $return$;
 4:     end if
 5:     if $!trylock(R_k)$ then
 6:         $unlock(R_j)$;
 7:         $return$;
 8:     end if
 9:     if $changesCounter\,[R_j] \neq c_j || changesCounter\,[R_k] \neq c_k$ then
10:         $unlock(R_k)$;
11:         $unlock(R_j)$;
12:         $return$;
13:     end if
14:     $++changesCounter\,[R_j]$;
15:     $++changesCounter\,[R_k]$;
16:     $S \leftarrow S \circ m$;
17:     $unlock(R_k)$;
18:     $unlock(R_j)$;
19: end function
```

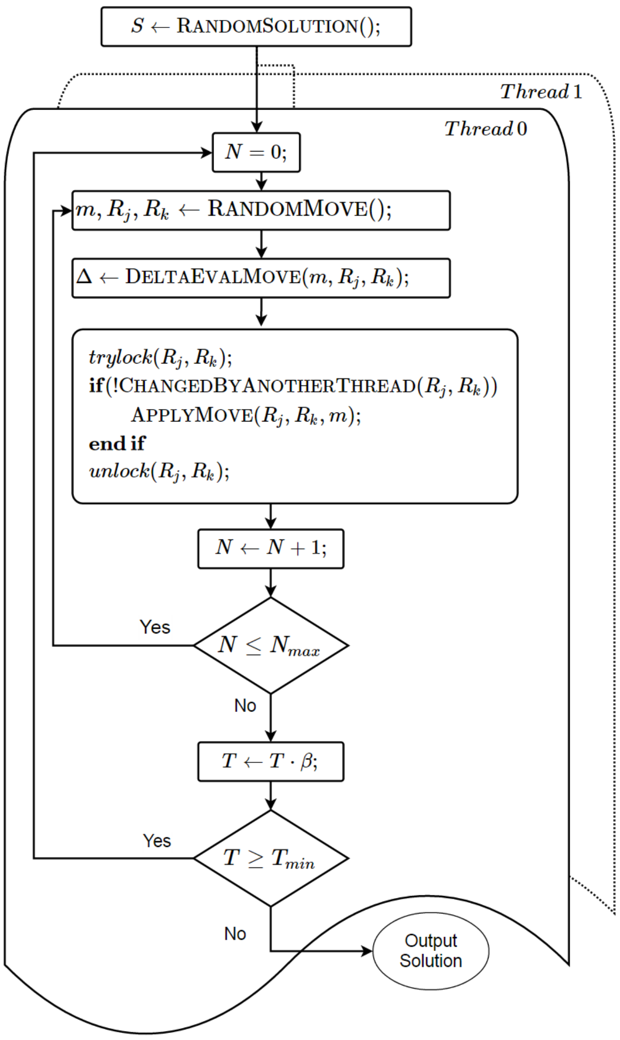
$S \leftarrow$ RANDOMSOLUTION();
Thread 1
Thread 0
$N = 0$;
$m, R_j, R_k \leftarrow$ RANDOMMOVE();
$\Delta \leftarrow$ DELTAEVALMOVE(m, R_j, R_k);
trylock(R_j, R_k);
if(!CHANGEDBYANOTHERTHREAD(R_j, R_k))
APPLYMOVE(R_j, R_k, m);
end if
unlock(R_j, R_k);
$N \leftarrow N + 1$;
Yes
$N \leq N_{max}$
No
$T \leftarrow T \cdot \beta$;
Yes
$T \geq T_{min}$
No
Output Solution

Figure 6.1: High-level scheme of the Parallel Algorithm II

Chapter 7

Experiments

In this chapter, the experiments and their results are presented. First, the datasets and the hardware configuration are introduced. Then, the experiments which are divided into two parts. First part presents the raw performance via the number of delta evaluations per second. The second focuses on the quality of the solutions.

The datasets for the PAS problem were publicly available on PAS website [6], as of 25. 4. 2016, they are unfortunately not available. Thanks to the Peter Demeester the offline copy is on the enclosed CD. On the website, there are 12 standard datasets which have been used in the experiments. There is also a validator application that processes the output file with scheduled patients and prints detailed analysis of constraints that have been broken and penalty for that. That significantly helps to ensure that all constraints are understood well, and to minimize the room for error. The website contains a detailed description of the dataset format and a brief description of the penalty calculation and validator.

The experiments were performed on a computer with two Intel Xeon E5-2620 CPUs, 64GB of memory and Intel Xeon Phi 31S1P coprocessor. Because the decisions of the algorithm depend on probability, the presented results are average values over 20 runs. The results were checked by the validator.

7.1 Delta evaluations per second

The number of delta evaluations per second represents the number of moves evaluated in one second. The evaluation of the schedule is one of the most computationally demanding parts of the algorithm, it is therefore an appropriate metric to measure the raw performance of the proposed algorithms on different hardware.

	Ceschia&Schaerf	Sequential	Parallel I	Parallel II
CPU	$1.7 \cdot 10^6$ ⊛	$5.2 \cdot 10^6$	$72.9 \cdot 10^6$	$27.9 \cdot 10^6$
Intel Xeon Phi	-	$0.2 \cdot 10^6$	$22.2 \cdot 10^6$	$5.6 \cdot 10^6$

⊛ Note that Ceschia and Schaerf[8] used different hardware.

Table 7.1: Number of delta evaluations per second

From the Table 7.1 is clear that the performance of Intel Xeon Phi is in this case not comparable with the two CPUs, since we have not been able to efficiently employ vectorization in this problem. The poor performance on the sequential version also comes from the simple core architecture and lower frequency in comparison with the modern CPU.

7.2 Quality and time

The following experiments were performed on all 12 public datasets. The ID column in the tables contains their ordinal number.

7.2.1 Sequential version

The first Table 7.2 compares sequential algorithm proposed by Ceschia and Schaerf [8] with our implementation and with ILP model created in ILOG CPLEX [17]. The ILP model is based on the mathematical model presented in Chapter 2. Both sequential algorithms used same parameters that Ceschia and Schaerf [8] identified as the best:

- $T_{min} = 0.85$
- $T_{init} = 114.81$
- $\beta = 0.9999$
- $N_{max} = 13653$

ID	Ceschia&Schaerf	Sequential		ILOG CPLEX	
	Quality	Quality	Time [s]	Quality	Time [s]
1	666	666	129	788	7200
2	1151	1150	143	3345332	7200
3	787	787	134	7462	7200
4	1191	1193	137	20007	7200
5	632	633	130	630	7200
6	811	812	129	12333	7200
7	1216	1222	106	1216	7200
8	4192	4189	117	4583	7200
9	22053	22372	113	90506	7200
10	8261	8281	134	49965	7200
11	12106	12161	155	153911	7200
12	23969	24114	156	243832	7200

Table 7.2: Sequential and CPLEX versions

The quality of both implementations is comparable which indicates that we have been successful with the reconstruction of the of the algorithm from available information and Table 7.1 shows that the implementation is efficient.

Comparing with ILP model, the CPLEX has not been able to find optimal solution for any of the instances when given 2 hours of computational time for each instance. Peter Deemester was able to find an optimal solution for one of the smallest datasets. According to him [16], it took over 60 hours of CPLEX computational time, presumably on different hardware.

7.2.2 Parallel versions on CPU

Table 7.3 compares the two versions of the parallel algorithm. The parameters were tuned so that the quality of the solutions was approximately the same as in the sequential version. Unfortunately, we were not able to tune the parameters of Parallel I algorithm to perform well on all instances. It had good results mostly on the smaller instances, and the results for the best setting found are presented in Table 7.3 for CPU and 7.4 for Intel Xeon Phi.

Settings for the CPU:

Parallel I algorithm

- $T_{min} = 0.85$
- $T_{reop} = 10$
- $T_{pool} = 114.81$
- $\beta = 0.997$
- $\beta_{reop} = 0.999$
- $N_{max} = 10000$
- $itersPerCheck = 100$
- $u_{min} = 0.2$
- $u_{max} = 2$
- $\mu = 6$
- $\sigma^2 = 1$
- $poolSize = 1$

Parallel II algorithm

- $T_{min} = 1$
- $T_{init} = 114.81$
- $\beta = 0.9991$
- $N_{max} = 10000$

From Table 7.3 is apparent that the parallelization was successful on the CPU. The Parallel I version gives very good results on smaller instances with average speedup 4.8 on them. On the large instances, the performance is worse than the sequential version. This version proved to be difficult to tune for large instances and the Parallel II version showed better results there.

The Parallel II version yielded much more consistent results than the first version, with average speedup of 2.9. The speedup on all instances is quite close to the average, this approach therefore seems like a better choice, especially for large instances. When solving the unknown instances, it is

ID	Sequential		Parallel I			Parallel II		
	Quality	Time[s]	Quality	Time[s]	Speedup	Quality	Time[s]	Speedup
1	666	129	666	26	5.0	666	57	2.3
2	1150	143	1151	126	1.1	1149	44	3.3
3	787	134	787	34	3.9	785	49	2.8
4	1193	137	1193	66	2.1	1192	45	3.0
5	633	130	633	10	13.4	632	66	2.0
6	812	129	812	36	3.6	809	48	2.7
7	1222	106	1225	117	0.9	1226	38	2.8
8	4189	117	4255	180∗	-	4195	39	3.0
9	22372	113	23910	180∗	-	22278	29	3.9
10	8281	134	8610	180∗	-	8353	50	2.7
11	12161	155	13070	180∗	-	12290	53	2.9
12	24114	156	26036	180∗	-	24133	50	3.1

* timeout

Table 7.3: Comparison of parallel versions on CPU

better not to rely on the size of an instance and know that the approach would work well on all sizes.

7.2.3 Parallel versions on Intel Xeon Phi

Because only negligible part of the code of both parallel versions is sequential, the solvers were compiled as a native application for Intel Xeon Phi. It therefore runs solely on the coprocessor. The results are compared with the sequential CPU version in Table 7.4. The settings were tuned to perform on Intel Xeon Phi in the best possible way, for Parallel I algorithm:

- $T_{min} = 0.85$
- $T_{reop} = 10$
- $T_{pool} = 114.81$
- $\beta = 0.995$
- $\beta_{reop} = 0.997$
- $N_{max} = 10000$
- $itersPerCheck = 100$
- $u_{min} = 0.2$
- $u_{max} = 2$
- $\mu = 6$
- $\sigma^2 = 1$
- $poolSize = 1$

and for Parallel II algorithm:

- $T_{min} = 1$
- $T_{init} = 114.81$
- $\beta = 0.9991$
- $N_{max} = 10000$.

ID	Sequential CPU		Parallel I		Parallel II	
	Quality	Time [s]	Quality	Time [s]	Quality	Time [s]
1	666	129	666	191	668	188
2	1150	143	1151	413	1162	153
3	787	134	787	163	789	179
4	1193	137	1193	199	1212	164
5	633	130	633	106	633	208
6	812	129	812	219	817	162
7	1222	106	1228	495	1317	128
8	4189	117	4272	600$*$	4330	137
9	22372	113	24038	600$*$	23553	120
10	8281	134	8729	600$*$	8423	225
11	12161	155	13292	600$*$	12447	248
12	24114	156	26395	600$*$	24392	229

* timeout

Table 7.4: Comparison of parallel versions on Intel Xeon Phi

From Table 7.4 is apparent that the Intel Xeon Phi performed worse than sequential CPU version. That happened mainly because we were not able to utilize vectorization efficiently. There was no room for that in delta evaluation. If it was possible the results would have been much more optimistic. From Table 7.1, it is clear that the cores in Intel Xeon Phi are much slower when compared to the state-of-the-art CPUs with more advanced architecture, and that the vectorization is necessary in order to gain a significant speedup.

7.3 Progress of the algorithms

The graphs in Figure 7.1 and 7.2 illustrate the run of the sequential algorithm. Figure 7.1 shows the progress of the active solution cost ($F(S)$) and best solution cost ($F(S_{best})$) during the execution of SA algorithm. The active solution cost is oscillating due to the SA ability to accept even non-improving moves, however, the overall tendency is decreasing and oscillating mitigates with decreasing temperature. The temperature is plotted in Figure 7.2.

Similar graphs are presented for one thread of the Parallel I algorithm in Figures 7.3 and 7.4. The difference in the run of the algorithm is apparent. In Parallel I, there is a number of consecutive SA runs. The first is starting with high temperature and then the reoptimizations start with low temperature as depicted in graph in Figure 7.4. The increase of temperature at the start of each reoptimization is bound to the increase in active solution cost in Figure 7.3. That are the parts with increased acceptance of non-improving solutions which help to escape the local minima and allow to search more extensively.

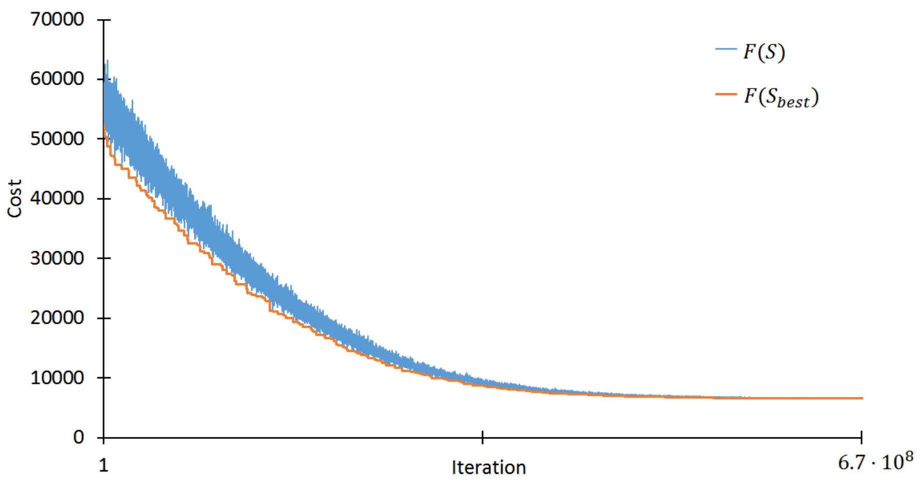

Figure 7.1: Sequential algorithm - progress of the active (S) and best solution (S_{best}) cost on dataset1

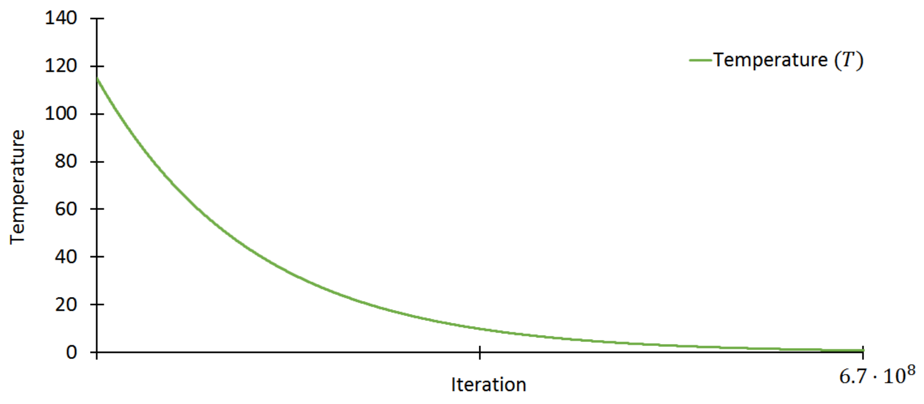

Figure 7.2: Sequential algorithm - temperature during the run on dataset1

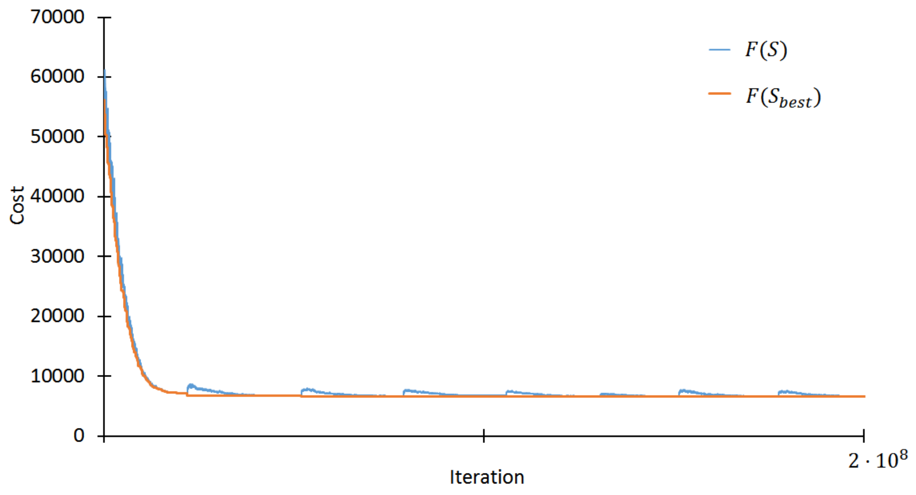

Figure 7.3: Parallel I algorithm - progress of the active (S) and best solution (S_{best}) cost of one thread on dataset1

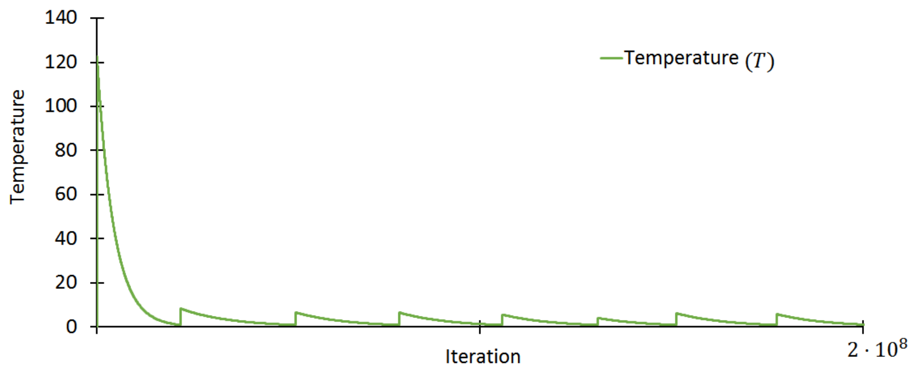

Figure 7.4: Parallel I algorithm - temperature during the run on dataset1

Chapter 8

Conclusion

We have implemented the state-of-the art sequential algorithm for the Patient Admission Scheduling (PAS) problem and proposed two versions of parallel algorithm. The algorithm is based on the local search metaheuristic called Simulated Annealing (SA). Our implementation of the sequential algorithm is comparable with the one proposed by Ceschia and Schaerf [8] in terms of quality and efficiency. Publicly available datasets were used to compare our results with others and the available validator was utilized for verifying the results.

We tried two approaches to parallelization. The Parallel I version runs multiple instances of SA each in its own thread. The threads share solution via solution pool, from which they could load the solution when starting the SA or when deviating to overly poor solutions comparing with the pool. During the run, the treads also update the solution pool when they find a better solution. The Parallel II version runs only one SA, however the moves are evaluated in parallel. The parallelization lies on the effective data representation and conflict detection.

The experiments were performed on both CPU and Intel Xeon Phi coprocessor. Parallelization on CPU was successful resulting in significant speedup. The Parallel I version proved to be hard to tune for large instances and we were not able to find parameters that would perform well on all instances. We have been able to tune it for small instances on which we obtained average speedup of 4.8. The results of Parallel II version were much more consistent, giving an average speedup of 2.9 over all instances.

Focusing on Intel Xeon Phi coprocessor, we have been able to successfully set it up in Gentoo Linux, despite the fact that this operating system is not officially supported by Intel. The experiments on Intel Xeon Phi suggested that it is not suitable for this problem. One of the most important features of the coprocessor is its vector processing unit, and without vectorization, it was not able to compete with modern CPUs on this problem. The key to the efficient implementation was delta evaluation of solutions, which means that the whole schedule was not evaluated each time the change to the solution was calculated; only the changes were analyzed. Unfortunately, the delta evaluation cannot be efficiently vectorized and that undermined the benefits of Intel Xeon Phi.

Appendix A

Contents of the attached CD

Directory	**Content description**
datasets	PAS datasets and validator
micro-benchmark	Micro-benchmark from section 3.2
parallel-I	Source code of the Parallel I version of algorithm.
parallel-II	Source code of the Parallel II version of algorithm.
pas-website	Offline copy of PAS website where datasets and validator are described.
post-processing-tool	Tool for postprocessing of patient-room assignment into the patient-bed assignment.
sequential	Source code of the sequential version of algorithm.
thesis	This thesis in PDF format
xeon-phi-ebuilds	Ebuilds for setup of Intel Xeon Phi on Gentoo Linux.

Appendix B
Acronyms

Notation	Description	Page List
CPU	Central Processing Unit	3, 4, 11, 12, 35–37, 39, 40, 43
CR	Change Room	22
DS	department specialism	7, 51, 53
EMU	Extended Math Unit	12
GCC	GNU Compiler Collection	19, 20
GD	Great Deluge	2
GPGPU	general-purpose computing on graphics processing units	3
ICPC	Intel C++ Compiler	19, 20
ILP	Integer Linear Programming	2, 3, 9, 36
IMCI	Initial Many Core Instructions	12
IVM	Integer–Vector–Matrix	4
MIC	Many Integrated Core	3, 4, 11, 12, 14
MKL	Math Kernel Library	24
MPSS	Manycore Platform Software Stack	13, 14, 17
MRP	mandatory room property	5, 7, 8, 51–53

Notation	Description	Page List
OECD	Organisation for Economic Co-operation and Development	1
PAS	Patient Admission Scheduling	vi, 1–3, 5, 9, 25, 35, 43, 45
PASU	Dynamic Patient Admission Scheduling under Uncertainty	3
PRC	patient-room cost	9, 49
PRP	preferred room property	5, 7, 8, 51–53
RG	room gender	9, 49, 53
RHEL	Red Hat Enterprise Linux	13
RP	room preference	5, 7, 8, 51–53
RS	room specialism	7, 51, 53
SA	Simulated Annealing	vi, 21, 22, 25, 26, 31, 40, 43, 49, 50, 52
SDK	software development kit	14
SIMD	Single Instruction, Multiple Data	12, 19
SLES	SUSE Linux Enterprise Server	13
SP	Swap Patients	22, 23
SRG	static room gender	7, 51, 53
VPU	Vector Processing Unit	12

Appendix C
Notation

Notation	Description	Page List
A	The parameter of an exponential threshold curve $A \cdot e^{K \cdot T}$ which defines how big the relative difference between SA best solution S_{best} and the solution in pool must be to load the better solution from pool to SA and continue from that new solution.	26, 27, 29, 49
$C(p, r)$	Patient-room penalty matrix	7, 9
F_{PRC}	Patient-room cost (PRC) component of the objective function	9, 10
F_{RG}	Room gender (RG) component of the objective function	9, 10
F_{Tr}	Transfer component of the objective function	9, 10
F	The value of the objective functions of a solution	21, 23, 28, 29, 32, 40
G	The number of attempts in micro-benchmark 3.2	19
H	The size of the input and output array in micro-benchmark 3.2	19
K	The parameter of an exponential threshold curve $A \cdot e^{K \cdot T}$ which defines how big the relative difference between SA best solution S_{best} and the solution in pool must be to load the better solution from pool to SA and continue from that new solution.	26, 27, 29, 49
N_{max}	The maximum number of iterations on the same value of temperature of the SA algorithm	22, 23, 25, 28, 32, 36, 37, 39
N	The counter of iterations of the SA algorithm	23, 28, 32

Notation	Description	Page List
R	The part of the solution representing one room in Parallel II algorithm	31, 32, 51
S_{best}	The representation of the best solution found so far by SA	22, 23, 25, 26, 28, 29, 40, 41, 49, 52
S	The representation of the active solution of SA	21–23, 26–29, 31, 32, 40, 41, 51
T_{init}	The initial temperature of the SA algorithm	22, 23, 27, 28, 32, 36, 37, 39
T_{min}	The minimal temperature of the SA algorithm	22, 23, 25, 27, 28, 32, 36, 37, 39, 52
T_{pool}	The initial temperature of random solutions in the solution pool in Parallel I algorithm	25, 27, 37, 39
T_{reop}	The initial temperature threshold from which the β_{reop} is used	25, 27, 37, 39, 50
T	The working temperature of the SA algorithm	21, 23, 26–29, 32, 49, 52
ΔF	The difference of the objective functions in SA algorithm	21–24, 28, 32
$\Delta_{d_S}(z, s)$	Degree-specialism difference	8, 9
β_{reop}	The cooling rate of the SA algorithm for reoptimizations in Parallel I algorithm (the T_{reop} is a threshold temperature below which this value is used)	25, 27, 37, 39, 50
β	The cooling rate of the SA algorithm	21–23, 25, 27, 28, 32, 36, 37, 39
$\mathcal{D}_p(p)$	Set of all days the patient p is present in hospital	5, 10

Notation	Description	Page List
$\mathcal{D}$	Set of all days in planning horizon	5, 10, 51
$\mathcal{P}_F$	Set of female patients	5, 10
$\mathcal{P}_M$	Set of male patients	5, 10
$\mathcal{P}$	Set of all patients	5, 10, 52
$\mathcal{R}_D$	All D type rooms	9
$\mathcal{R}_Q$	Set of all room properties	6–8, 52
$\mathcal{R}$	Set of all rooms	6, 10, 52
$\mathcal{S}(z)$	Set of all specialisms of a department z	6, 8, 9
$\mathcal{S}_r(r)$	Set of specialisms of the room r	6, 9
$\mathcal{S}$	Set of all specialisms	52
$\mathrm{AD}(p)$	Admission day	5, 10
$\mathrm{DD}(p)$	Discharge day	5, 10
$\mathrm{LB}_{age}(r)$	Age lower bound of a room r	8
$\mathrm{UB}_{age}(r)$	Age upper bound of a room r	8
$changesCounter$	Array of the counters for all rooms R_i in the solution S, that increments when a change to the room R_i is made.	32
c	The saved value of the changes counter in Parallel II algorithm	32
$d_S(z)$	Degree of specialism of the department z	8, 9
d	Day $d \in \mathcal{D}$	5, 9, 10, 51–53
$f(r, d)$	Decision variable 2.12 that stores the number of female patients in room r on day d.	9, 10
$f_{RS}(p, r)$	Cost of patient p being in the room r for room specialism soft constraint	7, 9
$f_{\mathrm{DS}}(p, r)$	Cost of patient p being in the room r for department specialism soft constraint	7, 8
$f_{\mathrm{MRP}}(p, r)$	Cost of patient p being in the room r for mandatory room property soft constraint	7
$f_{\mathrm{PRP}}(p, r)$	Cost of patient p being in the room r for preferred room property soft constraint	7, 8
$f_{\mathrm{RP}}(p, r)$	Cost of patient p being in the room r for room preference soft constraint	7, 8
$f_{\mathrm{SRG}}(p, r)$	Cost of patient p being in the room r for static room gender soft constraint	7
$f_{age}(p, r)$	Cost of patient p being in the room r for age soft constraint	7, 8

Notation	Description	Page List
$itersPerCheck$	The interval between checks of solution pool (in Parallel Algorithm I) that are performed in order to keep the solution in SA not much worse than the the solution the pool.	25–28, 37, 39
$m(r, d)$	Decision variable 2.13 that stores the number of male patients in room r on day d.	9, 10
m	The neighborhood move in the SA algorithm	21–23, 28, 32
$p_S(p)$	Specialism of the patient p	5, 8, 9
$p_{\mathrm{MRP}}(p, r_q)$	MRP from point of view of a patient p	5, 7
$p_{\mathrm{PRP}}(p, r_q)$	PRP from point of view of a patient p	5, 8
$p_{\mathrm{RP}}(p)$	Room preference (RP) of the patient p	5, 8
$p_{age}(p)$	Age of the patient p	5, 8
$poolSize$	The size of the solution pool in Parallel Algorithm I	25–27, 37, 39
p	Patient $p \in \mathcal{P}$	5, 7–10, 49–53
$r_c(r)$	Capacity of the room r	6, 8, 10
r_q	Room property $r_q \in \mathcal{R}_Q$	5–8, 52
$r_z(r)$	Department the room r belongs to	6, 8
$r_{\mathrm{MRP}}(r, r_q)$	MRP	7, 8
$r_{\mathrm{PRP}}(r, r_q)$	PRP	8
r	Room $r \in \mathcal{R}$	6–10, 49, 51–53
s	Specialism $s \in \mathcal{S}$	8, 9, 50
$t(p, r, d)$	Decision variable 2.11 that is 1 iff patient p is transferred from room r on day d, and 0 otherwise.	9, 10
u_{max}	The threshold for temperature T_{pool} which defines how big the relative difference between SA best solution S_{best} and the solution in pool must be to load the better solution from pool to SA and continue from that new solution. Used to compute exponential threshold for all temperatures.	25–27, 37, 39
u_{min}	The threshold for temperature T_{min} which defines how big the relative difference between SA best solution S_{best} and the solution in pool must be to load the better solution from pool to SA and continue from that new solution. Used to compute exponential threshold for all temperatures.	25–27, 37, 39

Notation	Description	Page List
w_{DS}	Department specialism weight, default value is 1	8
w_{MRP}	Mandatory room property weight, default value is 5	7, 8
w_{PRP}	Preferred room property weight, default value is 2	8
w_{RG}	Room gender weight, default value is 1	8, 9
w_{RP}	Room preference weight, default value is 0.8	8
w_{RS}	Room specialism weight, default value is 1	8, 9
w_{SRG}	Static room gender weight, default value is 5	7, 8
w_{Tr}	Transfer weight, default value is 1	8, 9
w_{age}	Age weight, default value is 10	8
$x(p, r, d)$	Decision variable 2.10 that is 1 iff patient p is assigned to room r on day d, and 0 otherwise.	9, 10
z	Department	6, 8, 9, 50–52

Appendix D

Bibliography

[1] OECD. Oecd.stat - health expenditure and financing, March 2016. URL http://stats.oecd.org/index.aspx?DatasetCode=SHA.

[2] OECD. Health at a glance 2013. doi: http://dx.doi.org/10.1787/health_glance-2013-en. URL http://www.oecd-ilibrary.org/social-issues-migration-health/health-at-a-glance-2013_health_glance-2013-en.

[3] Doctor Zubair Clinic pvt. ltd. Image gallery, March 2016. URL http://drzubairahmad.com/.

[4] Peter Demeester, Wouter Souffriau, Patrick De Causmaecker, and Greet Vanden Berghe. A hybrid tabu search algorithm for automatically assigning patients to beds. *Artificial Intelligence in Medicine*, 48(1): 61–70, 2010.

[5] W Vancroonenburg, D Goossens, and FCR Spieksma. On the complexity of the patient assignment problem. Technical report, Tech. rep., KAHO Sint-Lieven, Gebroeders De Smetstraat 1, Gent, Belgium, 2011. URL http://allserv.kahosl.be/~wimvc/pas-complexity-techreport.pdf.

[6] Peter Demeester. Patient admission scheduling. URL http://allserv.kahosl.be/~peter/pas/. Not available at the moment. The offline copy of the website is on the enclosed CD.

[7] Burak Bilgin, Peter Demeester, Mustafa Misir, Wim Vancroonenburg, and Greet Vanden Berghe. One hyper-heuristic approach to two timetabling problems in health care. *Journal of Heuristics*, 18(3):401–434.

[8] Sara Ceschia and Andrea Schaerf. Local search and lower bounds for the patient admission scheduling problem. *Computers & Operations Research*, 38(10):1452–1463, 2011.

[9] IBM. ILOG CPLEX Optimization Studio v. 12.1, 2009. URL http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud.

[10] Saif Kifah and Salwani Abdullah. An adaptive non-linear great deluge algorithm for the patient-admission problem. *Information Sciences*, 295: 573–585, 2015.

[11] Troels Martin Range, Richard Martin Lusby, and Jesper Larsen. A column generation approach for solving the patient admission scheduling problem. *European Journal of Operational Research*, 235(1):252–264, 2014.

[12] Sara Ceschia and Andrea Schaerf. Modeling and solving the dynamic patient admission scheduling problem under uncertainty. *Artificial intelligence in medicine*, 56(3):199–205, 2012.

[13] Scheduling and Timetabling Research Group at the University of Udine. Dynamic patient admission scheduling problems. URL `http://satt.diegm.uniud.it/projects/pasu/`.

[14] Intel. Intel newsroom - Intel Xeon Phi coprocessor 5110P/3000 series, . URL `https://newsroom.intel.com/press-kits/intel-xeon-phi-coprocessor-5110p3000-series/`.

[15] N. Melab, R. Leroy, M. Mezmaz, and D. Tuyttens. Parallel Branch-and-Bound using private IVM-based work stealing on Xeon Phi MIC coprocessor. In *High Performance Computing Simulation (HPCS), 2015 International Conference on*, pages 394–399, July 2015. doi: 10.1109/HPCSim.2015.7237067.

[16] Sara Ceschia and Andrea Schaerf. Multi-neighborhood local search for the patient admission problem. In *Hybrid Metaheuristics*, pages 156–170. Springer, 2009.

[17] IBM. ILOG CPLEX Optimization Studio v. 12.6, 2013. URL `http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud`.

[18] Intel. Intel Xeon Phi product family: Product brief, . URL `http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html`.

[19] Intel. Intel Xeon Phi X100 family coprocessor - the architecture, . URL `https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner`.

[20] Intel. Intel Xeon Phi coprocessor architecture overview, August 2013. URL `https://software.intel.com/sites/default/files/Intel%C2%AE_Xeon_Phi%E2%84%A2_Coprocessor_Architecture_Overview.pdf`.

[21] Intel. Inside Intel Knights Landing architecture, January 2016. URL `http://www.hpctoday.com/viewpoints/inside-intel-knights-landing-architecture/`.

[22] James Reinders. An overview of programming for Intel Xeon processors and Intel Xeon Phi coprocessors. Technical report, Intel, 2012. URL https://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf.

[23] Intel. Intel Manycore Platform Software Stack (Intel MPSS) 3.5.2, . URL https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss.

[24] Anselm Busse. Xeon Phi overlay, January 2014. URL https://github.com/abusse/xeon-phi-overlay.

[25] Anselm Busse. Xeon Phi: Setting up a Gentoo host system, January 2014. URL http://anselm-busse.de/?p=32.

[26] Taylor Kidd Frances Roth, Sunny Gogar. *System Administration for the Intel Xeon Phi Coprocessor*. Intel, December 2014.

[27] Intel. *Configuring Intel Xeon Phi coprocessors inside a cluster*. IntelIntel, March 2013. URL https://software.intel.com/en-us/articles/configuring-intel-xeon-phi-coprocessors-inside-a-cluster.

[28] Intel. *MPSS Readme (includes installation instructions) for Linux*. Intel, . URL https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss.

[29] Intel. *Flash Issues & Remedies*. Intel, 2013. URL https://software.intel.com/sites/default/files/Flash%20FAQ.pdf.

[30] Intel. Intel Parallel Studio XE, 2016. URL https://software.intel.com/en-us/intel-parallel-studio-xe.

[31] Intel. Intel Parallel Studio XE 2016 update 3 - installation guide for Linux* OS. Technical report, Intel, 2016. URL https://software.intel.com/en-us/parallel-studio-xe-2016-install-guide-linux.

[32] Intel. *Offload Compiler Runtime for the Intel Xeon Phi Coprocessor*. Intel, 2013. URL https://software.intel.com/sites/default/files/article/366893/offload-runtime-for-the-intelr-xeon-phitm-coprocessor.pdf.

[33] Intel. *Native and Offload Programming Models*. Intel, 2014. URL https://software.intel.com/en-us/articles/native-and-offload-programming-models.

[34] Glibc wiki - libmvec, May 2016. URL https://sourceware.org/glibc/wiki/libmvec.

[35] Free Software Foundation. GCC, the GNU Compiler Collection, May 2016. URL https://gcc.gnu.org/.

[36] Intel. Intel C++ Compilers, May 2016. URL `https://software.intel.com/en-us/c-compilers`.

[37] CD Gelatt, MP Vecchi, et al. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[38] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[39] Intel. Intel Math Kernel Library (Intel MKL), 2016. URL `https://software.intel.com/en-us/intel-mkl`.