NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF NAVAL ARCHITECTURE AND MARINE ENGINEERING
DIVISION OF MARINE ENGINEERING

DIPLOMA THESIS

DEVELOPMENT OF HIGH PERFORMANCE PARALLEL ALGORITHMS FOR HYDRODYNAMIC LUBRICATION PROBLEMS

NICKOLAS FOSKOLOS

THESIS COMMITTEE
Supervisor:
Christos I. Papadopoulos - Assistant Professor NTUA
Members:
Lambros Kaiktsis - Associate Professor NTUA
Alexandros Gkinis - Assistant Professor NTUA

Development of High Performance Parallel Algorithms for Hydrodynamic Lubrication Problems

Nickolas Foskolos

July 29, 2015

Acknowledgments

I would like to express my very great appreciation to my supervisor, Assistant Professor Dr. Christos I. Papadopoulos for his inspiration and involvement in this project. Without his deep insight and encouragement, the current work would not be feasible at any rate.

I want to thank my colleagues Eleftherios Koukoulopoulos and Leonidas Raptis as their hard work set the foundation for the current project and the entire tribology research team for their crucial support in numerous occasions.

I must also thank all those who share their knowledge through courses and textbooks over the internet without profit and particularly Prof. Wen-mei Hwu and his team at the University of Illinois. My grateful thanks are also extended to the entire free software community for providing the tools utilized in this project.

Finally, I wish to thank my family for their support and encouragement throughout my studies.

Abstract

Contemporary research in tribology is often confronted with the simulation of complex phenomena and interactions. Numerical models of systems and processes become progressively more detailed and computationally costly, whereas design optimization procedures often require conduction of a large amount of simulations. Thus, the development and validation of efficient solution methods, which reduce computational time and exploit the available power of modern computer systems, is imperative. In the present work, parallel programming algorithms for the solution of the Reynolds equation for hydrodynamic lubrication problems using the finite difference method (FDM) are developed and evaluated. Two different parallel programming approaches are examined: (a) The Open Multi-Processing Application Programming Interface (OpenMP API), which utilizes shared-memory multicore computer systems, and (b) the CUDA framework by NVIDIA, which utilizes the cores of Graphics Processing Units (GPUs). Those programming approaches are implemented in an in-house algorithm, developed in C++, capable of solving steady-state and time-dependent hydrodynamic lubrication problems, taking into consideration cavitation phenomena and slip boundary conditions. More specifically, a portable CUDA Static Library solving the discretized partial differential equations with "Red-Black" Successive Over-Relaxation (SOR) is developed and validated for certain test cases, including converging-diverging (CD), diverging-converging (DC), textured and hydrophobic sliders. Additionally, the corresponding C++/OpenMP functions are also implemented, and shared-memory parallel computations for the same cases are performed, utilizing different number of CPU cores. The results of the present study demonstrate that parallel programming can greatly reduce computational time, especially in cases where large grid sizes are required, and thus runtime increases exponentially. In certain cases, utilization of GPU may result in a decrease in computational time by a factor of 30, in comparison to single-core calculations.

Περίληψη

Οι σύγχρονες ερευνητικές προσπάθειες στην περιοχή της τριβολογίας βρίσκονται συχνά αντιμέτωπες με τη μοντελοποίηση και προσομοίωση πολύπλοκων φαινομένων και αλληλεπιδράσεων. Τα αριθμητικά μοντέλα συστημάτων και διαδικασιών γίνονται όλο και πιό πολύπλοκα, με αυξούμενο υπολογιστικό κόστος, ενώ οι διαδικασίες βελτιστοποίησης απαιτούν τη διενέργεια ενός μεγάλου αριθμού προσομοιώσεων. Ως επακόλουθο, είναι αναγκαία η ανάπτυξη και επαλήθευση αποδοτικών μεθόδων επίλυσης, οι οποίες μπορούν να μειώσουν τον απαιτούμενο υπολογιστικό χρόνο, και να εκμεταλλευτουν τις δυνατότητες των σύγχρονων υπολογιστικών συστημάτων. Στην παρούσα εργασία αναπτύχθηκαν και αξιολογήθηκαν παράλληλοι αλγόριθμοι για την επίλυση της εξίσωσης Reynolds σε έδρανα υδροδυναμικής λίπανσης με τη μέθοδο των πεπερασμένων διαφορών. Συγκεκριμένα, εξετάστηκαν δύο μοντέλα παράλληλου προγραμματισμού: (α) Το Open Multi-Processing Application Programming Interface (OpenMP API) για υπολογιστικά συστήματα πολλών επεξεργαστών κοινόχρηστης μνήμης και (β) το Compute Unified Device Architecture (CUDA) framework της Nvidia που δίνει την δυνατότητα προγραμματισμού σε κάρτες γραφικών. Οι παραπάνω μέθοδοι εφαρμόστηκαν σε υφιστάμενο κώδικα, ο οποίος χρησιμοποιείται για την επίλυση προβλημάτων υδροδυναμικής λίπανσης με σταθερά ή χρονικά μεταβαλλόμενα φορτία, λαμβάνοντας υπόψιν φαινόμενα σπηλαίωσης και ιδιότητες υδροφοβικότητας. Πιό αναλυτικά, αναπτύχθηκε στατική βιβλιοθήκη κώδικα σε CUDA, για τη λύση των διακριτοποιημένων μερικών διαφορικών εξισώσεων με χρήση της μεθόδου "Red-Black", χρησιμοποιώντας την τεχνική Successive Over-Relaxation (SOR). Τα αποτελέσματα της αναπτυχθείσας μεθόδου, επαληθεύτηκαν για απλές γεωμετρίες συγκλινόντων-αποκλινόντων (CD) και αποκλινόντων-συγκλινόντων (DC) ολισθητών, καθώς και σε έδρανα με τεχνητή επιφανειακή τραχύτητα ή υδροφοβικότητα. Επιπλέον, αναπτύχθηκαν οι αντίστοιχες συναρτήσεις για παράλληλ επεξεργασία σε OpenMP και πραγματοποιήθηκαν οι ίδιοι υπολογισμοί σε συστήματα πολλών επεξεργαστών κοινόχρηστης μνήμης, χρησιμοποιώντας διαφορετικούς αριθμούς επεξεργαστών ανά περίπτωση. Τα αποτελέσματα δείχνουν ότι με χρήση μεθόδων παράλληλου προγραμματισμού μπορεί να μειωθεί σημαντικά ο χρόνος επίλυσης, ειδικά σε περιπτώσεις που έιναι απαραίτητη η χρήση μεγάλων υπολογιστικών πλεγμάτων, στις οποίες ο χρόνος επίλυσης αυξάνεται εκθετικά. Σε κάποιες περιπτώσεις, η χρήση κάρτας γραφικών μπορεί να μειώσει τον χρόνο επίλυσης έως και κατά 30 φορές σε σχέση με την επίλυση σειριακού κώδικα σε έναν πυρήνα υπολογιστή.

Contents

1 Introduction **13**
- 1.1 Tribology . . . 13
- 1.2 Literature review & thesis outline . . . 14

2 Main Bearing Types in Marine Engineering **18**
- 2.1 Introduction . . . 18
- 2.2 Thrust Bearings . . . 18
  - 2.2.1 Fundamentals of thrust bearings . . . 18
  - 2.2.2 Geometry of a thrust bearing . . . 20
- 2.3 Journal Bearings . . . 21
  - 2.3.1 Fundamentals of journal bearings . . . 21
  - 2.3.2 Geometry of journal bearings . . . 21

3 Mathematical Analysis **24**
- 3.1 Introduction . . . 24
- 3.2 Reynolds equation of hydrodynamic lubrication . . . 24
- 3.3 Reynolds equation for hydrophobic surfaces . . . 29
- 3.4 Cavitation in journal bearings . . . 30

4 Computational Methods **32**
- 4.1 Introduction . . . 32
- 4.2 The finite difference method . . . 32
- 4.3 Discretization of the basic equations . . . 35
  - 4.3.1 Discretization of the Reynolds equation for simple bearings 35
  - 4.3.2 Discretization of the Reynolds equation for bearings with hydrophobic surface treatment . . . 37
  - 4.3.3 Discretization of the Reynolds equation with the Elrod-Adams mass-conservation model for simple bearings . . . 38
- 4.4 The iterative solver . . . 40

5 High Performance Computing **42**
- 5.1 Introduction . . . 42
- 5.2 The age of parallel computing . . . 42
- 5.3 The serial code . . . 43

5.4 Parallel programming models 48
5.4.1 Open MultiProcessing (OpenMP) 49
5.4.2 Compute Unified Device Architecture (CUDA) 56
5.4.3 Message Passing Interface (MPI) 64
5.4.4 Hybrid models 66
5.5 Code Analysis 68

6 Numerical Simulations 72
6.1 Introduction 72
6.2 Validation of the developed code 72
6.3 Benchmark case: Steady-state equilibrium of a plain journal bearing. 74
6.4 Time-dependent response of a plain journal bearing 87
6.5 Steady-state response of a hydrophobic journal bearing 89
6.6 Steady-state response of a textured journal bearing 91
6.7 Steady-state equilibrium of a scratched journal bearing 94
6.8 Steady-state equilibrium of a plain journal bearing with oil-groove 97
6.9 Steady-state equilibrium of a textured journal bearing with oil-groove 99

7 Conclusions and Future Work 102

List of Figures

2.1 Development of thrust bearings: **(a)** parallel disks, **(b)** fixed inclined pad, **(c)** tilting pad. The shaded pattern is the stationary disk (Hori, 2006). 19
2.2 Example of a thrust bearing application used to sustain the thrust loads generated from the propeller of a ship (Stachowiak and Batchelor, 2001). 19
2.3 Profile sketch of a simple thrust bearing (Hori, 2006). 20
2.4 (a) Geometry of a plain journal bearing, and (b) geometry of the unwrapped lubricant film in a journal bearing. 22
2.5 Sommerfeld number Δ against eccentricity ratio ϵ (Stachowiak and Batchelor, 2001). 23

3.1 Hydrodynamic pressure generation between two moving surfaces (Stachowiak and Batchelor, 2001). 25
3.2 Equilibrium of a fluid film element; p is the pressure and τ_x the shear stress in the x direction (Stachowiak and Batchelor, 2001). 26
3.3 Profiles of velocity at the entrance of the fluid film (Stachowiak and Batchelor, 2001). 27
3.4 Continuity of flow in a column of the fluid film (Stachowiak and Batchelor, 2001). 28
3.5 Velocity profile near the fluid-wall boundary with (a) no-slip condition and (b) slip condition with slip length **b**. 29

4.1 Finite difference method: The computational grid. 33

5.1 Non-dimensional load capacity as a function of convergence ratio, for different values of length-to-width ratio of the bearing. Comparison between the results of the present code and those presented in (Stachowiak and Batchelor, 2001). 45
5.2 Influence of the value of relaxation factor ω on convergence speed. Performance of the Jacobi algorithm is also presented for comparison. 46
5.3 Algorithm of the serial SOR iterative solver. 48
5.4 A contemporary multicore Intel CPU. 50

5.5 Shared memory multicore CPU architecture. 51
5.6 The fork-join multithreading model. 52
5.7 OpenMP version algorithm of the default SOR iterative solver. . 54
5.8 A contemporary Nvidia GPU. 56
5.9 Typical CPU architecture. 57
5.10 Typical GPU architecture. 58
5.11 A three-dimensional grid of threads. 59
5.12 Architecture of a Von-Neumann processor. 59
5.13 (a) The CUDA multiprocessor model and (b) the CUDA device memories. 60
5.14 The Red-Black SOR domain decomposition. 61
5.15 The modified CUDA algorithm of the Red-Black SOR iterative solver. 63
5.16 A distributed memory cluster. 64
5.17 Vector addition algorithm with MPI. 66
5.18 A distributed memory cluster with multicore processors. 67
5.19 Vector addition algorithm with hybrid MPI + OpenMP. 68
5.20 The basic flow diagram of the developed code. 70
5.21 The architecture of the developed C++ project. 71

6.1 Validation of the developed algorithms for (a) a plain journal bearing, (b) a hydrophobic slider, (c) a diverging-converging slider, (d) a simple textured slider. 74
6.2 Equilibrium of a plain journal bearing under steady load: Execution times on an Intel i5-760 system for different number of utilized cores. 78
6.3 Equilibrium of a plain journal bearing under steady load: Speedup and parallel efficiency on an Intel i5-760 system for different number of utilized cores. 79
6.4 Equilibrium of a plain journal bearing under steady load: Execution times on an Intel Xeon E5-2620 system for different number of utilized cores. 80
6.5 Equilibrium of a plain journal bearing under steady load: Speedup and parallel efficiency on an Intel Xeon E5-2620 system for different number of utilized cores. 81
6.6 Equilibrium of a plain journal bearing under steady load: Execution times on an AMD Opteron 6276 system for different number of utilized cores. 82
6.7 Equilibrium of a plain journal bearing under steady load: Speedup and parallel efficiency on an AMD Opteron 7276 system for different number of utilized cores. 83
6.8 Topology of a ccNUMA Bulldozer server using hwloc. 85
6.9 Steady state equilibrium of a plain journal bearing: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of pressure on the interior surface of the bearing. 86

6.10 Time-dependent equilibrium of a plain journal bearing: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Plot of time-dependent vertical and horizontal load. 88
6.11 Geometry of the hydrophobic journal bearing. 90
6.12 Steady state equilibrium of a hydrophobic journal bearing: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of pressure on the interior surface of the bearing. 91
6.13 Geometry of the textured journal bearing. 93
6.14 Steady state equilibrium of a textured journal bearing: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of film thickness and pressure on the interior surface of the bearing. 94
6.15 Geometry of the scratched journal bearing. 96
6.16 Steady state equilibrium of a scratched journal bearing: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of film thickness and pressure on the interior surface of the bearing. 97
6.17 Steady state equilibrium of a plain journal bearing with oil groove: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of pressure and lubricant density on the interior surface of the bearing. 99
6.18 Steady state equilibrium of a textured journal bearing with oil groove: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of pressure and lubricant density on the interior surface of the bearing. 101

List of Tables

5.1 Simple converging thrust bearing model. Non-dimensional load capacity for different values of convergence ratio k and length to width ratio of the bearing. 44
5.2 Influence of the value of relaxation factor ω on convergence speed. 45
5.3 Profiling details of the Reynolds equation solver for simple converging bearings. 47

6.1 Description of the hardware tested in the present work. 73
6.2 Steady-state equilibrium of a plain journal bearing: Geometric, operational and performance parameters. 75
6.3 Equilibrium of a plain journal bearing under steady load: OpenMP simulations utilizing (a) an Intel i5-760, and (b) an Intel Xeon E5-2620 system. 76
6.4 Equilibrium of a plain journal bearing under steady load: OpenMP simulations utilizing an AMD Opteron 6276. 77
6.5 Time-dependent response of a plain journal bearing: Geometric and operational parameters. 87
6.6 Steady-state equilibrium of a hydrophobic journal bearing: Geometric, operational and performance parameters. 89
6.7 Steady-state equilibrium of a textured journal bearing: Geometric, operational and performance parameters. 92
6.8 Steady-state equilibrium of a scratched journal bearing: Geometric, operational and performance parameters. 95
6.9 Steady-state equilibrium of a plain journal bearing with oil groove: Geometric, operational and performance parameters. 98
6.10 Steady-state equilibrium of a textured journal bearing with oil groove: Performance parameters. 100

Nomenclature

α	slip coefficient $[\frac{m}{Pas*sec}]$: $\alpha = b\eta$
β	lubricant bulk modulus
ϵ	dimensionless eccentricity: $\epsilon = \frac{e}{c}$
η	fluid dynamic viscosity $[Pa \cdot s]$
μ	friction coefficient: $\mu = \frac{F}{W}$
ω	angular velocity $[s^{-1}]$
ϕ	attitude angle $[degrees]$
ρ_c	lubricant density at cavitation pressure p_c $[\frac{kg}{m^3}]$
ρ	lubricant density $[\frac{kg}{m^3}]$
τ_c	critical shear stress $[Pa]$
τ_x	shear stress in x-direction $[Pa]$
τ_y	shear stress in y-direction $[Pa]$
θ_c	contact angle $[degrees]$
θ	hydrodynamic film angle $[degrees]$
$\theta'_{h,end}$	end angle of hydrophobic region in y-direction [degrees]
$\theta'_{h,start}$	start angle of hydrophobic region in y-direction [degrees]
$\theta'_{s,end}$	end angle of scratched region in y-direction [degrees]
$\theta'_{s,start}$	start angle of scratched region in y-direction [degrees]
$\theta'_{t,end}$	end angle of texture region in y-direction [degrees]
$\theta'_{t,start}$	start angle of texture region in y-direction [degrees]
B	bearing width $[m]$: $B = \pi D$

b	slip length $[m]$
B_s	slip width $[m]$
c	bearing clearance $[m]$: $c = R_1 - R_2$
c_m	center of mass$[m]$
D	bearing diameter $[m]$: $D = 2R$
$d_{t,\theta}$	density of dimples in y-direction
$d_{t,L}$	density of dimples in x-direction
e	eccentricity $[m]$
F	friction force $[N]$
$F*$	dimensionless friction force: $F* = \frac{Fh_{min}}{U\eta BL}$
F_x	external bearing load $[N]$: x-axis component
F_z	external bearing load $[N]$: z-axis component
g	switch function of the Elrod-Adams model
h	lubricant film thickness $[m]$
h_d	depth of dimples [mm]
h_{max}	maximum lubricant film thickness $[m]$
h_{min}	minimum lubricant film thickness $[m]$
h_s	depth of scratches [mm]
k	convergence ratio: $k = \frac{h_{max}}{h_{min}} - 1$
L	bearing length $[m]$
$L_{h,end}$	end angle of hydrophobic region x- direction [m]
$L_{h,start}$	start angle of hydrophobic region x- direction [m]
$L_{s,end}$	end angle of scratched region x- direction [m]
$L_{s,start}$	start angle of scratched region x- direction [m]
$L_{t,end}$	end angle of texture region x- direction [m]
$L_{t,start}$	start angle of texture region x- direction [m]
N	shaft rotational speed $[rps]$
$N_{d,\theta}$	number of dimples in y-direction

$N_{d,L}$	number of dimples in x-direction
$N_{s,L}$	number of scratches in x-direction
O_B	bush center $[m]$
O_S	shaft center $[m]$
P	total exernal force $[N]$
p	pressure $[Pa]$
p_0	boundary pressure $[Pa]$
p_c	cavitation pressure $[Pa]$
p_{max}	maximum pressure $[Pa]$
R	bearing radius $[m]$
r	crankshaft radius $[m]$
R_1	bush radius $[m]$
R_2	shaft radius $[m]$
S	Sommerfield number: $S = \frac{W}{LU\eta}(\frac{c}{R})^2$
t	time $[s]$
U	rotor linear velocity $[\frac{m}{s}]$
u	x-axis fluid velocity $[\frac{m}{s}]$
U_1	x-axis stator speed $[\frac{m}{s}]$
U_2	x-axis rotor speed $[\frac{m}{s}]$
v	y-axis fluid velocity $[\frac{m}{s}]$
V_1	y-axis stator speed $[\frac{m}{s}]$
V_2	y-axis rotor speed $[\frac{m}{s}]$
W	total hydrodynamic force $[N]$: $W = \sqrt{W_x + W_z}$
w	z-axis fluid velocity $[\frac{m}{s}]$
$W*$	dimensionless hydrodynamic force: $W* = \frac{Wh_{min}}{U\eta BL}$
W_x	x-axis hydrodynamic force component $[N]$
W_z	z-axis hydrodynamic force component $[N]$
x	x-axis coordinate
y	y-axis coordinate
z	z-axis coordinate

Chapter 1

Introduction

1.1 Tribology

Tribology is a relatively new branch of science that studies friction, lubrication and wear of interacting surfaces in relative motion.

- *Friction* is the force that resists motion when the surfaces of two solid objects slide over each other.
- *Lubrication* is the process of reducing the wear of one or both surfaces in close proximity, and moving relative to each other, by interposing a substance called lubricant between the surfaces, to carry or to help carry the load (pressure generated) between the opposing surfaces. The lubricant can be a solid, a solid/liquid dispersion, a liquid, a liquid/liquid dispersion or a gas.
- *Wear* is the erosion or sideways displacement of material from a solid surface performed by the action of another surface.

Since the beginning of recorded history, humans tried to reduce friction between surfaces in order to satisfy certain needs in their everyday lives. For that purpose, low friction surfaces and different types of bearing applications had been developed, whereas the positive impact of lubrication had been widely understood. However, tribology as a scientific domain was first cultivated by Osborne Reynolds, who proved that hydrodynamic pressure of a liquid entrained between two surfaces sliding relative to each other was sufficient to prevent contact, and thus wear, between the surfaces even at low sliding speeds. The work of Reynolds inspired several scientists and engineers to study the lubrication phenomena in order to reduce friction in mechanical parts of contemporary engines.

Tribology is a new field of science and engineering and subject to controversy among the scientific circles. A significant part of the present knowledge was established during the second world war. Taking into consideration the importance of complex machinery in our modern civilizations, one can understand

the significance of tribology. Especially during the latest environmental crisis, the struggle for energy efficiency is getting more and more attention and one way to reduce energy consumption is by reducing friction between the countless sliding surfaces which are present in modern mechanical systems. Besides that, proper lubrication can limit wear and seizure, extending the life of components and thus conserving natural resources.

1.2 Literature review & thesis outline

Tribology plays an important role in our efforts for energy efficiency and environmental protection. In industrialized countries almost 30% of the total energy production is wasted due to friction. In the United States, energy losses caused by friction add up to a total cost of $100 billion US dollars per year. It is estimated that research in tribology can save up to $21 billion US dollars (Czichos, 1978).

In marine Diesel engines, a significant amount of energy - around 5% of the total generated power - is lost due to mechanical friction (Volund, 2003) . Taking into consideration that the power of a large two-stroke engine can be higher than 80000 kW, frictional losses correspond to a very large amount of energy. In addition, regulations for greenhouse gas emissions enforced by the International Maritime Organization (IMO) become stricter, making improvements in energy efficiency more important than ever before. The distribution of frictional losses in large two-stroke marine Diesel engines is presented below (Clausen, 2014):

- Guide Shoe Bearings (31%)
- Piston Ring Packs (26%)
- Crankshaft Bearings (23%)
- Connecting Rods (10%)
- Thrust Bearing (5%)

In a different study (Comfort, 2003), the distribution of energy losses in small heavy-duty Diesel engines has been estimated as:

- Piston Ring Assembly (45% to 50%)
- Engine Bearings (20% to 30%)
- Engine Auxiliaries (20% to 25%)
- Valve Train (7% to 15%)

By studying these figures, it becomes evident that tribology can play an important role in the effort to reduce friction losses in Diesel engines. It should be noted that CO_2 emissions caused by shipping alone correspond to 4% of the total CO_2 emissions, although shipping, in terms of transporting one cargo tonne

for one mile, exhibits the lowest CO_2 emission, in comparison to any other type of transportation (Clausen, 2014).

Friction and wear can be significantly reduced when a thin liquid film is maintained between two interacting surfaces, by minimizing friction forces and preventing direct surface to surface contact. The majority of tribological contacts, such as fluid bearings, piston rings, seals etc., mainly operate within the hydrodynamic lubrication regime (Stachowiak and Batchelor, 2001), (Hori, 2006). The tribological behavior of hydrodynamically lubricated contacts can be efficiently predicted by the solution of Reynolds equation, requiring appropriate application of conditions at the boundaries of the fluid domain. Special treatment is necessary for cases where pressure falls below vapor pressure in certain areas of the lubricant domain, giving rise to cavitation. The simplest approach of cavitation modeling is the utilization of the Reynolds boundary condition, zeroing all negative pressure values in the solution domain, and imposing zero pressure derivative at the transition boundary between the full-film and the cavitation regions (Stachowiak and Batchelor, 2001). This approach predicts accurately lubricant pressure, however it violates mass conservation in the cavitation region and fails to accurately calculate the location of the film rupture boundary (Giacopini et al., 2010). The former deficiency can be tackled by the use of cavitation models; for the purposes of the present work, the Elrod-Adams cavitation algorithm has been selected (Elrod and Adams, 1974), (Elrod, 1981). According to this algorithm, the lubricant domain is divided into two regions; the full-film (pressurized) and the cavitation region. In the full-film region, the Reynolds equation is valid, whereas in the cavitation region, only a fraction r of the film thickness is occupied with lubricant, while the remaining volume is occupied by lubricant vapors and air. Concerning the numerical solution, Elrod and Adams proposed the employment of a "global" differential equation, which is valid in both regions of the lubricant film, representing different quantities in each region. This requires the use of a cavitation factor g, taking the values of 1 and 0 within the pressure and cavitation regions, respectively. Vijayaraghavan and Keith (Vijayaraghavan and Keith, 1989) have presented a numerically more stable implementation of the Elrod-Adams algorithm, using a half-step finite difference scheme and deriving the shear and pressure flow terms in a more consistent way. Fesanghary and Khonsari (Fesanghary and Khonsari, 2011) have proposed a modification of the original Elrod-Adams switch function, which regularizes the transition between the full film and the cavitating region, and accelerates the convergence of the solution procedure.

In the last decade, the scientific community has been heavily concerned with the improvement of the operational behavior of hydrodynamic bearings. To this end, surface treatment technologies, such as artificial surface texturing and hydrophobicity, have emerged as promising candidates, if appropriately implemented. Artificial surface texturing is associated with the introduction of small periodic geometrical irregularities in the form of dimples, grooves or pockets at part of the interacting surfaces. Recent numerical and experimental studies have demonstrated that surface texturing may substantially improve the performance of sliding contacts, in terms of load capacity and friction coeffi-

cient (Etsion et al., 2004), (Papadopoulos et al., 2011). On the other hand, wetting-resistant hydrophobic or super-hydrophobic surfaces, display a complex nanostructure, characterized by a combination of micro- and nano- scale roughness. The contact angle of a liquid drop resting on a super-hydrophobic surface may attain values higher than 150 degrees. As a result, flow over a hydrophobic surface exhibits very low levels of friction in the fluid-solid interface. The difference between a plain and a hydrophobic surface is that for the latter, the no-slip boundary condition at the fluid-solid interface (where the velocity of the fluid molecules in contact with the boundary should be equal to boundary velocity) is no longer valid. In particular, the molecules of the fluid adjacent to the hydrophobic wall exhibit a different velocity than the velocity of the wall, when the local value of shear stress exceeds a critical value. Several studies, such as (Fatu et al., 2011),(GuoJun et al., 2007) have demonstrated that proper introduction of hydrophobicity at part of the stator of a bearing may lead to substantial improvement in load capacity and friction coefficient.

The performance of lubricated contacts can also be greatly affected by quality degradation of the bearing surface. In journal bearings, lubricant contamination, caused either by solid particles, severe impact loading or frequent starts/stops, may lead to deterioration of the bearing surface, usually in the form of scratches following the running pattern of the engine. In order to support the external load, scratched bearings will operate at higher eccentricity values, and thus at smaller values of film thickness and, generally, higher values of temperature (Dobrica and Fillon, 2012), (Helene et al., 2013). The critical question is whether a scratched bearing is capable of operating in a safe manner, or its maintenance/replacement is obligatory. Detailed numerical simulations of the response of scratched bearings would be invaluable in supporting this decision making. However, scratch dimensions usually vary in the micrometer scale, and, as a result, numerical simulations require very fine computational grids, which can result in prohibitively large amounts of computational time.

The problems mentioned so far present various numerical difficulties. Due to the large grid sizes demanded by the problem complexity and the geometry irregularities, the execution duration of the computational algorithms could be unbearable. In this respect, the use of High Performance Computing (HPC) provides viable and efficient means for tackling computationally intensive problems. A typical method for solving the Reynolds equation for hydrodynamic lubrication is the Finite Difference Method (FDM) (Stachowiak and Batchelor, 2001),(Hori, 2006),(Khonsari and Booser, 2008). This method typically employs a Gauss Seidel iterative procedure, which can converge faster if successive overrelaxation (SOR) is introduced. Numerical solution of the FDM method may be substantially accelerated by utilizing the Open Multi-Processing Application Programming Interface (OpenMP API), which is associated with shared memory multicore computer systems, the Message Passing Interface (MPI), for distributed memory parallel computer networks, and the CUDA framework by NVIDIA, which utilizes the cores of Graphics Processing Units (GPUs). In (Konstantinidis and Cotronis, 2012), (Liu et al., 2011), algorithms for accelerating the SOR method using GPUs with CUDA were developed for Compu-

tational Fluid Dynamics (CFD) problems; the obtained results demonstrate a potential for impressive reduction of computational time, compared to single-core simulations. Other research groups have utilized the CUDA platform along with OpenMP to solve the discretized incompressible and compressible Reynolds equations (Wang et al., 2012), (Wang et al., 2009), showing that, for large grid sizes, the performance of the less expensive GPU computing is on a par with the performance of shared memory multi-threaded computing, with the latter being substantially more expensive.

In the present work, parallel programming algorithms utilizing the OpenMP and CUDA interfaces have been developed for the FDM solution of the Reynolds equation in two-dimensional steady and time-dependent hydrodynamic lubrication problems. Two different forms of the Reynolds equation, with and without cavitation modeling, have been considered. First, the developed algorithms have been validated against published literature results, for several test-cases. Next, the algorithms have been applied for solving the steady-state and time-dependent response of plain, textured, hydrophobic and scratched journal bearings, for different grid sizes and CPU/GPU types. The outline of the present thesis is as follows. In chapter 2 the main types of bearings, used in marine Diesel engines are presented. In chapter 3, the Reynolds equation for solving hydrodynamic lubrication problems is presented, along with the appropriate modifications for taking into account hydrophobic surfaces or cavitation phenomena. In chapter 4, the computational methods for solving the Reynolds equation are examined, whereas in chapter 5, high performance parallel computing methods are introduced. In chapter 6 the results of the numerical simulations are presented. Finally, in chapter 7 the conclusions of the current thesis along with proposals for future work are discussed.

Chapter 2

Main Bearing Types in Marine Engineering

2.1 Introduction

In this chapter, the two basic types of hydrodynamic bearings, namely thrust and journal bearings, will be presented. The basic characteristics of these components, their geometry and the basic parameters that affect their operational performance will be examined.

2.2 Thrust Bearings

2.2.1 Fundamentals of thrust bearings

An axial load acting on a shaft is called a thrust. A bearing that supports a thrust is called a thrust bearing. A thrust bearing consists of a rotating disk (the rotor) that is a part of the shaft, and a stationary disk (the stator - shaded part) as shown in Fig. 2.1-a. In the early days, the disks were parallel to each other so load capacity was very low, as no pressure was generated from the motion of the fluid between the disks. As a result, significant friction, wear and thus energy loss occurred. The next logical step was to create an oil wedge between the surfaces of the rotor and stator, in order to generate fluid pressure rise, and keep a minimum distance between the moving surfaces. To achieve this, an inclined pad was fixed to the stationary disk (the shaded wedge-shaped pad), as shown in Fig. 2.1-b. This solution was clearly better than the first but not so robust. That is because the optimal angle of inclination is very small and even if it is accurately calculated, it will probably change throughout the lifetime of the component due to the action of different loads or due to thermal and elastic deformations of the initial geometry. A solution to this problem was found independently by A. G. M. Michell (1870 – 1959) in Australia and A. Kingsbury (1863 – 1943) in the USA. In their design, the inclined pad was

supported by a pivot and the angle of inclination became self-adjusted depending on the bearing operational conditions (load, speed). The only thing that needs to be defined is the appropriate pivot position. This is shown schematically in Fig. 2.1-c.

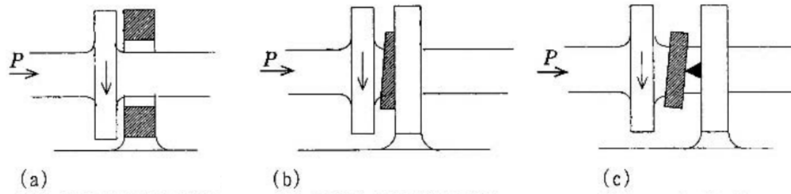


Figure 2.1: Development of thrust bearings: **(a)** parallel disks, **(b)** fixed inclined pad, **(c)** tilting pad. The shaded pattern is the stationary disk (Hori, 2006).

Michell's and Kingsbury's inventions were also applied to the shaft system of ships (Fig. 2.2). These small size (one-tenth the size of old bearing designs), low friction and long life designs made possible the development of more powerful engines and propellers. They were placed in ships during the first World War and became a standard for ships and power plants all around the world.

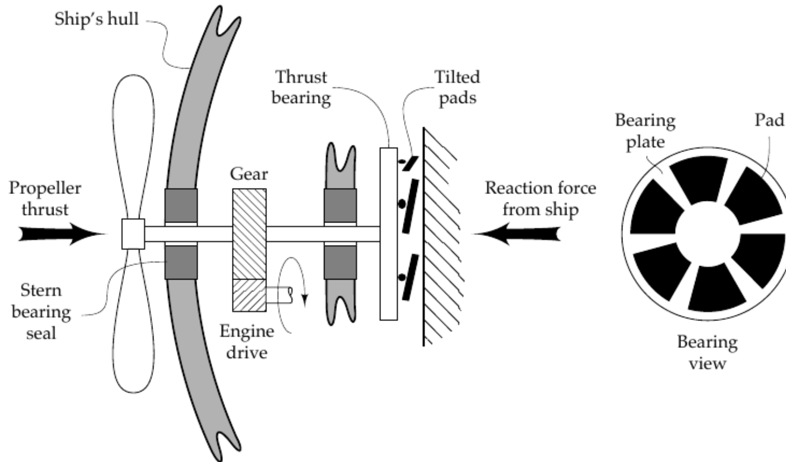


Figure 2.2: Example of a thrust bearing application used to sustain the thrust loads generated from the propeller of a ship (Stachowiak and Batchelor, 2001).

2.2.2 Geometry of a thrust bearing

A thrust bearing consists of an inclined pad (stator), that remains stationary during bearing operation conditions and a moving part (rotor), which rotates with the shaft. The basic parameters that define the bearing geometry are (Fig 2.3):

- The length of the bearing L
- The breadth of the bearing B (perpendicular to the sketch of Fig. 2.3)
- The value of maximum film thickness h_1
- The value of minimum film thickness h_2

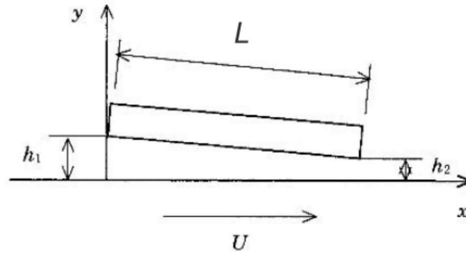


Figure 2.3: Profile sketch of a simple thrust bearing (Hori, 2006).

Film thickness is distributed in a linear way, and can be calculated from the length of the bearing and its inlet and outlet values

$$h(x) = h_1 - \frac{x}{L}(h_1 - h_2) \tag{2.1}$$

There are many parameters that affect the performance of the bearing during its operational conditions. These include:

- The viscosity of the lubricant η
- The temperature of the lubricant T, which in turn alters its viscosity
- The rotational speed of the rotor U

Hydrodynamic lubrication ensures that the rotor and stator surfaces do not come into contact. During the startup or shutdown of the engine, when the rotational speed is not sufficient to maintain the required film thickness, metal to metal contact occurs, which may result in serious damage of the surface of the bearing. To avoid contact, hydrostatic lift is occasionally enforced to keep the two surfaces apart in such conditions. It is also worth mentioning that temperature plays an important role in the performance of the bearing as heat

generation during the operation of the bearing decreases lubricant viscosity, which in turn affects the thickness of the film and the load that the bearing can support.

2.3 Journal Bearings

2.3.1 Fundamentals of journal bearings

Journal bearings are engineering components used to support the radial loads of rotating shafts. Geometrically, a journal bearing is a hollow cylinder, which encloses a solid shaft that rotates about its axis. The radius of the bearing is slightly larger than that of the shaft; the difference between the bearing and the shaft radius is called (radial) clearance, c. The bearing cylinder is usually held stationary. The hydrodynamic film, which supports the radial loads, is generated between the surfaces of the rotating shaft and the stationary bearing.

2.3.2 Geometry of journal bearings

Here, the geometry of the typical journal bearing of Fig. 2.4 is examined. Depending on the magnitude and direction of the external shaft load, the shaft will adjust its position within the inner volume of the bearing, until force equilibrium is attained. Shaft position can be defined by the values of eccentricity e (distance between the bearing and shaft centers) and attitude angle ϕ (angle between the load line and the line of centers), see Fig. 2.4(a). Equilibrium is attained when the sum of the hydrodynamic forces in the lubricant domain and the sum of the externally applied loads on the shaft are equal. In the present work, a two-dimensional Newton-Raphson root-finding method is applied to calculate the journal equilibrium position. In each iteration step, new values for eccentricity, e, and attitude angle, ϕ, are determined.

The basic characteristics of the bearing are:

- The clearance of the bearing c, defined as the difference between the bearing and shaft radii.
- The value of film thickness at any any angle θ, defined as:

$$h = c(1 + \epsilon cos\theta) \tag{2.2}$$

  where ϵ the eccentricity ratio (e/c)

- The "Sommerfeld Number"

$$\Delta = \frac{W}{LU\eta}(\frac{c}{R})^2 \tag{2.3}$$

  which is a critical non-dimensional parameter in journal bearing design. Here, W is the total bearing load, L the bearing length and $U = \frac{\omega}{R}$ the tangential velocity of the shaft. In Fig. 2.5, Sommerfeld number is

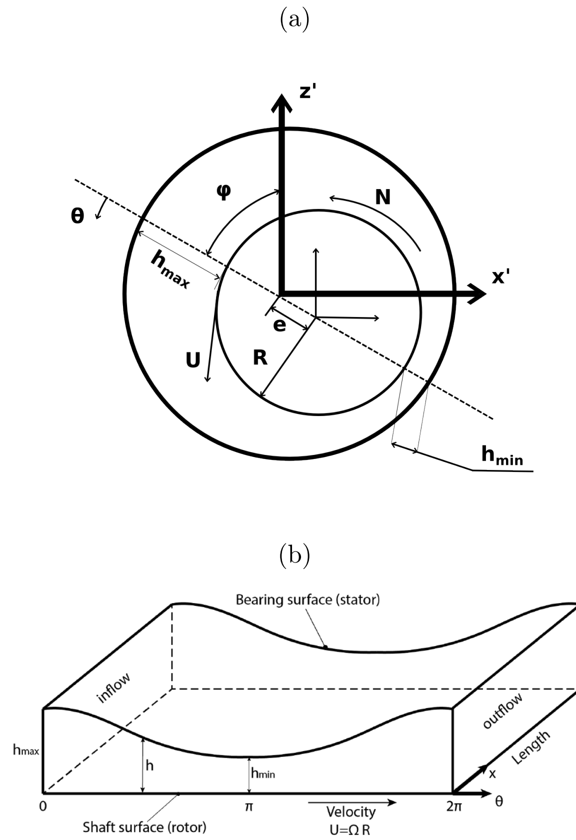


Figure 2.4: (a) Geometry of a plain journal bearing, and (b) geometry of the unwrapped lubricant film in a journal bearing.

plotted against bearing eccentricity ratio ϵ for different values of length to diameter, $\frac{L}{D}$, ratios of the bearing.

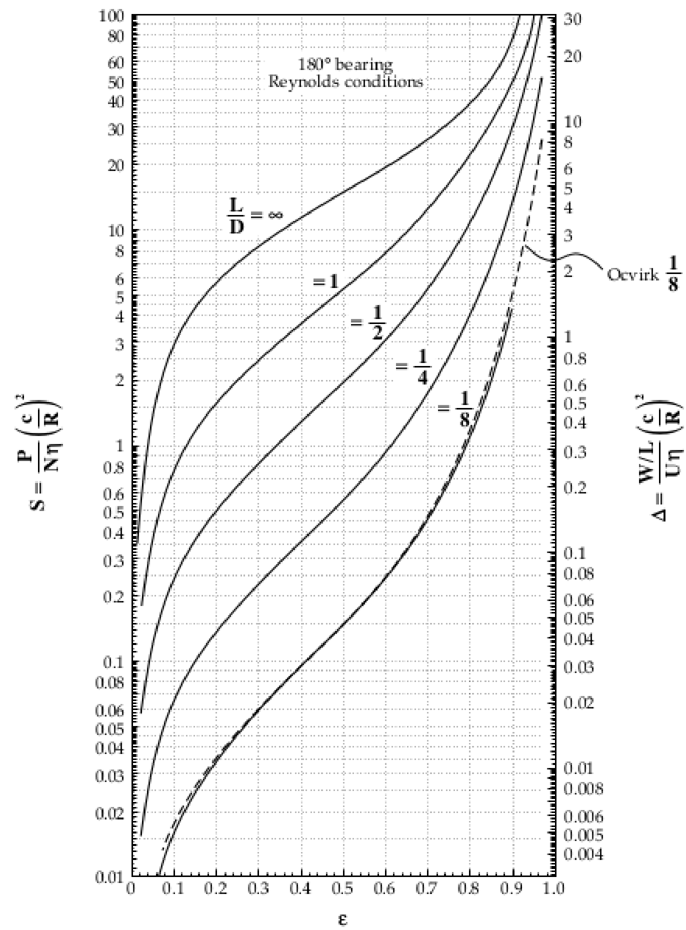


Figure 2.5: Sommerfeld number Δ against eccentricity ratio ϵ (Stachowiak and Batchelor, 2001).

Chapter 3

Mathematical Analysis

3.1 Introduction

In this chapter, the equations that describe the phenomenon of hydrodynamic lubrication and their derivation from fluid dynamics theory are discussed. Three different types of equations are presented, which can be used to analyze (a) conventional bearings, (b) bearings with hydrophobic surfaces and (c) bearings with diverging regions where cavitation phenomena play an important role.

3.2 Reynolds equation of hydrodynamic lubrication

Based on hydrodynamic lubrication theory, when a viscous fluid (the lubricant) is inserted between two inclined surfaces that move relative to each other, a fluid film is generated, that prevents the contact of the two surfaces by developing a pressure field, capable of supporting the acting load (Fig 3.1). Hydrodynamic lubrication theory can be expressed mathematically by the Reynolds equation. This equation is basically a simplification of the Navier-Stokes momentum and continuity equations. The process followed in this section is similar to that in (Stachowiak and Batchelor, 2001); the reader is encouraged to seek further details in similar textbooks.

The basic simplifying assumptions for derivation of the Reynolds equation are (Stachowiak and Batchelor, 2001):

- Body forces are neglected, as there are not any external fields of forces acting on the fluids.
- Pressure is constant in the film thickness direction.
- There is no slip at the solid boundaries (velocity of fluid adjacent to the boundary is that of the boundary). This assumption is not valid in the case of hydrophobic treatment on the surface of the bearing.

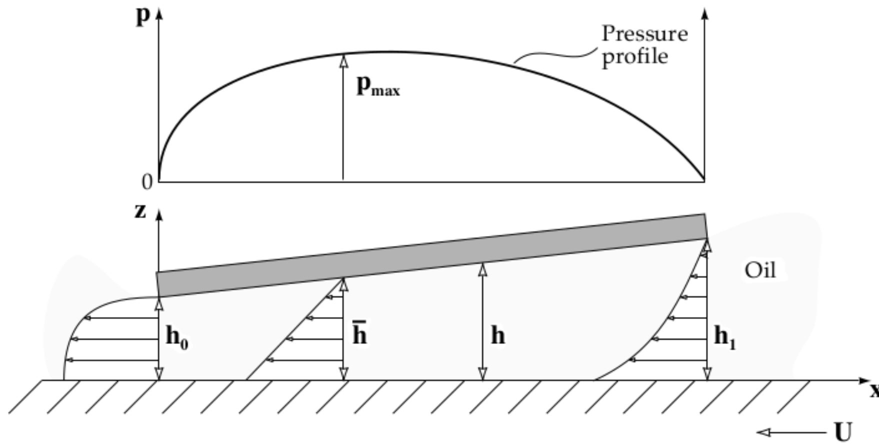


Figure 3.1: Hydrodynamic pressure generation between two moving surfaces (Stachowiak and Batchelor, 2001).

- The lubricant behaves as a Newtonian fluid.
- The flow is laminar.
- Fluid inertia is neglected.
- Fluid density is constant. This is generally true for liquid lubricants.
- Viscosity is constant in the film thickness direction. In the present analysis an isothermal fluid is assumed, therefore viscosity is constant throughout the fluid domain.

First, the equilibrium of a fluid element is considered (Fig 3.2).

The equilibrium of forces on both sides gives

$$\frac{\vartheta \tau_x}{\vartheta z} dxdydz = \frac{\vartheta p}{\vartheta x} dxdydz \tag{3.1}$$

and for a non zero volume ($dxdydz \neq 0$) the result is

$$\frac{\vartheta \tau_x}{\vartheta z} = \frac{\vartheta p}{\vartheta x} \tag{3.2}$$

Following the same method in the y direction we have that

$$\frac{\vartheta \tau_y}{\vartheta z} = \frac{\vartheta p}{\vartheta y} \tag{3.3}$$

In the z direction we have assumed that pressure is constant throughout the film

$$\frac{\vartheta p}{\vartheta z} = 0 \tag{3.4}$$

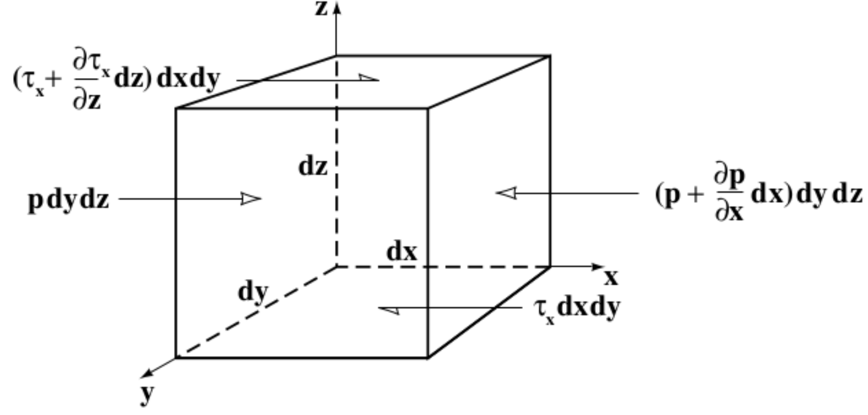


Figure 3.2: Equilibrium of a fluid film element; p is the pressure and τ_x the shear stress in the x direction (Stachowiak and Batchelor, 2001).

Shear stress can be expressed using dynamic viscosity η as:

$$\tau_x = \eta \frac{\vartheta u}{\vartheta z} \tag{3.5}$$

$$\tau_y = \eta \frac{\vartheta v}{\vartheta z} \tag{3.6}$$

Substituting shear stress in equations (3.2) and (3.3) we get the final equilibrium conditions for the fores acting on the fluid element of Fig 3.2:

$$\frac{\vartheta p}{\vartheta x} = \frac{\vartheta}{\vartheta z}(\eta \frac{\vartheta u}{\vartheta z}) \tag{3.7}$$

$$\frac{\vartheta p}{\vartheta y} = \frac{\vartheta}{\vartheta z}(\eta \frac{\vartheta v}{\vartheta z}) \tag{3.8}$$

Integrating equations (3.7) and (3.8) and considering non-slip conditions ($u = U_2$ at $z = 0$ and $u = U_1$ at $z = h$) , where U_1 and U_2 are the velocities of the moving surfaces, we get the final expressions for velocities towards the x and y directions (Fig 3.3)

$$u = (\frac{z^2 - zh}{2\eta})\frac{\vartheta p}{\vartheta x} + (U_1 - U_2)\frac{z}{h} + U_2 \tag{3.9}$$

$$v = (\frac{z^2 - zh}{2\eta})\frac{\vartheta p}{\vartheta y} + (V_1 - V_2)\frac{z}{h} + V_2 \tag{3.10}$$

The second condition that must be met, besides the equilibrium of forces, is the continuity of flow. For a column of flow in the hydrodynamic film and under

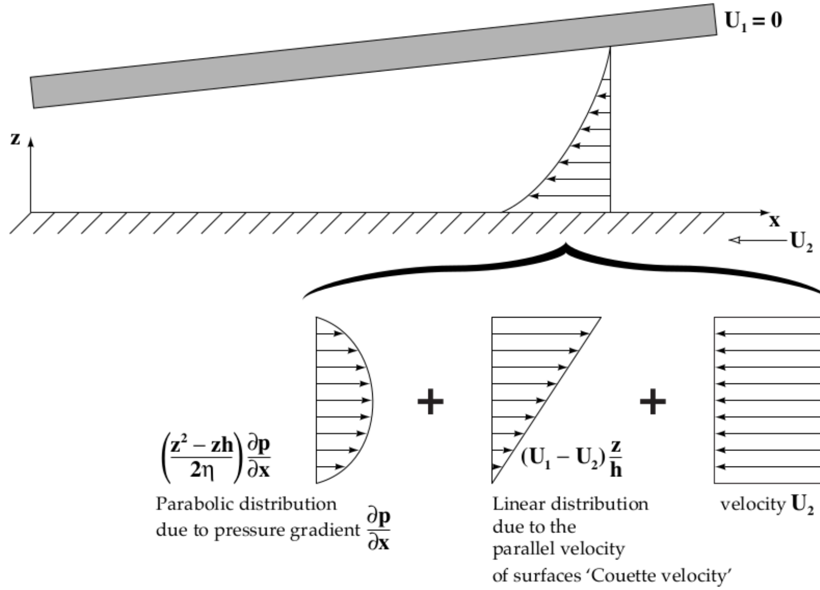


Figure 3.3: Profiles of velocity at the entrance of the fluid film (Stachowiak and Batchelor, 2001).

steady conditions, the mass of liquid entering the volume must be equal to the amount exiting on the other side (Fig 3.4) or for a non zero volume $dxdy \neq 0$

$$\frac{\vartheta q_x}{\vartheta x} + \frac{\vartheta q_y}{\vartheta y} + (w_h - w_0) = 0 \tag{3.11}$$

where w_h is the velocity at which the top of the column moves down, and w_0 the one at which the bottom of the column moves up. The flow rates (per unit length) are simply the integrals of the velocities through the film thickness:

$$q_x = \int_0^h u dz \tag{3.12}$$

$$q_y = \int_0^h v dz \tag{3.13}$$

Substituting the calculated expressions for the velocity profiles (3.9-3.10), the flow rates finally are

$$q_x = -\frac{h^3}{12\eta}\frac{\vartheta p}{\vartheta x} + (U_1 + U_2)\frac{h}{2} \tag{3.14}$$

$$q_x = -\frac{h^3}{12\eta}\frac{\vartheta p}{\vartheta y} + (V_1 + V_2)\frac{h}{2} \tag{3.15}$$

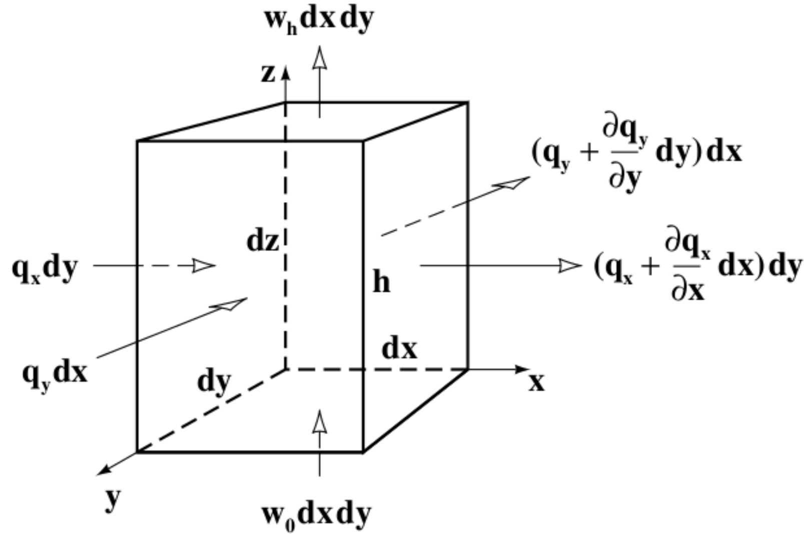


Figure 3.4: Continuity of flow in a column of the fluid film (Stachowiak and Batchelor, 2001).

If we insert the flow rates in equation (3.11) and assume that $U = U_1 + U_2$ and $V = V_1 + V_2$, that is, the velocities of the surface do not change locally throughout the entire film we get the full Reynolds equation in three dimensions

$$\frac{\vartheta}{\vartheta x}(\frac{h^3}{\eta}\frac{\vartheta p}{\vartheta x}) + \frac{\vartheta}{\vartheta y}(\frac{h^3}{\eta}\frac{\vartheta p}{\vartheta y}) = 6(U\frac{dh}{dx} + V\frac{dh}{dy}) + 12(w_h - w_0) \tag{3.16}$$

There are a few assumptions that are usually true and can further simplify Reynolds equation:

- With a proper selection of axis one of the velocities (U and V) can be equal to zero, here let $V = 0$
- Film thickness remains constant throughout the operation of the bearing $w_h - w_0 = 0$
- The viscosity of the lubricant is constant $\eta = const$ in isothermal models

Considering the operational conditions above the Reynolds equation is simplified to

$$\frac{\vartheta}{\vartheta x}(h^3\frac{\vartheta p}{\vartheta x}) + \frac{\vartheta}{\vartheta y}(h^3\frac{\vartheta p}{\vartheta y}) = 6U\eta\frac{dh}{dx} \tag{3.17}$$

which is the most common form of the Reynolds equation. In time-dependent simulations the lubricant changes in time as the operating conditions

change as well. In this case the term $\frac{dh}{dt}$ must be inserted in equation (3.17), yielding:

$$\frac{1}{\eta}[\frac{\vartheta}{\vartheta x}(\frac{h^3}{12}\frac{\vartheta p}{\vartheta x}) + \frac{\vartheta}{\vartheta y}(\frac{h^3}{12}\frac{\vartheta p}{\vartheta y})] = \frac{\vartheta h}{\vartheta t} + \frac{U}{2}\frac{dh}{dx} \quad (3.18)$$

3.3 Reynolds equation for hydrophobic surfaces

Hydrophobicity is a physical property of materials that exhibit very low shear at the fluid-solid boundary. When a drop of water is placed on a hydrophobic surface a high contact angle will be observed. The fundamental difference between a non-hydrophobic and a hydrophobic surface is that the no-slip boundary condition, where the velocity of the fluid molecules on the boundary should be equal to this of the solid wall, is no longer true. The molecules of the fluid have a different velocity than this of the wall when shear stress exceeds a critical value τ_c. In that case, the resulting slip velocity is proportional to the difference between the shear stress and the critical value

$$u_s = (\tau - \tau_c)\frac{b}{\eta} \quad (3.19)$$

In this equation, the slip length b represents a fictional distance below the solid wall, where the fluid velocity extrapolates to zero with a constant rate, equal to the velocity gradient at the fluid-wall boundary (Fig 3.5).

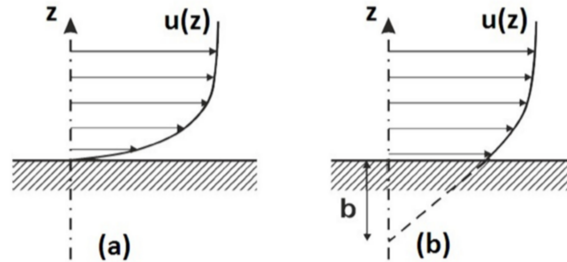


Figure 3.5: Velocity profile near the fluid-wall boundary with (a) no-slip condition and (b) slip condition with slip length **b**.

Excessive work has been done for inserting the proper modifications in the Reynolds equation in order to include the slip boundary conditions for bearings subjected to hydrophobic surface treatment, see e.g. (Raptis, 2014; Koukoulopoulos, 2014). The final form of the modified Reynolds equation for slip boundary conditions in the case of a journal bearing is presented below . The equation consists of five terms $(I - V)$ in the form of:

$$(I) + (II) = (III) + (IV) + (V) \quad (3.20)$$

where each part is analyzed below

$$(I) \rightarrow \frac{\vartheta}{\vartheta x}(\frac{h^3}{12\eta}\frac{h^2+4h\eta(a_S+a_h)+12\eta^2(a_S a_h)}{h(h+\eta(a_S+a_h))}\frac{\vartheta p}{\vartheta x})$$

$$(II) \rightarrow \frac{\vartheta}{\vartheta y}(\frac{h^3}{12\eta}\frac{h^2+4h\eta(a_S+a_h)+12\eta^2(a_S a_h)}{h(h+\eta(a_S+a_h))}\frac{\vartheta p}{\vartheta y})$$

$$(III) \rightarrow \frac{U}{2}\frac{\vartheta}{\vartheta x}(\frac{h^2+2h\eta a_h}{h+\eta(a_S+a_h)})+\frac{h}{2\eta}\frac{h\eta a_h+2\eta^2 a_S a_h}{h+\eta(a_S+a_h)}(\frac{\vartheta p}{\vartheta x}\frac{\vartheta h}{\vartheta x}+\frac{\vartheta p}{\vartheta y}\frac{\vartheta h}{\vartheta y})-U\frac{\eta a_h}{h+\eta(a_S+a_h)}\frac{\vartheta h}{\vartheta x}+\frac{\vartheta h}{\vartheta t}$$

$$(IV) \rightarrow \frac{1}{2}\frac{\vartheta}{\vartheta x}(\frac{h^2 a_S\tau_{C,S}+2a_S\eta a_h h\tau_{C,h}-h^2 a_h\tau_{C,h}}{h+\eta(a_S+a_h)})+\frac{h a_h\tau_{C,h}-\eta a_S a_h\tau_{C,S}+\eta a_S a_h\tau_{C,h}}{h+\eta(a_S+a_h)}\frac{\vartheta h}{\vartheta x}$$

$$(V) \rightarrow -\frac{1}{2}\frac{\vartheta}{\vartheta y}(\frac{(a_S\tau_{C,S}+a_h\tau_{C,h})h^2+2h\eta a_S a_h(\tau_{C,h}+\tau_{C,S})}{h+\eta(a_S+a_h)})+\frac{h a_h\tau_{C,h}+\eta a_S a_h(\tau_{C,h}+\tau_{C,S})}{h+\eta(a_S+a_h)}\frac{\vartheta h}{\vartheta y}$$

where

- Subscripts S and h are used to characterize the geometry that they refer to $shaft$ or $housing$ respectively.
- a_S and a_h are the slip parameters, (ratios of the the slip length to viscosity ratio ($\frac{b}{\eta}$)) for the $shaft$ and $housing$ geometries, respectively.
- $\tau_{C,S}$ and $\tau_{C,h}$ are the corresponding shear stress critical values.

3.4 Cavitation in journal bearings

Cavitation is the formation of vapor or air cavities in the liquid film. This phenomenon occurs when lubricant pressure gradients falls below cavitation pressure and as a result, the lubricating film breaks into streamers of lubricant and air. As film thickness increases, the space occupied by the lubricant reduces. The balance between the two volumes, is determined by the demand for continuity of flow and is expressed by the density of the lubricant-gas mixture ρ. The Reynolds equation for constant pressure is reduced in the following form for the cavitation region (time-dependent state):

$$\frac{\vartheta(\rho h)}{\vartheta t}+\frac{U}{2}\frac{\vartheta(\rho h)}{\vartheta x}=0 \tag{3.21}$$

For the region of the bearing with positive pressure, the lubricant density is assumed to be constant and the Reynolds equation is that presented in eq. (3.20).

When cavitation zones are present in journal bearings with hydrophobic surface regions, the modified version of the Reynolds equation is of the form

$$(I)+(II)+(III)+(IV)+(V)=0 \tag{3.22}$$

where the factors can be analyzed to

$$(I) \rightarrow \frac{U}{2}\frac{\vartheta}{\vartheta x}(\rho\frac{h^2+2h\eta a_h}{h+\eta(a_S+a_h)}) - U\frac{\rho\eta a_h}{h+\eta(a_S+a_h)}\frac{\vartheta h}{\vartheta x} + \frac{\vartheta(\rho h)}{\vartheta t}$$

$$(II) \rightarrow \frac{1}{2}\frac{\vartheta}{\vartheta x}(\rho\frac{h^2 a_S\tau_{C,S} + 2a_S\eta a_h h\tau_{C,S} - 2a_S\eta a_h h\tau_{C,h} - h^2 a_h\tau_{C,h}}{h+\eta(a_S+a_h)})$$

$$(III) \rightarrow \rho\frac{ha_h\tau_{C,h} - \eta a_S a_h\tau_{C,S} + \eta a_S a_h\tau_{C,h}}{h+\eta(a_S+a_h)}\frac{\vartheta h}{\vartheta x}$$

$$(IV) \rightarrow -\frac{1}{2}\frac{\vartheta}{\vartheta y}(\frac{(\rho a_S\tau_{C,S} + \rho a_h\tau_{C,h})h^2 + 2h\rho\eta a_S a_h(\tau_{C,S}+\tau_{C,h})}{h+\eta(a_S+a_h)})$$

$$(V) \rightarrow \rho\frac{ha_h\tau_{C,h} + \eta a_S a_h(\tau_{C,S}+\tau_{C,h})}{h+\eta(a_S+a_h)}\frac{\vartheta h}{\vartheta y}$$

Chapter 4

Computational Methods

4.1 Introduction

In order to solve the equations described in Chapter 3, special computer programs must be developed. Several numerical methods can be used and with the appropriate algorithms, these methods can aid in simulating the involved complex fluid phenomena. In the present work, the finite difference method is chosen for numerical simulations. First, the computational domain must be discretized. Discretization of equations leads to a linear system which can be solved using an iterative method such as the Jacobi or Gauss-Seidel method. When the converged pressure field is obtained, post-processing can lead to the final results and the problem is solved. In this chapter, all the necessary steps that lead to the solution of the equations are presented.

4.2 The finite difference method

In order to solve the differential equations that describe the phenomenon, the finite difference method is chosen. In this method, the computational space is discretized by a finite number of nodes. The derivatives of the differential equations are expressed as discrete quantities of dependent and independent variables in the neighboring nodes of the current grid point and the result is a system of algebraic equations that can be solved through an iterative solver. There are other ways to solve the system of algebraic equations, but in our case the successive over-relaxation form of the Gauss-Seidel iterative solver is utilized. The computational grid is uniform. That means that all the nodes have an equal distance from their neighbors in the x and y directions as shown below (Fig 4.1). If L is the length of the bearing and B the breadth then:

$$dx = \frac{L}{N_i - 1} \; , \; dy = \frac{B}{N_j - 1} \tag{4.1}$$

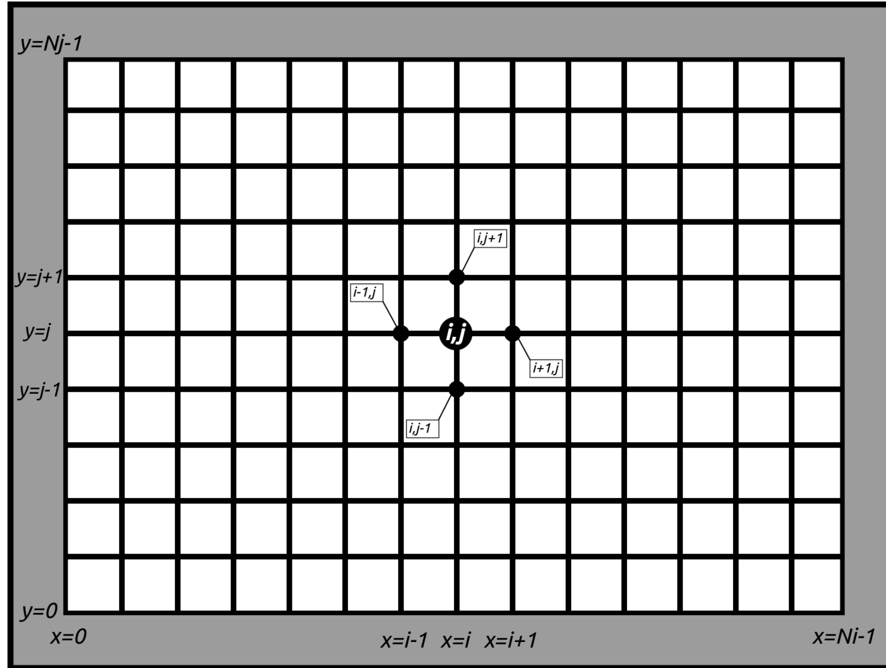


Figure 4.1: Finite difference method: The computational grid.

where N_i and N_j are the number of nodes in the x and y direction, respectively. The first order derivative at a point x of a function $u(x)$ is:

$$\frac{\vartheta u(x)}{\vartheta x} = \lim_{\Delta x \to 0} \frac{u(x+\Delta x) - u(x)}{\Delta x} \tag{4.2}$$

Using a Taylor series we can expand $u(x+\Delta x)$ about $u(x)$, which yields:

$$u(x+\Delta x) = u(x) + \Delta x \frac{\vartheta u(x)}{\vartheta x} + \frac{(\Delta x)^2}{2} \frac{\vartheta^2 u(x)}{\vartheta x^2} + \frac{(\Delta x)^3}{3!} \frac{\vartheta^3 u(x)}{\vartheta x^3} + ... \tag{4.3}$$

It is clear from the previous equations that

$$\frac{u(x+\Delta x) - u(x)}{\Delta x} = \frac{\vartheta u(x)}{\vartheta x} + \frac{\Delta x}{2} \frac{\vartheta^2 u(x)}{\vartheta x^2} + \frac{(\Delta x)^2}{3!} \frac{\vartheta^3 u(x)}{\vartheta x^3} + ... = \frac{\vartheta u(x)}{\vartheta x} + O(\Delta x) \tag{4.4}$$

In equation 4.4, $O(\Delta x)$ represents the truncation error. The finer the computational grid (when dx goes to zero), the smallest the truncation error of approximation. For a specific i node in the domain, the first order derivative can be approximated through this finite difference discretization in three different ways:

- Forward difference approximation: In this case the function derivative in the $i-th$ node is calculated through the value of the function in the $i+1$ and i nodes.

$$(\frac{\vartheta u(x)}{\vartheta x})_i = \frac{u_{i+1} - u_i}{\Delta x} + O(\Delta x) \tag{4.5}$$

- Backward difference approximation: Where the derivative depends on the function values at the i and $i-1$ nodes.

$$(\frac{\vartheta u(x)}{\vartheta x})_i = \frac{u_i - u_{i-1}}{\Delta x} + O(\Delta x) \tag{4.6}$$

- Central difference approximation: Which is a combination of the above methods and considers the function values at the $i+1$ and $i-1$ nodes.

$$(\frac{\vartheta u(x)}{\vartheta x})_i = \frac{u_{i+1} - u_{i-1}}{2\Delta x} + O(\Delta x^2) \tag{4.7}$$

  where dx is the constant distance between two nodes in the x-direction from (4.1).

It is clear that the forward and backward difference approximations produce a first order truncation error, while the central difference approximation a second order truncation error. In many cases. second or higher order derivatives are present in the differential equation. A finite difference approximation of a second order derivative can be obtained in a similar way, using the forward and backward first order derivative equations:

$$\frac{u_{i+1} - u_i}{\Delta x} - \frac{\Delta x}{2!}(\frac{\vartheta^2 u}{\vartheta x^2}) - \frac{(\Delta x)^2}{3!}(\frac{\vartheta^3 u}{\vartheta x^3}) - ... = \frac{u_i - u_{i-1}}{\Delta x} + \frac{\Delta x}{2!}(\frac{\vartheta^2 u}{\vartheta x^2}) - \frac{(\Delta x)^2}{3!}(\frac{\vartheta^3 u}{\vartheta x^3}) + ... \tag{4.8}$$

and the next step is to calculate the second order derivative as

$$\frac{\vartheta^2 u}{\vartheta x^2} = \frac{u_{i+1} - 2u_i + u_{i+1}}{dx^2} + O(dx^2) \tag{4.9}$$

If a two-dimension domain is examined, the same equations are valid in the y-direction, following the same steps, considering a $v(y)$ function:

$$\frac{\vartheta v(y)}{\vartheta y} = \lim_{\Delta y \to 0} \frac{v(y + \Delta y) - v(y)}{\Delta y} \tag{4.10}$$

$$\frac{v(y + \Delta y) - v(y)}{\Delta y} = \frac{\vartheta v(y)}{\vartheta y} + O(\Delta y) \tag{4.11}$$

$$(\frac{\vartheta v(y)}{\vartheta y})_j = \frac{v_{j+1} - v_j}{\Delta y} + O(\Delta y) \tag{4.12}$$

$$(\frac{\vartheta v(y)}{\vartheta y})_j = \frac{v_j - v_{j-1}}{\Delta y} + O(\Delta y) \tag{4.13}$$

$$(\frac{\vartheta v(y)}{\vartheta y})_j = \frac{v_{j+1} - v_{j-1}}{2\Delta y} + O(\Delta y^2) \tag{4.14}$$

and for the second order derivative in the y-direction

$$\frac{\vartheta^2 v}{\vartheta y^2} = \frac{v_{j+1} - 2v_j + v_{j-1}}{\Delta y^2} + O(d\Delta y) \tag{4.15}$$

4.3 Discretization of the basic equations

This section describes the part of the mathematical analysis, in order to derive in the final form of the discretized equations that will be used in this work. First, the discretization of the basic Reynolds equation is presented. Then, alternative forms of the same equation are given, for cases where a surface of the bearing has been, fully or partially, modified with hydrophobic surface treatment methods or cavitation phenomena are present. In some cases, a part of the mathematical procedure has not been presented for brevity

4.3.1 Discretization of the Reynolds equation for simple bearings

The basic Reynolds equation used in the case of a simple bearing is:

$$\frac{1}{\eta}[\frac{\vartheta}{\vartheta x}(\frac{h^3}{12}\frac{\vartheta p}{\vartheta x}) + \frac{\vartheta}{\vartheta y}(\frac{h^3}{12}\frac{\vartheta p}{\vartheta y})] = \frac{\vartheta h}{\vartheta t} + \frac{U}{2}\frac{dh}{dx} \tag{4.16}$$

On the right hand side of this equation the time-dependent term is used to describe phenomena that vary in time. In steady-state form this term is neglected. Using the derivative of product rule

$$h'(x) = [f(x)g(x)]' = f'(x)g(x) + f(x)g'(x) \tag{4.17}$$

eq. (4.17) can be discretized with a central difference scheme:

$$\frac{\vartheta}{\vartheta x}(\frac{h^3}{12}\frac{\vartheta p}{\vartheta x}) = \frac{1}{12}\frac{\vartheta h^3}{\vartheta x}\frac{\vartheta p}{\vartheta x} + \frac{h^3}{12}\frac{\vartheta^2 p}{\vartheta x^2} = \frac{h^3_{i+1,j} - h^3_{i-1,j}}{24dx}\frac{p_{i+1,j} - p_{i-1,j}}{dx} + \frac{h^3_{i,j}}{12}\frac{p_{i+1,j} - 2p_{i,j} + p_{i,j-1}}{dx^2} \tag{4.18}$$

$$\frac{\vartheta}{\vartheta y}(\frac{h^3}{12}\frac{\vartheta p}{\vartheta y}) = \frac{1}{12}\frac{\vartheta h^3}{\vartheta y}\frac{\vartheta p}{\vartheta y} + \frac{h^3}{12}\frac{\vartheta^2 p}{\vartheta y^2} = \frac{h^3_{i,j+1} - h^3_{i-1,j}}{24dy}\frac{p_{i,j+1} - p_{i-1,j}}{dy} + \frac{h^3_{i,j}}{12}\frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{dy^2} \tag{4.19}$$

and for the right hand side

$$\frac{U}{2}\frac{dh}{dx} = \frac{U}{2}\frac{h_{i+1,j} - h_{i-1,j}}{2dx} \tag{4.20}$$

Regarding the time-dependent term, a second-order backward difference to gain information from the two previous time steps is used. If m is the current time step and $m-1$, $m-2$ the two previous steps respectively:

$$\frac{\vartheta h}{\vartheta t} = \frac{3h_{i,j}^{t=m} - 4h_{i,j}^{t=m-1} + h_{i,j}^{t=m-2}}{2\Delta t} \tag{4.21}$$

The last step to arrive to the system of linear equations is the factorization of the pressure terms (as the pressure field is the unknown parameter) based on the grid point that they refer to. In this discretization, a five point method with the current node (i, j) updating its value in each iteration from the four neighboring elements: $(i+1, j)$, $(i-1, j)$, $(i, j+1)$ and $(i, j-1)$ is implemented (Fig 4.1). The result of the factorization is the following:

$$p_{i,j} \to -\frac{h_{i,j}^3}{12\eta dx^2} - \frac{h_{i,j}^3}{12\eta dy^2} \tag{4.22}$$

$$p_{i+1,j} \to \frac{h_{i+1,j}^3 - h_{i-1,j}^3}{48\eta dx^2} + \frac{h_{i,j}^3}{12\eta dx^2} \tag{4.23}$$

$$p_{i-1,j} \to -\frac{h_{i+1,j}^3 - h_{i-1,j}^3}{48\eta dx^2} + \frac{h_{i,j}^3}{12\eta dx^2} \tag{4.24}$$

$$p_{i,j+1} \to \frac{h_{i,j+1}^3 - h_{i,j-1}^3}{48\eta dy^2} + \frac{h_{i,j}^3}{12\eta dy^2} \tag{4.25}$$

$$p_{i,j-1} \to -\frac{h_{i,j+1}^3 - h_{i,j-1}^3}{48\eta dy^2} + \frac{h_{i,j}^3}{12\eta dy^2} \tag{4.26}$$

and for the right hand side

$$\frac{3h_{i,j}^{t=m} - 4h_{i,j}^{t=m-1} + h_{i,j}^{t=m-2}}{2\Delta t} + \frac{U}{2}\frac{h_{i+1,j} - h_{i-1,j}}{2dx} \tag{4.27}$$

The equation can now be written in the form

$$\mathbf{A}p_{i+1,j} + \mathbf{B}p_{i-1,j} + \mathbf{C}p_{i,j+1} + \mathbf{D}p_{i,j-1} + \mathbf{E}p_{i,j} = \mathbf{F} \tag{4.28}$$

where the factor $(i \times j)$ matrices $\mathbf{A}$ to $\mathbf{E}$ are multiplied with the appropriate element relative to the current (i, j) node. The matrices are derived from the previous equations (4.22 - 4.26).

$$\mathbf{A}_{i,j} = \frac{h_{i+1,j}^3 - h_{i-1,j}^3}{48\eta dx^2} + \frac{h_{i,j}^3}{12\eta dx^2}$$

$$\mathbf{B}_{i,j} = -\frac{h_{i+1,j}^3 - h_{i-1,j}^3}{48\eta dx^2} + \frac{h_{i,j}^3}{12\eta dx^2}$$

$$\mathbf{C}_{i,j} = \frac{h_{i,j+1}^3 - h_{i,j-1}^3}{48\eta dy^2} + \frac{h_{i,j}^3}{12\eta dy^2}$$

$$\mathbf{D}_{i,j} = -\frac{h_{i,j+1}^3 - h_{i,j-1}^3}{48\eta dy^2} + \frac{h_{i,j}^3}{12\eta dy^2}$$

$$\mathbf{E}_{i,j} = -\frac{h_{i,j}^3}{12\eta dx^2} - \frac{h_{i,j}^3}{12\eta dy^2}$$

$$\mathbf{F}_{i,j} = \frac{3h_{i,j}^{t=m} - 4h_{i,j}^{t=m-1} + h_{i,j}^{t=m-2}}{2\Delta t} + \frac{U}{2}\frac{h_{i+1,j} - h_{i-1,j}}{2dx}$$

Our goal is to solve for pressure for every node (i, j), that is to solve the modified equation

$$p_{i,j} = \frac{\mathbf{F}_{i,j} - \mathbf{A}_{i,j}p_{i+1,j} - \mathbf{B}_{i,j}p_{i-1,j} - \mathbf{C}_{i,j}p_{i,j+1} - \mathbf{D}_{i,j}p_{i,j-1}}{\mathbf{E}_{i,j}} \tag{4.29}$$

Therefore, for every node of the computational grid, a linear equation is obtained.

4.3.2 Discretization of the Reynolds equation for bearings with hydrophobic surface treatment

The Reynolds equation for hydrophobic boundary conditions has been presented earlier (eq. (3.20))

In the present work, the critical shear stress values for both the $shaft$ and the *housing* are considered to be equal to zero

$$\tau_{C,S} = \tau_{C,h} = 0 \tag{4.30}$$

Using a central difference method as before and the procedure followed in the work of L.Raptis (Raptis, 2014), the discretized form of eq. (3.20) is derived:

$$p_{i,j} = \frac{\mathbf{F}_{i,j} - \mathbf{A}_{i,j}p_{i+1,j} - \mathbf{B}_{i,j}p_{i-1,j} - \mathbf{C}_{i,j}p_{i,j+1} - \mathbf{D}_{i,j}p_{i,j-1}}{\mathbf{E}_{i,j}}$$

where:

$$\mathbf{A}_{i,j} = \tfrac{1}{2dx}\left(\tfrac{var1_{i+1,j} - var1_{i-1,j}}{2dx} - var3_{i,j}\tfrac{h_{i+1,j} - h_{i-1,j}}{2dx}\right) + var1_{i,j}\tfrac{1}{dx^2}$$

$$\mathbf{B}_{i,j} = -\tfrac{1}{2dx}\left(\tfrac{var1_{i+1,j} - var1_{i-1,j}}{2dx} - var3_{i,j}\tfrac{h_{i+1,j} - h_{i-1,j}}{2dx}\right) + var1_{i,j}\tfrac{1}{dx^2}$$

$$\mathbf{C}_{i,j} = \tfrac{1}{2dy}\left(\tfrac{var1_{i,j+1} - var1_{i,j-1}}{2dy} - var3_{i,j}\tfrac{h_{i,j+1} - h_{i,j-1}}{2dy}\right) + var1_{i,j}\tfrac{1}{dy^2}$$

$$\mathbf{D}_{i,j} = -\tfrac{1}{2dy}\left(\tfrac{var1_{i,j+1} - var1_{i,j-1}}{2dy} - var3_{i,j}\tfrac{h_{i,j+1} - h_{i,j-1}}{2dy}\right) + var1_{i,j}\tfrac{1}{dy^2}$$

$$\mathbf{E}_{i,j} = -\tfrac{2var1_{i,j}}{dx^2} - \tfrac{2var1_{i,j}}{dy^2}$$

$$\mathbf{F}_{i,j} = \tfrac{U}{2}\tfrac{var2_{i+1,j} - var2_{i-1,j}}{2dx} - var4_{i,j}\tfrac{h_{i+1,j} - h_{i-1,j}}{2dx} + \tfrac{3h_{i,j}^{t=m} - 4h_{i,j}^{t=m-1} + h_{i,j}^{t=m-2}}{2\Delta t}$$

and the auxiliary matrix variables

$$var1_{i,j} = \tfrac{h_{i,j}^3}{12\eta}\tfrac{h_{i,j}^2 + 4h_{i,j}\eta(a_{Si,j} + a_{hi,j}) + 12\eta^2 a_{Si,j}a_{hi,j}}{h_{i,j}(h_{i,j} + \eta(a_{Si,j} + a_{hi,j}))}$$

$$var2_{i,j} = \tfrac{h_{i,j}^2 + 2h_{i,j}\eta a_{hi,j}}{h_{i,j} + \eta(a_{Si,j} + a_{hi,j})}$$

$$var3_{i,j} = \tfrac{h_{i,j}}{12\eta}\tfrac{h_{i,j}\eta a_{hi,j} + 2\eta^2 a_{Si,j}a_{hi,j}}{h_{i,j} + \eta(a_{Si,j} + a_{hi,j})}$$

$$var4_{i,j} = U\tfrac{\eta a_{hi,j}}{h_{i,j} + \eta(a_{Si,j} + a_{hi,j})}$$

4.3.3 Discretization of the Reynolds equation with the Elrod-Adams mass-conservation model for simple bearings

To account for cavitation in lubricated contacts, the Reynolds equation with the Elrod-Adams cavitation algorithm is utilized, see eq. (4.31). This equation is solved in both the active and the cavitating regions of the lubricant domain for the fractional film content variable r, which is defined as $r = \frac{\rho}{\rho_c}$. Here ρ_c is the density of the lubricant in the non-cavitating film regions at cavitation pressure p_c, and ρ is the actual density of the lubricant within the lubricant domain. In the active zones, where pressure is above cavitation pressure, the actual fluid density ρ is slightly larger than ρ_c , due to the fact that pressure increase will cause a slight compression of the lubricant and therefore an increase of its mass content ($r >= 1$). In this case, lubricant pressure can be calculated using Eq. (4.32), where β is the lubricant bulk modulus (Eq. 4.33). On the other hand, within the cavitation zones, where pressure attains a constant value p_c , the lubricant maintains a constant density ρ_c . However, due to the rupture of lubricant film into streamers of lubricant, vapor or air, the actual density of the lubricant-gas mixture ρ is smaller. Thus, the cavitation phenomenon results in a decrease of mass content in those regions, $r > 1$, whereas $(1 - r)$ represents the void (gas) fraction. To take the above into consideration, a switch function g , whose index is zero within the cavitation zones and unity in the active zones, is introduced, as seen in Eq. (4.34).

$$\frac{U}{2}\frac{\vartheta(rh)}{\vartheta x} + \frac{\vartheta(rh)}{\vartheta t} = \frac{\vartheta}{\vartheta x}(\frac{g(r)\beta h^3}{12\eta}\frac{\vartheta r}{\vartheta x}) + \frac{\vartheta}{\vartheta y}(\frac{g(r)\beta h^3}{12\eta}\frac{\vartheta r}{\vartheta y}) \tag{4.31}$$

$$p = p_c + g\beta lnr \tag{4.32}$$

$$\beta = \rho\frac{\vartheta p}{\vartheta r} \tag{4.33}$$

$$g = \begin{cases} 0 & ,if\ r < 1 \\ 1 & ,if\ r \geq 1 \end{cases} \tag{4.34}$$

The derivatives of the r variable can be computed as (Vijayaraghavan and Keith, 1989)

$$g\frac{\vartheta r}{\vartheta x} = \frac{\vartheta g(r-1)}{\vartheta x} \tag{4.35}$$

$$g\frac{\vartheta r}{\vartheta y} = \frac{\vartheta g(r-1)}{\vartheta y} \tag{4.36}$$

and these equations can be substituted in (4.31). Values on the half step can be computed using the following equations

$$f_{i\pm\frac{1}{2},j} = \frac{f_{i\pm 1,j} + f_{i,j}}{2} \tag{4.37}$$

$$f_{i,j\pm\frac{1}{2}} = \frac{f_{i,j\pm1} + f_{i,j}}{2} \tag{4.38}$$

Elrod has proposed an equation for the convective mass flux

$$(m_x)_{convective} = \rho_c \frac{U}{2}(r_{i-1,j}h_{i-1,j}(1-g_{i-1,j})+g_{i-1,j}h_{i-1,j}+g_{i,j}g_{i-1,j}\frac{h_{i,j}-h_{i-1,j}}{2}) \tag{4.39}$$

to consider the following cases

- If the upstream point is in the cavitation zone $g_{i-1,j} = 0$ then

$$(m_x)_{convective} = \rho_c \frac{U}{2} r_{i-1,j}h_{i-1,j}$$

- If the upstream point is in the active zone $g_{i-1,j} = 1$ and the downstream point in the cavitation zone $g_{i,j} = 0$

$$(m_x)_{convective} = \rho_c \frac{U}{2} h_{i-1,j}$$

- If both points are in the active zone

$$(m_x)_{convective} = \rho_c \frac{U}{2} \frac{h_{i-1,j}+h_{i,j}}{2}$$

Using mathematical analysis (Raptis, 2014) and after rearranging the shear flow term and the time derivative term the equation reaches the known form

$$\mathbf{A}r_{i+1,j} + \mathbf{B}r_{i-1,j} + \mathbf{C}r_{i,j+1} + \mathbf{D}r_{i,j-1} + \mathbf{E}r_{i,j} = \mathbf{F}$$

with the factor matrices being

$$\mathbf{A}_{i,j} = \frac{\beta}{12\eta\Delta x^2} g_{i+1,j} h^3_{i+\frac{1}{2},j}$$

$$\mathbf{B}_{i,j} = \frac{\beta}{12\eta\Delta x^2} g_{i-1,j} h^3_{i-\frac{1}{2},j} + \frac{U}{2\Delta x} h_{i-1,j}(1-g_{i-1,j})$$

$$\mathbf{C}_{i,j} = \frac{\beta}{12\eta\Delta y^2} g_{i,j+1} h^3_{i,j+\frac{1}{2}}$$

$$\mathbf{D}_{i,j} = \mathbf{C}_{i,j} = \frac{\beta}{12\eta\Delta y^2} g_{i,j-1} h^3_{i,j-\frac{1}{2}}$$

$$\mathbf{E}_{i,j} = -\frac{\beta}{12\eta\Delta x^2} g_{i,j}(h^3_{i+\frac{1}{2},j}+h^3_{i-\frac{1}{2},j}) - \frac{\beta}{12\eta\Delta y^2} g_{i,j}(h^3_{i,j+\frac{1}{2}}+h^3_{i,j-\frac{1}{2}}) - \frac{U}{2\Delta x} h_{i,j}(1-g_{i,j}) - (\frac{dh}{dt}+\frac{h_{i,j}}{2\Delta t})$$

$$\mathbf{F}_{i,j} = \frac{\beta}{12\eta\Delta x^2}(g_{i+1,j}h^3_{i+\frac{1}{2},j} - g_{i,j}(h^3_{i+\frac{1}{2},j} + h^3_{i-\frac{1}{2},j}) + g_{i-1,j}h^3_{i-\frac{1}{2},j})$$
$$+\frac{\beta}{12\eta\Delta y^2}(g_{i,j+1}h^3_{i,j+\frac{1}{2}} - g_{i,j}(h^3_{i,j+\frac{1}{2}} + 1h^3_{i,j-\frac{1}{2}}) + g_{i,j-1}h^3_{i,j-\frac{1}{2}})$$

$$-\frac{U}{2\Delta x}[\frac{g_{i-1,j}h_{i-1,j}}{2}(2-g_{i,j})+\frac{g_{i,j}h_{i,j}}{2}(g_{i-1,j}-2+g_{i+1,j})-\frac{g_{i+1,j}g_{i,j}h_{i+1,j}}{2}]+h_{i,j}\frac{-4\theta_{i,j}|_{t-1}+\theta_{i,j}|_{t-1}}{2\Delta t}$$

where

$$\Delta x = x_{i+1} - x_i = x_i - x_{i-1} = x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}}$$

$$\Delta y = y_{j+1} - y_j = y_j - y_{j-1} = y_{j+\frac{1}{2}} - y_{j-\frac{1}{2}}$$

Pressure at every node can be calculated by:

$$p_{i,j} = \frac{\mathbf{F}_{i,j} - \mathbf{A}_{i,j}p_{i+1,j} - \mathbf{B}_{i,j}p_{i-1,j} - \mathbf{C}_{i,j}p_{i,j+1} - \mathbf{D}_{i,j}p_{i,j-1}}{\mathbf{E}_{i,j}}$$

4.4 The iterative solver

In all cases, a linear system of equations (in matrix form) is obtained. The system needs to be solved for pressure. There are various ways to solve such a system but in our case an iterative solver is chosen. An iterative method is a mathematical procedure that generates a sequence of improving estimates that lead to the final solution of the system when a convergence criterion is met. The other large category of solvers include the direct methods, which attempt to solve the problem by a finite sequence of operations, without rounding errors. These methods are often computationally expensive so the iterative methods are chosen when the number of variables is very large or when the system consists of non-linear equations. In our effort to analyze the process followed by the solver to reach a final solution, the simplest approach is used as a starting point. That is the *Jacobi* method. This method performs a number of successive iterations to continuously update the value of each node using the final form of the equations derived previously in the following form

$$p^{[k]}_{i,j} = \frac{\mathbf{F}_{i,j} - \mathbf{A}_{i,j}p^{[k-1]}_{i+1,j} - \mathbf{B}_{i,j}p^{[k-1]}_{i-1,j} - \mathbf{C}_{i,j}p^{[k-1]}_{i,j+1} - \mathbf{D}_{i,j}p^{[k-1]}_{i,j-1}}{\mathbf{E}_{i,j}} \tag{4.40}$$

where $p^{[k]}$ is the value of pressure for the current iteration and $p^{[k-1]}$ is the value of pressure for the previous iteration. By monitoring the error using an absolute error convergence criterion such as

$$|p^{[k]}_{i,j} - p^{[k-1]}_{i,j}| < d\epsilon, \forall(i,j) \tag{4.41}$$

where $d\epsilon$ is a very small number, we can decide if the solution for the pressure filed is accurate enough and stop the iterative process. Several improvements can be made to make the *Jacobi* method converge faster (in a smaller number of iterations). For example one can think that in the process of the *for loop* that is used to update the current node (i,j) , the values of two out of the four neighboring nodes have already been updated (in this case $(i-1,j)$ and $(i,j-1)$). If we use the already updated values for our current iteration step, the iterative solver is of the form

$$p^{[k]}_{i,j} = \frac{\mathbf{F}_{i,j} - \mathbf{A}_{i,j}p^{[k-1]}_{i+1,j} - \mathbf{B}_{i,j}p^{[k]}_{i-1,j} - \mathbf{C}_{i,j}p^{[k-1]}_{i,j+1} - \mathbf{D}_{i,j}p^{[k]}_{i,j-1}}{\mathbf{E}_{i,j}} \tag{4.42}$$

and seems to converge much faster than the previous one. This modification is known as the *Gauss − Seidel* method and improves the performance of our algorithms. A further optimization can be achieved by enforcing a relaxation factor on the *Gauss−Seidel* solver. This factor (ω) is called an under-relaxation factor if $0 < \omega < 1$ or an over-relaxation factor if $\omega > 1$. This new iterative method is called *Successive Over − Relaxation* or simply *SOR* and has the following form

$$p_{i,j}^{[k]} = (1-\omega)p_{i,j}^{[k-1]} + \omega \cdot \frac{\mathbf{F}_{i,j} - \mathbf{A}_{i,j}p_{i+1,j}^{[k-1]} - \mathbf{B}_{i,j}p_{i-1,j}^{[k]} - \mathbf{C}_{i,j}p_{i,j+1}^{[k-1]} - \mathbf{D}_{i,j}p_{i,j-1}^{[k]}}{\mathbf{E}_{i,j}} \quad (4.43)$$

It is clear from comparing the equations (4.42) and (4.43) that when $\omega = 1$ the *SOR* method degenerates to the *Gauss − Seidel* method. We will see in the following sections that the choice of the relaxation factor plays decisive role in the convergence speed and consequently, the performance of our program.

Chapter 5

High Performance Computing

5.1 Introduction

In this chapter, the significance of high performance computing for hard and time consuming computational problems will be discussed. Starting with examples which demonstrate why parallel programming is the only way to solve problems that seemed impossible to solve, the basics of the parallel programming models that are used in the context of this thesis will be presented. Then the characteristics of these models will be analyzed, as well as examples on how to implement specific algorithms and develop applications on these different models and the potential benefits that each one can offer are presented. Finally, a theoretical comparison is provided.

5.2 The age of parallel computing

Since the first computers were put together, computer scientists and engineers tried to find ways to increase their computational power. At first this procedure was quite straightforward, as by increasing the CPU clock frequency and/or decreasing she size of transistors to fit more of them in a certain CPU size, the performance is dramatically increased. The 1990's were the golden era of performance scaling for single core processors. Between 1994 and 1998 CPU clock speeds increased by 300%. Regarding the transistor size it is useful to mention that an Intel Pentium 4 processor from 2004 had 170 million transistors, while an 15-Core Intel Xeon processor of 2013 fits 4.3 billion transistors for roughly the same size. Transistors are the electronic devices that are found in every processor, which act as electronic switches in a binary manner (1 or 0) ,and, for different combinations of these values, form logical gates able to perform logical and arithmetic operations. The CPU clock frequency is the frequency at which these devices can switch from one to zero without losing their computational integrity. For example, a processor that operates on a frequency of 1GHz is capable of performing 1 billion operations per second and

still work as expected.

There are a few reasons why these parameters were not able to contribute to an increase of computational power beyond a certain threshold. The first one is that the size of transistors cannot get infinitely small. When the size of these devices becomes smaller, the logical gates become thinner and their structural integrity is lost, resulting in current leaks. Besides that, when the frequency speed increases and millions or even billions of transistors operate on a very high speeds, heat is generated. This heat leads to an increase in temperature that can damage the processor beyond repair. Increase in frequency also means higher voltage and the relation between voltage and power consumption is cubic. That means that by decreasing the CPU frequency by 50%, the power consumption is 12.5% of the original. Nowadays, energy efficiency is an even more important matter, therefore it is clear that prodigious clock frequencies are unacceptable.

The only viable solution to this problem, is increasing the number of cores per CPU. When more than one processors that can independently execute instructions at a lower frequency are utilized, a significant improvement in performance can be achieved, while remaining energy efficient. In our days, most personal computers are equipped with multicore processors, featuring up to 16 cores per unit. Finding new ways to exploit the potential performance of these machines can lead to a significant increase in the performance of common computational tasks. Parallel programming gives the opportunity to take advantage of modern hardware. Besides multicore CPUs, there is another large group of processors that are also suitable for parallel computations. Graphics Processing Units (GPUs) are electronic circuits commonly used for computer graphics and image processing as they rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer, to output to a display. They are highly parallel units that achieve massive throughput and are more efficient than CPUs for programs that require the manipulation of very large blocks of data. It is noted that a commercially available GPU can have from several hundreds to several thousands of computational cores. The differences in the architecture of these two processing units will be analyzed further in this work.

5.3 The serial code

Before applying the parallel programming methods it is useful to set up a reference case of serial execution code. This reference case corresponds to the solution of the simple Reynolds equation for a thrust bearing. Solution accuracy of the developed code was validated against the model of a simple thrust bearing; see (Stachowiak and Batchelor, 2001) (Table 5.1 and Fig 5.1).

$\frac{\mathbf{L}}{\mathbf{B}} = \mathbf{0.5}$				
k=1	**k=2**	**k=3**	**k=4**	**k=5**
0.0291096	0.0297203	0.0270916	0.0242234	0.021646
$\frac{\mathbf{L}}{\mathbf{B}} = \mathbf{0.75}$				
k=1	**k=2**	**k=3**	**k=4**	**k=5**
0.0505747	0.0501036	0.0444461	0.0388181	0.0339928
$\frac{\mathbf{L}}{\mathbf{B}} = \mathbf{1}$				
k=1	**k=2**	**k=3**	**k=4**	**k=5**
0.0691179	0.0672483	0.0587022	0.0505692	0.0437666
$\frac{\mathbf{L}}{\mathbf{B}} = \mathbf{1.5}$				
k=1	**k=2**	**k=3**	**k=4**	**k=5**
0.0948963	0.0905986	0.0777800	0.066069	0.0565053
$\frac{\mathbf{L}}{\mathbf{B}} = \mathbf{2}$				
k=1	**k=2**	**k=3**	**k=4**	**k=5**
0.1101110	0.104198	0.0887689	0.0749183	0.0637285
$\frac{\mathbf{L}}{\mathbf{B}} = \infty$				
k=1	**k=2**	**k=3**	**k=4**	**k=5**
0.157077	0.1458990	0.1222289	0.1018020	0.0855938

Table 5.1: Simple converging thrust bearing model. Non-dimensional load capacity for different values of convergence ratio k and length to width ratio of the bearing.

In Section 4.4 we discussed about the implementation of the successive over-relaxation (SOR) method. The importance of the ω relaxation factor is examined for the simple bearing model (Table 5.2). In particular, parametric analysis has been conducted and the number of iterations needed for final convergence are presented as a function of the relaxation factor. The SOR method, with a proper selection of ω is finally compared to the number of iterations needed for the basic *Jacobi* method, to display the improved performance that justifies the choice of the SOR iterative method (Fig 5.2).

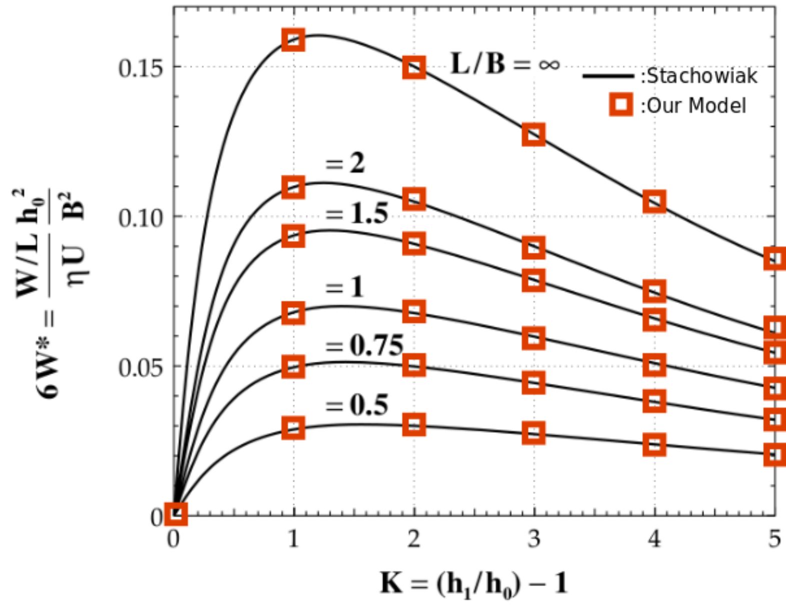


Figure 5.1: Non-dimensional load capacity as a function of convergence ratio, for different values of length-to-width ratio of the bearing. Comparison between the results of the present code and those presented in (Stachowiak and Batchelor, 2001).

Relaxation Factor	No. of Iterations
0.98	65811
0.99	55245
1	46500
1.1	39110
1.2	32752
1.3	27200
1.4	22283
1.5	17875
1.6	13872
1.7	10189
1.8	6744
1.9	3429
1.92	2762
1.94	2083
1.96	1367
1.98	**682**
1.99	1296

Table 5.2: Influence of the value of relaxation factor ω on convergence speed.

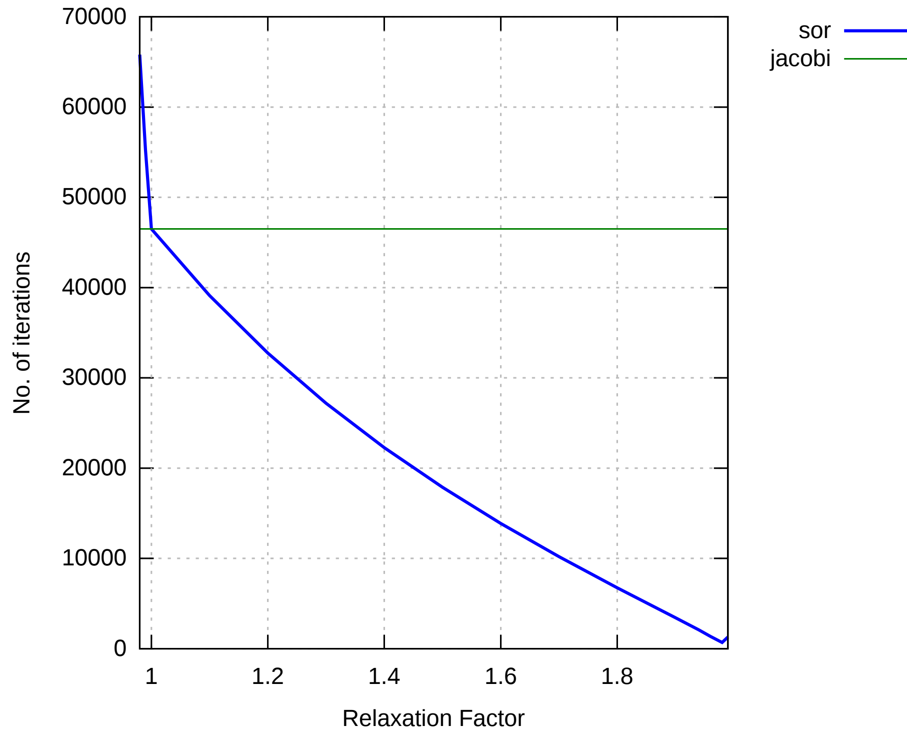


Figure 5.2: Influence of the value of relaxation factor ω on convergence speed. Performance of the Jacobi algorithm is also presented for comparison.

A very important step before trying to parallelize our code is *profiling*. Profilers, are programs that measure the execution time for each function that is called in an executable. It is crucial to have information on how long each function or fraction of the code takes to execute, as this will help to choose which part of the program needs to be parallelized to drastically improve the overall performance. Profilers range from simple non-intrusive time counting programs that provide information about the called functions, to complex, intrusive programs that are able to give much more detail on performance thresholds in both serial and parallel applications. For this example *gprof* was used. This profiler is part of the GNU operating system and is an open source utility. The present simple Reynolds solver consists of the following functions:

- *calculateGeometry*(): generates lubricant film geometry
- *reynolds*(): initializes the factor matrices (**A, B, C, D, E** & **F**) for the iterative solver (See sections 4.3.1-4.3.3)
- *sor*(): performs the iterative solution procedure

- *pressureIntegration*(): integrates over the pressure and shear stress to calculate load capacity and friction force
- *report*(): generates various result text files

The profiler output gives an overview of the time each function consumed as a percentage of the overall execution time in seconds (Table 5.3). It can be seen that the *sor*() function (iterative solver), makes up almost the entire runtime (99.2%). So it would be redundant to spend any programming effort for parallelizing any of the other functions. This example demonstrates the importance of profiling an application before optimizing it, and the value of this type of information.

Function	**Percentage of execution time [%]**
calculateGeometry()	0.10
reynolds()	0.42
sor()	**99.20**
pressureIntegration()	0.14
report()	0.14

Table 5.3: Profiling details of the Reynolds equation solver for simple converging bearings.

Following the profiling results, in the present work, only the SOR routines have been parallelized in order to gain performance benefits. This gives a very small overhead when comparing the execution times as not the entire code is parallelized. It is useful in this point to present the basic serial version of the present SOR routine (Fig 5.3).

In this routine, the SOR solver is used to solve the basic Reynolds equation (4.28) and define the pressure field. The variables are defined below:

- *ldiv*: The number of nodes length-wise.
- *bdiv*: The number of nodes breadth-wise.
- $difference$: the error of the current iteration step. In this implementation the error is calculated by adding the relative error of each node for the entire field and then dividing by the number of nodes. So an average relative error value is extracted. When the error drops below an acceptable accuracy threshold (here 10^{-7}) the iteration process stops as we consider that the solver has converged.
- *counter*: The number of the current iteration. We stop the process if this number exceeds 100000 as this is the maximum convergence limit that we have set to prevent infinite loops.
- $p[i][j]$: An array that holds the pressure values at each node.

```
while ((difference>0.0000001) && (counter<100000))
{
     difference=0.;
     for (i=1;i<ldiv-1;i++)
     {
         for (j=1;j<bdiv-1;j++)
         {
             p[i][j]=(1-omega)*pPrevious[i][j]
             +((omega/sorE[i][j])
             *(sorF[i][j]
             -sorA[i][j]*pPrevious[i+1][j]
             -sorB[i][j]*p[i-1][j]
             -sorC[i][j]*pPrevious[i][j+1]
             -sorD[i][j]*p[i][j-1]));
             if (p[i][j]<0.)
             {
               p[i][j]=0.;
             }
             difference+=(fabs(pPrevious[i][j]-p[i][j]))/p[i][j];

             pPrevious[i][j]=p[i][j];
         }
     }
     difference=difference/(ldiv*bdiv);
     counter++;
```

Figure 5.3: Algorithm of the serial SOR iterative solver.

- $pPrevious[i][j]$: An array that holds the pressure values from the previous iteration at each node. This array is essential for the convergence check.
- $sorA[i][j], sorB[i][j], ..., sorF[i][j]$: The factor arrays from the analysis that led to equation (4.28).
- $omega$: The relaxation factor.

This code segment in various forms is the basis for solving the differential equation and computing the pressure field. This routine will be modified, to fit our purposes on parallel computing and accelerate computations.

5.4 Parallel programming models

The parallel programming models that were implemented in the present work were two. The Open Multi-Processing ($OpenMP$) shared memory API and the Compute Unified Device Architecture ($CUDA$) model for accelerated computations on Graphics Processing Units (GPUs). Here, the gain in performance

by utilizing these architectures is examined. These methods will be evaluated mainly in terms of execution speedup in comparison to the speed of the reference, and other factors such as required programming effort and acquisition cost of the necessary hardware. Beyond that, the basic features of each programming approach are discussed and a comparison between the advantages and disadvantages for each case are provided. Also, the Message Passing Interface (MPI) for cluster computing with distributed memory nodes and Hybrid Models will be briefly discussed, for these two paradigms have not been assessed in the context of the current thesis.

5.4.1 Open MultiProcessing (OpenMP)

OpenMP (Open Multi-Processing) is an Application Programming Interface (API) used for shared memory multiprocessing. That means that the distinct threads that execute the instructions in parallel must share the same memory. That is true for all the contemporary Intel and AMD multicore CPUs (Fig 5.4 and Fig 5.5), so everyone using a modern computer can significantly improve the performance of a program, with a relatively small development effort. The OpenMP API works with C/C++ or fortran, and most (if not all) widely used compilers support it.

OpenMP is based on multithreading, a parallelizing method where a master thread forks a number of slave threads that concurrently run the tasks assigned to them by the system. The example of (Fig. 5.6), refers to a 4-core processor. When a task is parallelized, the master thread forks the workload to 3 slave threads and itself, entering the parallel region. When all threads have executed their instructions, a join is implemented and the program returns to single-thread mode and, later on, a new parallel region may be created. As mentioned, OpenMP can allocate tasks to multiple threads as long as they have access to the same globally shared memory. If the goal is to parallelize the code, using a distributed memory system, that is a system where various processors which have their own local memory belong to the same network, a message-passing implementation such as the Message-Passing Interface (MPI) to distribute data to the distinct processing nodes through the network must be implemented. Additionally, hybrid models can be implemented for maximum performance. In these hybrid models,we can distribute the data amongst different computers with MPI and then use OpenMP locally to take full advantage of the computational capability of each machine. This way, an even deeper level of parallelism can be reached and further improvement on code performance, can be obtained.

In general, OpenMP is a relatively easy and high-level model to implement and the initial serial code does not need to be modified in low level. The positive side of this, is that decent parallel performance can be gained for a small ‘price’, but fine optimization and linear scaling are generally hard to achieve, whereas in many cases, data and load distribution can be hidden from the programmer. Linear scaling means that when the number of available threads increases, the performance should also increase in a linear way. So when 4 threads are used in

Figure 5.4: A contemporary multicore Intel CPU.

parallel to execute a program, the execution time should be a fourth of the initial serial implementation. Generally, a code that achieves around 90% of parallel efficiency is considered to be well implemented. Data and load distribution show how the workload is distributed among the processing elements. The best scenario is, of course, to distribute the workload evenly, so for 4 threads, each one must process a fourth of the data. In C/C++, OpenMP uses *#pragma* directives to fork additional threads to work in parallel. Preprocessing directives are in general language constructs that specify how a compiler must process its input. The use of pragmas makes OpenMP an easy model to implement and it is recommended for a quick solution in improving code performance.

As mentioned, the basic advantage of OpenMP is actually its main weakness at the same time. In those high-level programming models, the biggest part of the information on how the actual tasks are going to be divided amongst the threads is hidden, and that can lead to possible errors, while setting a threshold on performance benefits. However, the OpenMP API provides several *directives* (OpenMP, 2015) that are very useful in customizing the parallelization to each specific problem's needs. The developer can include these commands as inline

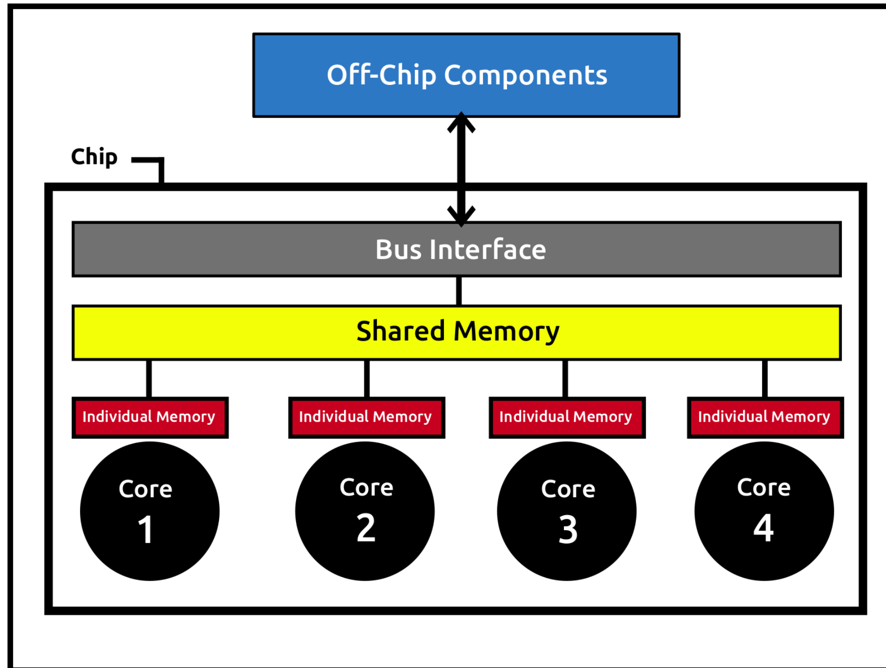


Figure 5.5: Shared memory multicore CPU architecture.

clauses when calling OpenMP in a parallel region through the pragmas.

- *parallel*: The **parallel** construct forms a team of threads and starts parallel execution. It is the most common construct to start multithreaded CPU execution.
- *single*: The **single** construct specifies that the associated structured block[1] is executed by only one of the threads in the team.
- *master*: The **master** construct specifies a structured block that is executed by the master thread. No other threads will try to execute this block.
- *critical*: The **critical** construct restricts execution of the associated structured block to a single threads at a time.
- *barrier*: The **barrier** construct sets an explicit barrier at the points at which it appears. We use this when all threads must reach the same point in the code before computations continue.

[1]A **structured block** is a single or compound statement with a single entry at the top and single exit at the bottom.

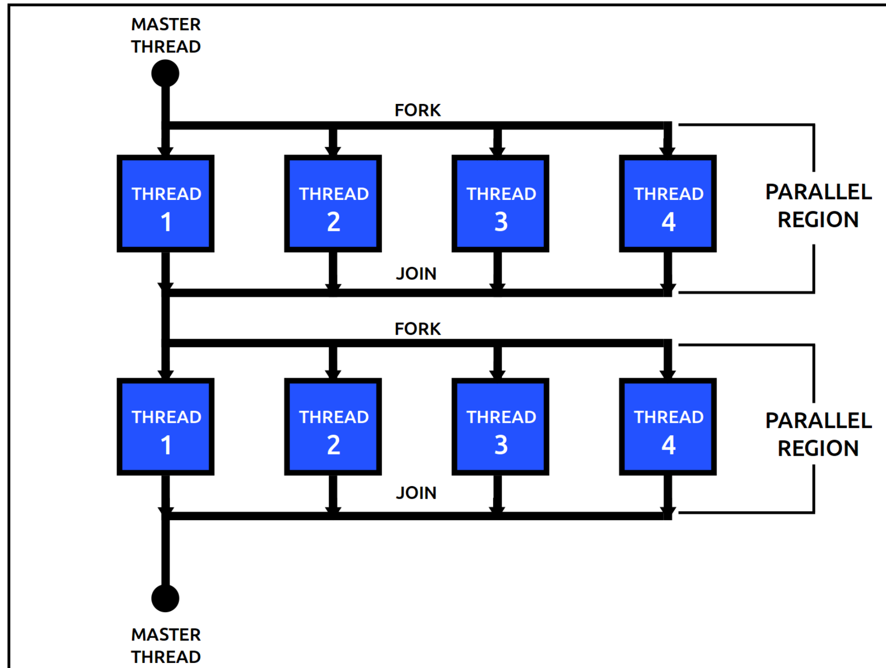


Figure 5.6: The fork-join multithreading model.

- *atomic*: The **atomic** construct ensures that a specific memory location is updated atomically, rather exposing it to the possibility of multiple simultaneous writing threads.

- *threadprivate*: The **threadprivate** directive specifies that variables are replicated, with each thread having its own copy.

These are only some of the directives one can use to tune the multithreaded parallelization based on the algorithm that is implemented. Another important group of orders are the *Data sharing attribute clauses*. These clauses apply only to variables whose names are visible in the construct on which the clause appears. The most notable include:

- $default(shared \mid none)$: Controls the default data-sharing attributes of variables that are referenced in a **parallel** or **task** construct.

- $shared(list)$: Declares one or more list items to be shared by tasks generated by a **parallel** or **task** construct.

- $private(list)$: Declares one or more list items to be private to a task.

- $reduction(operator : list)$: Declares accumulation into the list items using the indicated associative operator. Accumulation occurs into a private copy for each list item which is then combined with the original item.

In addition to the previous categories, there are the *Runtime library routines*, which affect and monitor threads, processors, and the parallel environment. The most commonly used are:

- **void omp_set_num_threads(int** num_threads**);** Affects the number of threads used for subsequent parallel regions that do not specify a **num_threads** clause. With this routine we set the number of threads that we want our program to use for a parallel region.
- **int omp_get_num_threads(void);** Returns the number of threads in the current team. It is used to monitor the number of threads that we use in the parallel regions of our code.
- **int omp_get_max_threads(void);** Returns the maximum number of threads that could be used to form a new team using a parallel **construct** without a **num_threads** clause.

All of the above information on *Directives*, *Data sharing attribute clauses* and *Runtime library routines* are found in the "Summary of OpenMP 3.0 C/C++ Syntax" which can be downloaded from *openmp.org*. The reader is encouraged to look for additional tools, in the effort to optimize custom OpenMP implementations. An example of the OpenMP algorithm for the solution of Reynolds equation is given. This example is based on the basic serial algorithm, as described before (Fig 5.3). The OpenMP variation, exposes the relative ease of development that characterizes the OpenMP model (Fig 5.7).

```
while ((difference>0.0000001) && (counter<100000))
{
      difference = 0.;
      #ifdef OPENMP_SUPPORT
      omp_set_num_threads(numberOfCores);
      #endif
      #pragma omp parallel for default(shared) private(j, resid)
      for (i = 1; i < Ddiv - 1; i++)
      {
           for (j = 1; j < Ldiv - 1; j++)
           {
                resid = ((SORomega / sorE[i*Ldiv + j])
                *(sorF[i*Ldiv + j]
                - sorA[i*Ldiv + j] * pPrevious[(i + 1)*Ldiv + j]
                - sorB[i*Ldiv + j] * p[(i - 1)*Ldiv + j]
                - sorC[i*Ldiv + j] * pPrevious[i*Ldiv + (j + 1)]
                - sorD[i*Ldiv + j] * p[i*Ldiv + (j - 1)]));
                p[i*Ldiv + j] = (1 - SORomega)*pPrevious[i*Ldiv + j] + resid;
           }
      }
      #pragma omp parallel for default(shared) private(j)
      for (i = 1; i < Ddiv - 1; i++)
      {
           for (j = 1; j < Ldiv - 1; j++)
           {
            if (p[i*Ldiv + j]<0.)
                {
                  p[i*Ldiv + j] = 0.;
                }
                if ((fabs(pPrevious[i*Ldiv + j])>DBL_EPSILON) &&
                   (fabs((pPrevious[i*Ldiv + j] - p[i*Ldiv + j])

                    / pPrevious[i*Ldiv + j]) > difference))
                {
                difference = fabs((pPrevious[i*Ldiv + j] - p[i*Ldiv + j])
                /pPrevious[i*Ldiv + j]);
                }
                pPrevious[i*Ldiv + j] = p[i*Ldiv + j];
           }
      }
}
```

Figure 5.7: OpenMP version algorithm of the default SOR iterative solver.

This code (Fig. 5.7) looks similar to the serial version (Fig 5.3) but with some very crucial modifications:

- A preprocessor flag $\#ifdef$ is inserted to define if we want to use OpenMP. When this flag is on, the system forks the parallel loop to a selected number of threads ($numberOfCores$).

- The two *#pragma* directives define the parallel regions. In this case two parallel $for\ loops$ are created.
- The SOR routine is modified. Now, the right part of the Reynolds equation (4.28) is saved in an auxiliary $resid$ value, which is private to each thread. This prevents threads from accidentally modifying another thread's value. When the fork takes place, each thread gets its own chunk of j values to process. By declaring j and $resid$ as private, we make sure that faulty local memory writes are avoided.
- A different convergence criterion is used as an example. The node with the smallest relative error value is monitored and when this satisfies the convergence criterion, the iterative process stops. In the numerical simulations a universal convergence criterion is used, for the execution times to be comparable.

A final detail is to be discussed at this point. The matrix expressions have changed from $p[i][j]$ to $p[i * Ldiv + j]$ between the serial and parallel version. This has nothing to do with the parallel OpenMP execution, but is important to mention that such a minor alteration can significantly improve the behavior of the code. It involves, how C/C++ allocates arrays in memory. When a 2D array is defined, that happens through a pointer to pointer model (*double* $^{**}a$). For a $(i \times j)$ 2-D matrix the program will allocate one 1-D array of j pointers, that will point to j blocks of i variables. This does not ensure memory locality, as the blocks may be in different areas in memory. The more sparse they are, the overhead in fetching them to cache will increase. By allocating a 1-D array and accessing the necessary values by $[i * Ldiv + j]$ we ensure that data will be placed in a consistent way in memory and the overhead will be eliminated. In certain cases an improvement of 20% in performance was noticed, by making this simple alteration for no additional development cost. Besides this, it is explained in the next section, why it is crucial for CUDA programming to have the data in 1-D array form in memory.

The last matter to analyze, is **scheduling**, a very important OpenMP concept. The scheduler, is responsible for dividing the task that is to be parallelized to the available threads. There are two types of scheduling, *static* and *dynamic*. The *static* schedule distributes the workload evenly among the threads. This means that in a for-loop that needs to execute a floating-point operation 400 times and for 4 threads available, each thread will get a chunk of 100 floating-point operations to perform. It is only logical, that if one thread is slower than the others, the execution time will depend on how long this thread will take as the others will have to wait for everyone to finish to move to the next set of instructions. In our case, all the threads must calculate the pressure values for the current iteration, or else the next iteration will not have the required data and the program will not be able to continue. Especially when a large number of threads is utilized and there are other system processes running on the same computer, the overhead is more likely to be bigger. *Dynamic* scheduling on the other hand, creates a queue of chunks that need to be processed and every

thread that finishes its task can dynamically take on the next chunk waiting in the queue. Dynamic scheduling must be used when load is distributed unevenly between the threads to exploit the threads that have a small workload and relieve those that have the heaviest. On data that display even load distribution between the chunks, dynamic scheduling can lead to a big overhead. Scheduling, is implemented in OpenMP through the clause [$schedule(type, size)$], where *type* is set to *static* or *dynamic* and *size* refers to the chunk of data that each thread will take over. Through scheduling, a finer tuning for OpenMP can be achieved that may or may not lead to improved performance or reduce unnecessary overheads.

5.4.2 Compute Unified Device Architecture (CUDA)

The second parallel computing platform tested in the context of this work, is Nvidia's Compute Unified Device Architecture (CUDA) API. CUDA is fundamentally different from OpenMP as it uses specific CUDA- enabled Graphics Processing Units (GPUs) from Nvidia to accelerate computational applications (Fig 5.8). Initially, GPUs were used to manipulate computer graphics and image processing in an efficient way but when Nvidia announced the first CUDA GPUs for general purpose computing, they became available for a whole new range of applications. General purpose GPUs have a different architecture than CPUs and it's their design that gives them an advantage when it comes to intensive computations that process huge amounts of data.

Figure 5.8: A contemporary Nvidia GPU.

The main point is that CPUs can be characterized as latency devices, while GPUs as throughput devices. Latency is the time delay between the execution of two consecutive instructions from the computer. Throughput can be seen

as the amount of instructions that a processing unit can execute in a specific time interval. That means that a CPU is designed to executed an order as fast as possible, while GPU as many orders as possible in a certain time. CPUs are expensive and sophisticated processors. On the other hand GPUs utilize a massive number of low-cost elementary threads to tolerate latencies. The architectures of the two units are presented in (Fig 5.9 and Fig 5.10). Hereinafter, some details about each architecture are highlighted:

- Central Processing Units (CPUs)
  - Powerful *Arithmetic Logic Units* (ALUs) for reduced operational latency.
  - Sophisticated *Control* units with branch prediction for reduced branch latency and data forwarding for reduced data latency.
  - Large *Caches* to convert long latency memory accesses, to short latency cache accesses.
- Graphics Processing Unis (GPUs)
  - Many energy efficient *Arithmetic Logic Units*, that display long latency but are heavily pipelined for high throughput.
  - Simple *Control* units with no branch or data forwarding.
  - Small *Caches* to boost memory throughput.

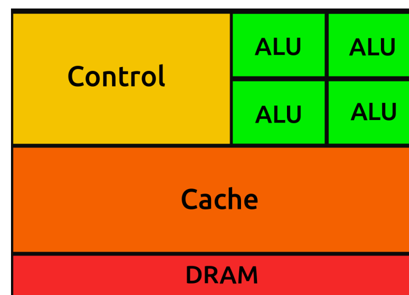


Figure 5.9: Typical CPU architecture.

In conclusion, GPU computing can make a big difference in really heavy numerical applications that require a lot of floating point operations and simple control. As it is a lot more difficult to develop GPU applications (in comparison to OpenMP) they are recommended only for the most demanding cases, when a very fine discretization is required. One advantage is that GPUs are quite inexpensive considering their computational performance.

In CUDA, the main code has a serial part that is implemented in the CPU (host) and just a segment of the program is parallelized in the GPU (device).

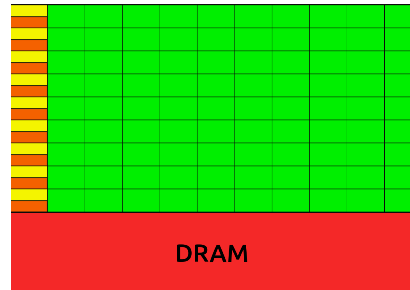


Figure 5.10: Typical GPU architecture.

The programmer must write specific device functions that are executed in the GPU to accelerate the code. These new functions are called *kernels* and are the fundamental core of a CUDA application. Generally the steps that must be followed to execute a program in the GPU are the following:

- Allocate memory on device
- Copy data from host to device
- Initialize grid and block dimensions
- Execute the kernel
- Copy data from device to host
- Free device memory

When a kernel function is called, a specific number of threads is allocated to execute it. These threads are organized in a *grid* that contains *blocks* of threads (Fig 5.11). All threads that are in a grid execute the same function. Each block in a grid is organized as a 3-D array of threads characterized by its dimensions $\{blockDim.x, blockDim.y, blockDim.z\}$. Then a grid is organized as a 2-D array of blocks $\{gridDim.x, gridDim.y\}$. Grouping blocks into grids helps avoid the limitations and apply the kernel to more threads per call. It also helps in scaling. If a GPU does not have enough resources, it will execute blocks one by one. The programmer can adjust the block and grid dimensions in order to utilize the program in the maximum capacity of the GPU and have the optimal performance. There are many instructions online on how to select the best combination of dimensions for specific hardware.

A thread is a singular processing unit in a GPU. It is based on the basic Von-Neumann processor model (Fig 5.12). In this model the processor consists of the following units:

- A *Central Processing Unit* that carries out the instructions of a computer program by performing the basic arithmetical, logical and input/output operations. It includes:

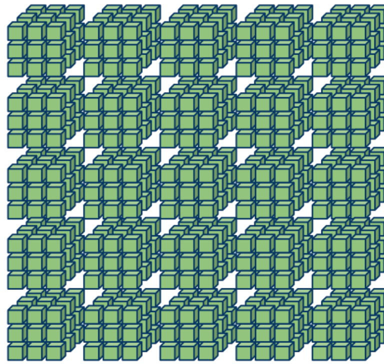


Figure 5.11: A three-dimensional grid of threads.

- An *Arithmetic Logical Unit*, the digital circuit that performs arithmetic and logical operations.
- A *Control Unit*, which controls communication between input/output devices. It reads and interprets instructions and determines the sequence for processing the data.

- A *Memory Unit*, that stores data and instructions.

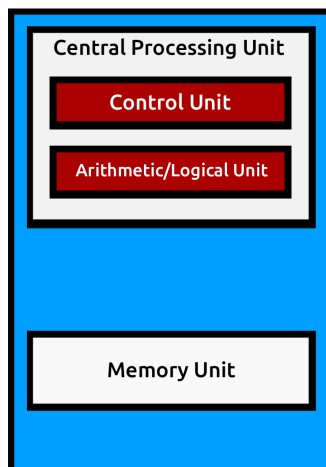


Figure 5.12: Architecture of a Von-Neumann processor.

A thread can get instructions as an input and carry out the necessary computations to produce a result as an output. The utilization of threads during runtime is based on the *streaming multiprocessors*. As the *CUDA programming guide* (NVIDIA, Corporation, 2007)states:

> "The CUDA architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors".

Based on the compute capability of the GPU, a different number of CUDA cores is included in each streaming multiprocessor. Then, at any time a single wrap of a number of threads is executed (Fig 5.13-a). It is clear from this notation that for a GPU with a certain number of CUDA cores (e.g. 384 for the basic GeForce GTX650 used in this thesis) we do not expect a linear scaling as only a predefined number of them can run at a specific point in time. A final comment on the CUDA specification can be made by discussing the *CUDA device memories* (Fig 5.13-b). There are a few different types of memories in a CUDA device. The *Global Memory* which is the largest memory and offers high bandwidth but suffers from long latencies (several hundred cycles). Then the *Local Memory* is a small volume of memory that can be accessed by only one streaming multiprocessor and is also relatively slow. The *Shared Memory* is shared between all streaming processors in a multiprocessor. It is a fast memory, just like registers. This memory provides interaction between threads, it is controlled by developers directly and features low latencies. Finally, the *Constant* and *Texture* memories are read-only and are available for all multiprocessors.

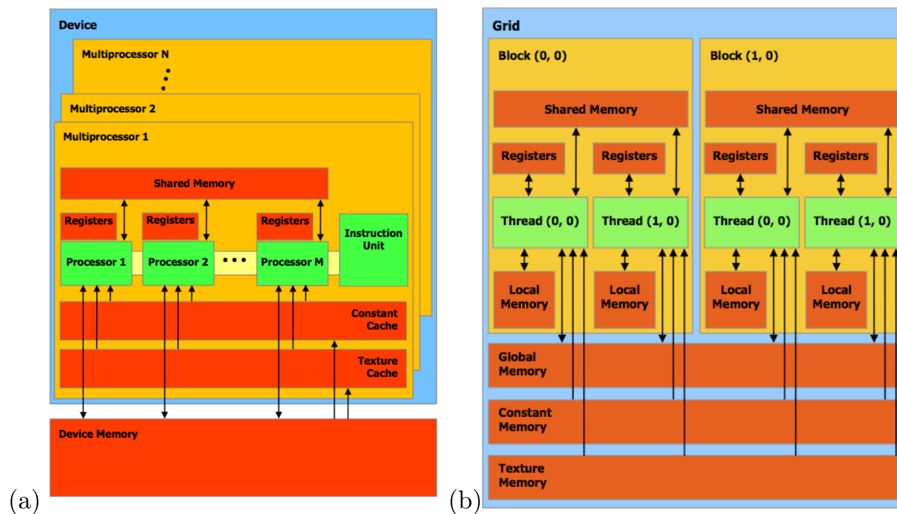


Figure 5.13: (a) The CUDA multiprocessor model and (b) the CUDA device memories.

It is already clear that CUDA programming is much more difficult and time consuming than OpenMP, where a simple preprocessing directive in an existing serial loop is sufficient to parallelize the code in the enabled threads. CUDA is a low-level programming model and the developer must manually tune the distribution of data and computational load between the computational threads. A good programming practice is to try and ensure the scalability and portability of the developed code. Scalability, means that the application should be able to run with the same performance in newer generations of hardware and scale in devices with a bigger number of cores. Portability is the usability of software in different environments and hardware architectures.

The CUDA code developed in the present work implements the Red-Black Successive-Over Relaxation (SOR) method. That is a parallel modification of the traditional SOR iterative method, suitable for this type of applications (Liu et al., 2011; Konstantinidis and Cotronis, 2012). In the Red-Black SOR method the computational grid is split in two subdomains. Half of the computational nodes (ie. those that have an even ID number) are marked as Red and the other half (ie. those that have an odd id number) are marked as Black (Fig 5.14). Here as an ID number we can assign the sum of $i + j$ for each node. In a typical differential equation such as the Reynolds equation, each node's value is updated for the current iteration through the values of his neighboring nodes (North, South, East and West). By executing a kernel that updates the Red nodes based on the Black nodes' values (left figure) and then executing a kernel that updates the Black nodes based on the Red nodes' values (right figure) in parallel, a final solution is obtained after convergence. Using this method we can compute the pressure field in a much more efficient way than by simple serial for loop.

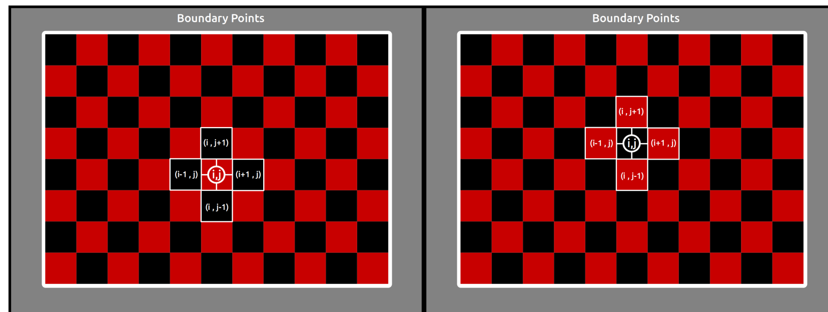


Figure 5.14: The Red-Black SOR domain decomposition.

A matter that is somewhat difficult to resolve is the convergence check. Generally, when a program is executed in the device through the kernels there is no communication between the device and the host. The only way to communicate is to copy out the appropriate variables, which is a time consuming process and must be avoided. So, to improve performance, after a certain chunk of iterations N, the pressure matrices are copied out and the convergence check is

implemented on the CPU. The number N can be seen as a different parameter and affects the execution time of the developed application.

As mentioned before, there are a few basic steps that one must follow in a CUDA program. Because the device (GPU) is different that the host machine, the algorithm is modified to include these procedures (Fig. 5.15). Firstly, one must allocate the required memory. This is done by declaring some device variables that will be exact copies of the original variables, along with their allocation size. Then, the variables are copied from device to host. After this, initialization of the block and grid dimensions and kernel execution follows. Finally, it is necessary to copy the results back to the host system and free the device memory.

This is just a sample of the complexity of CUDA programming, compared to the OpenMP API. CUDA requires a lot of low-level fine tuning but results display clear superiority when compared to OpenMP, especially considering the value-for-money factor which is increased for large grid computations.

```
#define BLOCK_SIZE_X
#define BLOCK_SIZE_Y
//kernel definitions
__global__ void redKernel
(double *sorA, double *sorB,double *sorC, double *sorD, double *sorE, double *sorF, double *p, double omega, int Ni, int Nj)
{red kernel implementation}
__global__ void blackKernel
(double *sorA, double *sorB, double *sorC, double *sorD, double *sorE, double *sorF, double *p, double omega, int Ni, int Nj)
{black kernel implementation}
//function that performs the GPU computations
void performGpuComputations()
{

      int size = Ni*Nj*sizeof(double);
      //Initialize device variables
      double* d_p;
      double* d_pPrevious;
      double* d_sorA;
      double* d_sorB;
      double* d_sorC;
      double* d_sorD;
      double* d_sorE;
      double* d_sorF;
      //gpu memory allocation
      cudaMalloc((void**)&d_sorA, size);
      cudaMalloc((void**)&d_sorB, size);
      cudaMalloc((void**)&d_sorC, size);
      cudaMalloc((void**)&d_sorD, size);
      cudaMalloc((void**)&d_sorE, size);
      cudaMalloc((void**)&d_sorF, size);
      cudaMalloc((void**)&d_p, size);
      cudaMalloc((void**)&d_pPrevious, size);
      //copy matrices on device
      cudaMemcpy(d_sorA, sorA, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_sorB, sorB, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_sorC, sorC, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_sorD, sorD, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_sorE, sorE, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_sorF, sorF, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_p, p, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_pPrevious, pPrevious, size, cudaMemcpyHostToDevice);
      //initialize grid and block dimensions
      dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y);
      dim3 dimGrid(((Ni - 1) / dimBlock.x) + 1, ((Nj - 1) / dimBlock.y) + 1);
      //kernel execution(call red-black kernel in each iteration)
      while (difference > accuracy) //convergence criterion
      {
           for (int i = 0; i < N; i++)
           {
                //Launch kernel to update red squares
                redKernel << <dimGrid, dimBlock >> >(d_sorA, d_sorB, d_sorC, d_sorD, d_sorE, d_sorF, d_p, omega, Ni, Nj);
                cudaThreadSynchronize();
                //Launch kernel to update black squares
                blackKernel << <dimGrid, dimBlock >> >(d_sorA, d_sorB, d_sorC, d_sorD, d_sorE, d_sorF, d_p, omega, Ni, Nj);
                cudaThreadSynchronize();
           }
      }
      //copy results (pressure array) back to host
      cudaMemcpy( p, d_p, size, cudaMemcpyDeviceToHost);
      //Free GPU memory
      cudaFree(d_sorA);
      cudaFree(d_sorB);
      cudaFree(d_sorC);
      cudaFree(d_sorD);
      cudaFree(d_sorE);
      cudaFree(d_sorF);
      cudaFree(d_p);

}
```

Figure 5.15: The modified CUDA algorithm of the Red-Black SOR iterative solver.

5.4.3 Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a message-passing system used on distributed memory computer systems to develop large-scale parallel applications. In message-passing, a process or function instead of being invoked by name, is send as a message and the supporting infrastructure is responsible to invoke the actual code to run. MPI is the dominant model used in high-performance computing today.

Most large-scale parallel systems, such as supercomputers are distributed memory clusters (Fig 5.16). That means that an explicit shared-memory model (such as OpenMP) cannot be applied to take advantage of their full computational capacity. MPI has an advantage over OpenMP even in shared memory systems that are based on the NUMA (Non-Uniform Memory Access) architecture. In fact, in the NUMA multicore systems that were used in this thesis, the OpenMP struggled to achieve optimal performance when a very large number of cores was used.

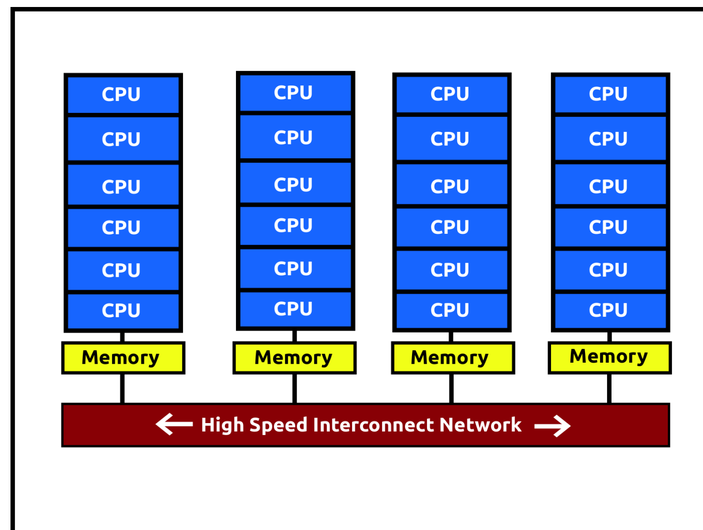


Figure 5.16: A distributed memory cluster.

In a distributed memory computer network, the individual nodes (and the CPUs they bear) cannot access the data from a shared memory as each one has its own local memory. The program must distribute the chunks of data to the different nodes through a high-speed interconnect network, and then collect the results when the scattered CPUs have finished processing them (Fig. 5.17). In this simple vector addition example we can see the following predefined MPI APIs:

- *MPI_Init*: Must be called before any other MPI routine is called and MPI can be initialized at most once.

- MPI_Comm_Rank: Gives the rank of the process in the particular communicator's group. Many programs will be written with the master-slave model, where one process (such as the rank-zero process) will play a supervisory role, and the other processes will serve as compute nodes.
- MPI_Comm_Size: Indicates the number of processes involved in a communicator.
- $MPI_Scatter$: The outcome is as if the root executed n MPI_Send operations. MPI_Send performs a standard-mode, blocking send.
- MPI_Gather: Is the inverse of $MPI_Scatter$. When called, each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order.
- $MPI_Finalize$: Cleans up all MPI states. Once this routine is called, no MPI routine (not even MPI_Init) may be called.

These and all other MPI API functions can be found at the $open - mpi.org$ (OpenMPI, 2015) documentation. A deeper insight into MPI is avoided, as it was not used in the present work.

```
int main(int argc, char **argv)
{
    int rank, size, n, i;
    double *x, *y, *buff;
    //Initialize parallel environment
    MPI_Init(&argc, &argv);
    //Get my rank (MPI ID)
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    //Find out how many procs
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // initialize buffer array to 2*n length on master
    if ( rank == 0 )
    {
        buff=calloc( 2*n , sizeof(double));
        init_vec(buff,2*n);
        print_vec(buff,2*n);
    }
    //Assume size divides n exactly
    int chunk = n / size;
    // initialize distributed vector placeholders
    x=calloc(chunk , sizeof(double));
    y=calloc(chunk , sizeof(double));
    MPI_Scatter(buff,chunk,MPI_DOUBLE,x,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
    MPI_Scatter(&buff[n],chunk,MPI_DOUBLE,y,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
    for (i=0; i<chunk; i++)
        x[i] = x[i] + y[i];
    MPI_Gather(x,chunk,MPI_DOUBLE,buff,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
    if ( rank == 0 )
        print_vec(buff,n);
    MPI_Finalize();
    return 0;
}
```

Figure 5.17: Vector addition algorithm with MPI.

5.4.4 Hybrid models

Hybrid models combine two or more parallel programming models. As an example a cluster of distributed memory nodes that contain multicore shared memory processors is examined (Fig 5.18). In this system, the same MPI example can be modified so that when the data chunks are sent to the various nodes, an OpenMP implementation can further parallelize the code to take advantage of the multicore CPUs (Fig 5.19). By adding the OpenMP directives mentioned in the previous section significant performance gains can be achieved. Here the *chunk* and *i* variables are declared as private to ensure that they have a per-

sonal value for each core and the arrays x and y as shared so that all threads can access them.

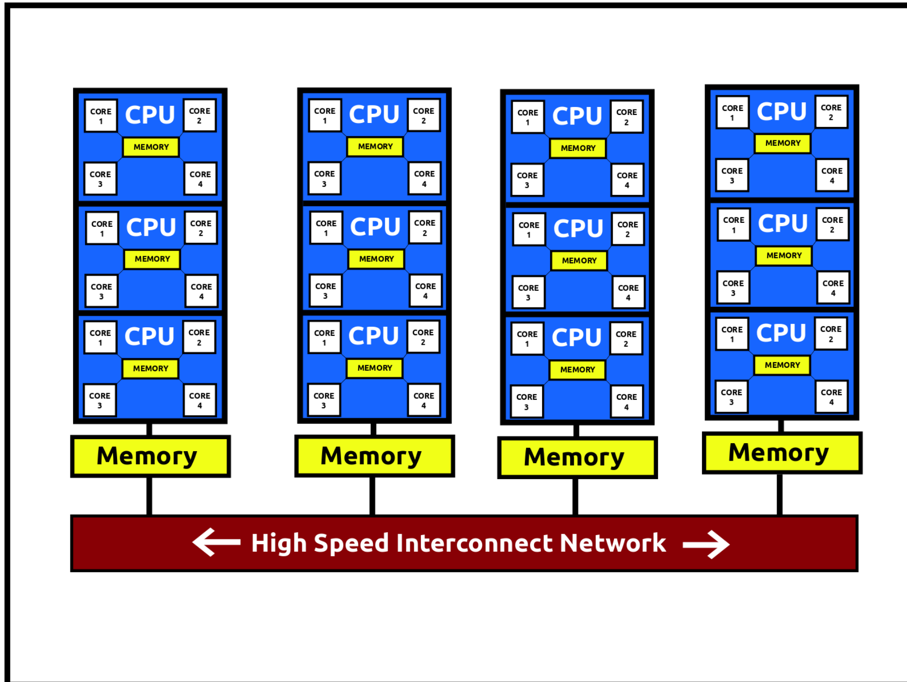


Figure 5.18: A distributed memory cluster with multicore processors.

In a similar way, for distributed nodes that have distinct Nvidia GPUs a hybrid MPI + CUDA model can be implemented, in which the MPI will pass the data between the nodes and the GPUs will be used to accelerate the large-scale computations with CUDA. These models, were not implemented in the present thesis but are briefly presented here in an effort to expose the reader to all the basic concepts regarding parallel computing.

```
int main(int argc, char **argv)
{
     int rank, size, n, i;
     double *x, *y, *buff;
     //Initialize parallel environment
     MPI_Init(&argc, &argv);
     //Get my rank (MPI ID)
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     //Find out how many procs
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     // initialize buffer array to 2*n length on master
     if ( rank == 0 )
     {
         buff=calloc( 2*n , sizeof(double));
         init_vec(buff,2*n);
         print_vec(buff,2*n);
     }
     //Assume size divides n exactly
     int chunk = n / size;
     // initialize distributed vector placeholders
     x=calloc(chunk , sizeof(double));
     y=calloc(chunk , sizeof(double));
     MPI_Scatter(buff,chunk,MPI_DOUBLE,x,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
     MPI_Scatter(&buff[n],chunk,MPI_DOUBLE,y,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
     for (i=0; i<chunk; i++)
     #pragma omp parallel for private(i, chunk) shared(x, y)
     x[i] = x[i] + y[i];
     #pragma omp barrier
     MPI_Gather(x,chunk,MPI_DOUBLE,buff,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
     if ( rank == 0 )
         print_vec(buff,n);
     MPI_Finalize();
     return 0;
}
```

Figure 5.19: Vector addition algorithm with hybrid MPI + OpenMP.

5.5 Code Analysis

The computational code of the present study was developed in two phases. The first part was developed in the context of two master thesis at the School of Naval Architecture and Marine Engineering at the National Technical University of Athens by L. Raptis and L. Koukoulopoulos (Koukoulopoulos, 2014; Raptis, 2014) and it is used for computing the response of main journal bearings of a large two-stroke marine Diesel engine, in steady-state and time-dependent

loading conditions. The developed code is capable of taking into consideration cavitation phenomena and slip boundary conditions. The code is implemented in object-oriented C++. For steady-state simulations, a constant external load is applied to the bearing. The solver, through a Newton-Raphson method, seeks proper values of the bearing eccentricity and attitude angle that will lead to a pressurized lubricant film geometry able to support the external load. In time-dependent simulations, bearing load is a function of time (or crankangle, CA). Two full engine cycles are solved, whereas 360 calculations are performed for each cycle (every 1 degree CA). Therefore, to simulate the entire problem, the code must calculate 720 separate equilibria whereas for each equilibrium Reynolds equation needs to be solved several times through the SOR method, until the Newton-Raphson method converges. For large grid sizes, these computations can be very time consuming so parallel programming becomes imperative especially when simulating cavitation phenomena or when geometrical optimization problems are solved. The mathematical analysis can be found in the third and fourth chapters of the present work but the reader may look in to the mentioned textbooks for a deeper understanding. The developed code is validated against results from various research papers; a validation section is presented in the next chapter.

When the concept of parallel programming for improving the performance of our applications was set as a subject for this thesis, a different project was set up by the writer, in order to implement the parallel algorithms for the case of a simple pad bearing. This was to confirm that these methods would give the desirable results and to start the development in a smaller scale and a simpler code. The next phase, included the embodiment of the parallel code segments in the form of independent C/C++ functions, giving users the ability to select which implementation they want to run, and compare the results between serial and parallel executions. In the OpenMP case the user is also able to select the number of threads to be activated for benchmarking the CPU or to limit overloading of the computer. A flow diagram of the developed program can give valuable information about the utilized algorithms (Fig 5.20).

The individual parts of the developed algorithm and the order that they are implemented is presented below:

- *Data Entry*: In the first part, variable initialization takes place(journal bearing dimensions, rotational speed, lubricant viscosity, vertical and horizontal loads, number of computational nodes in each dimension, number of engine cycles, number of temporal points per cycle and solver type).
- *First Time Point*: The initial assumption for the values of eccentricity and attitude angle is set and film thickness geometry is calculated.
- *Solution of Equation*: The Reynolds equation is solved for pressure field and then pressure is integrated over the lubricant domain. The hydrodynamic load components, as well as the inlet and leakage flow rates and friction force are calculated.
- *Solution Evaluator*: Examines if the equilibrium of forces is achieved.

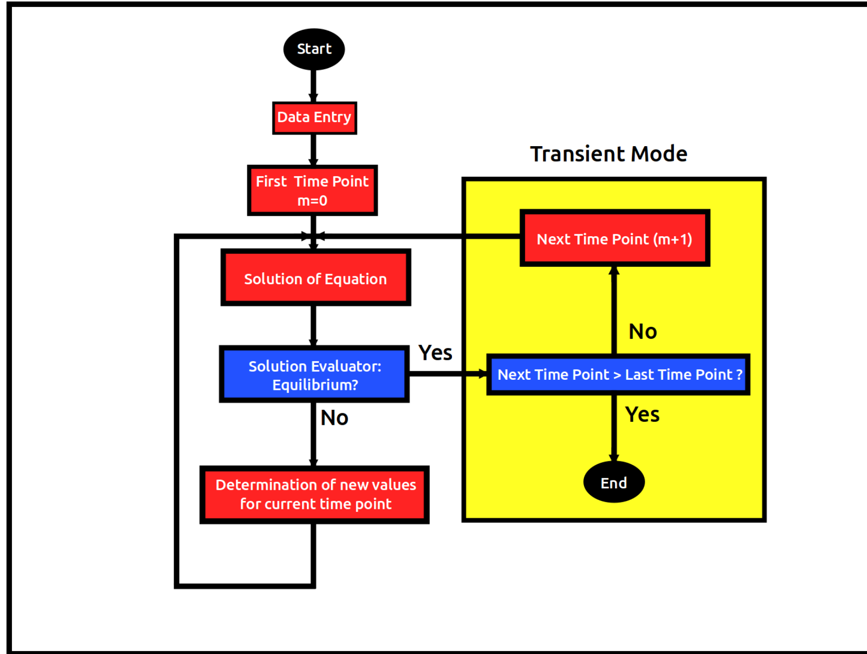


Figure 5.20: The basic flow diagram of the developed code.

- *Determination of New Values for Current Time Point*: New values for the eccentricity and attitude angle are determined through the Newton-Raphson method. Then new film thickness geometry is generated. The procedure is repeated until force equilibrium is reached.

If time-dependent mode is ON, two extra procedures follow:

- *Next Time Point* > *Last Time Point* ?: Stops the program if the net time point is after the time limit established, that is the complete engine cycles have been simulated.

- *Next Time Point*: Initializes the eccentricity and attitude angle parameters based on the two previous time steps and calculates the new film thickness geometry for the next point in time. Reynolds equation is solved, equilibrium of forces is evaluated and the procedure is repeated until force equilibrium is reached.

The developed computational code, implements these CUDA kernels in a static library form. The project architecture can give information on how the individual parts that make up the code are implemented and connected (Fig 5.21). The program is developed entirely in object-oriented C++ and has both a Win32 and a GNU/Linux version. In Linux the gcc compiler for the C++ source (.cpp) and header (.h) files was used and the nvcc compiler for the CUDA source (.cu)

and header (.cuh) files. In Win32, the project is built in Visual Studio Nsight Edition using the CUDA Software Development Kit (SDK). In the current version four different executables can be produced. The *bearingGUI* uses the Fox Toolkit to create a Graphical User Interface (GUI) to handle the input and output code operations and of course execute it. The *bearing* executable simulates the bearing's function in steady state or time-dependent mode as described before. The user can choose the parallelization method from the input file (serial execution, OpenMP with number of cores and CUDA) along with the rest of the simulation parameters. The *generateTabularValues* is used to run a parametric analysis with multiple sets of input data and the *validateAlgorithms* validates that the results between the serial and the parallel executions are the same during the development phase. All the computational functions are in the *bearingLib* static library, and call the *CUDA Lib* library which is responsible for executing the CUDA kernels. The decision to implement the GPU code in a static library has many advantages as the last version of the code is extremely portable and can be used to execute the computations in CUDA enabled machines without the need of a specific development environment or compilers. Finally, two open source libraries are used. The *Fox Toolkit* for GUI development and string input-output manipulation and the *Eigen* linear algebra library of template headers for a few matrix-vector operations.

With the same profiling procedure we determined that the only the **Solution of Equation** section needed to be parallelized and more specifically the iterative SOR method that solves the linear system and computes the pressure field.

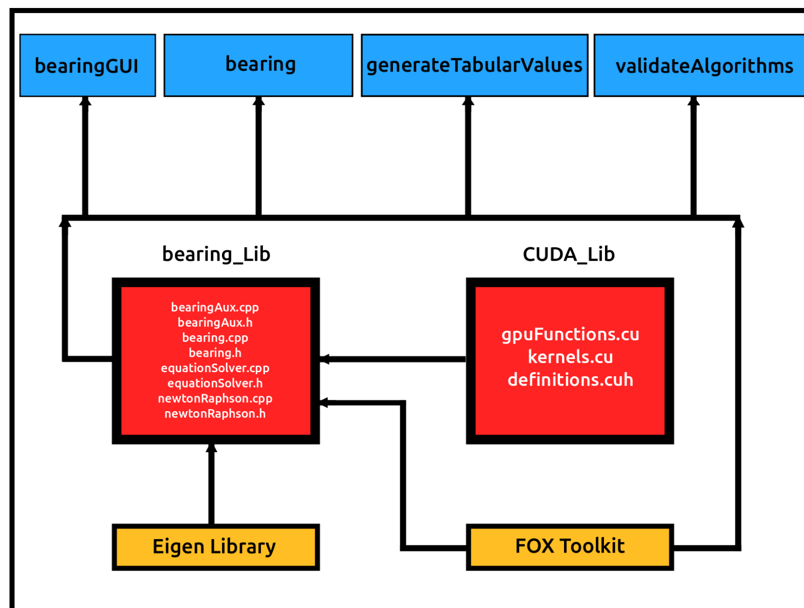


Figure 5.21: The architecture of the developed C++ project.

Chapter 6

Numerical Simulations

6.1 Introduction

In this chapter, the parallel numerical algorithms presented previously will be utilized for solving the Reynolds equation for different hydrodynamic lubrication problems. The goal of the present chapter is to demonstrate the potential increase in computational performance. The necessary code validation is presented for every examined case. That holds for the Reynolds equation, along with the modifications for hydrophobic boundary conditions and cavitation phenomena, discussed in chapter 3. Then, a benchmark case is implemented, to demonstrate how the OpenMP implementation behaves for different numbers of active threads and then compare it with the single core and CUDA implementations. Finally, a series of computational runs is performed and discussed, along with the advantages and disadvantages of each method.

The simulations of this work were run on several workstations. The corresponding hardware characteristics are presented in Table 6.1.

6.2 Validation of the developed code

The serial and parallel algorithms of the present study have been validated against published literature results. In particular, the solution algorithm of Eq. (3.17) has been validated against the results of Khonsari et al.(Khonsari and Booser, 2008), for the case of a plain journal bearing, with slenderness (length to diameter) ratio equal to 1. The comparison of results is presented in Fig. 6.1(a) where the dimensionless load capacity as a function of Sommerfeld number shows to be in very good agreement with the corresponding literature results. The solution algorithm of Eq. (3.20) for hydrophobic surfaces has been validated against the results of Guo-Jun et al. (GuoJun et al., 2007), for the case of a partially hydrophobic slider (see Fig. 6.1(b)). Finally, the solution

Central Processing Units					
ID	**Parallel Model**	**No. of Cores**	**Clockspeed**	**Memory**	**Market Value (June 2015)**
Intel i5-760	OpenMP	4	2.8 GHz	4 GB	~ $213
Intel Xeon E5-2620	OpenMP	24 (2x12)	2.1 GHz	32 GB	~ $820
AMD Opteron 6276	OpenMP	64 (4x16)	2.3 GHz	96 GB	~ $3000
Graphics Processing Units					
ID	**Parallel Model**	**CUDA Cores**	**Base Clock**	**Memory Clock**	**Market value (June 2015)**
Nvidia GeForce GTX650	CUDA	384	1058 MHz	5.0 Gbps	~ $100
Nvidia GeForce GTX980	CUDA	2048	1126 MHz	7.0 Gbps	~ $500

Table 6.1: Description of the hardware tested in the present work.

algorithm of Eq. (3.22), incorporating the Elrod-Adams cavitation model, has been validated against the results of Giacopini et al. (Giacopini et al., 2010), for the cases of a diverging-converging and a textured slider. The corresponding comparisons are presented in Fig. 6.1(c,d), also demonstrating a very good agreement.

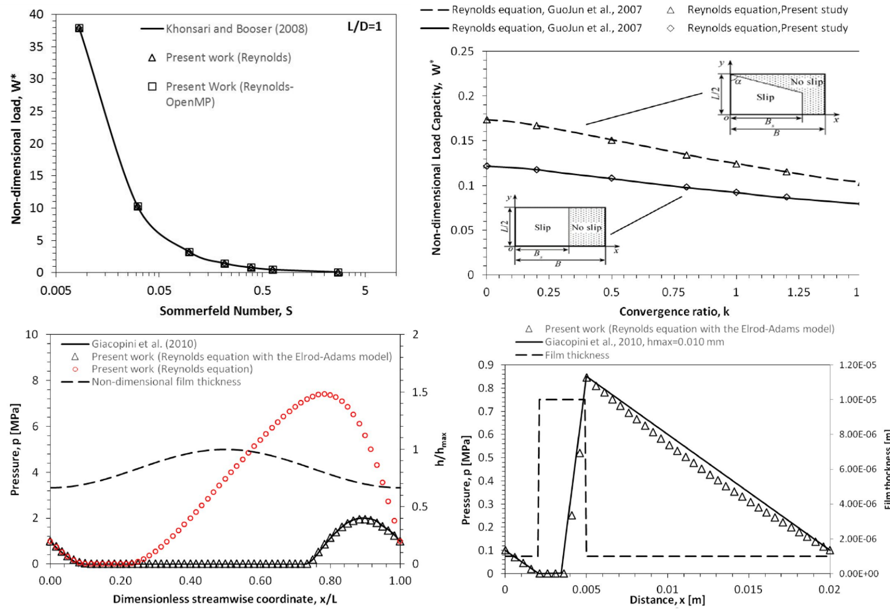


Figure 6.1: Validation of the developed algorithms for (a) a plain journal bearing, (b) a hydrophobic slider, (c) a diverging-converging slider, (d) a simple textured slider.

6.3 Benchmark case: Steady-state equilibrium of a plain journal bearing.

In the present section, the equilibrium of a plain journal bearing under steady load has been computed. The geometric and operational parameters of the bearing are presented in Table 6.2, along with the equilibrium results. Three different computational grid sizes are studied, namely those of 256x256, 512x512 and 1024x1024 nodes. Simulations have been performed utilizing single-core, OpenMP and CUDA algorithms.

OpenMP simulations have been performed on three different computer systems. The CPU characteristics of those systems have been presented in Table 6.1. In Tables 6.3 and 6.4, the execution time of the OpenMP implementations are presented. The results are also displayed in graphical form (Fig. 6.2, 6.4

and 6.6). In Figures 6.3, 6.5 and 6.7, speedup ratio S, being the ratio of total computational time of the sequential algorithm over the computation time of the parallel algorithm (quantifying relative performance improvement) for different number of utilized cores and the parallel efficiency being the percentage of the parallel execution time in respect to the serial algorithm, are presented. In all cases, the blue continuous line corresponds to the ideal linear speedup ratio and parallel efficiency. It is noted that the reference single-core simulations are performed with the Intel i5-760 processor, as it is faster than the others in sequential execution.

Diameter [m]	0.3
Length [m]	0.3
Clearance [m]	0.00005
Lubricant Viscosity [Pas]	0.026
Vertical Load [N]	1000
Horizontal Load [N]	0
Rotational Speed [rpm]	1000

Eccentricity [m]	0.627
Attitude Angle [degrees]	48.77
Minimum film thickness [μm]	18.6
Normalized friction coefficient	0.026
Maximum pressure [Pa]	1.377×10^6

Table 6.2: Steady-state equilibrium of a plain journal bearing: Geometric, operational and performance parameters.

(a)

	256x256 Grid			512x512 Grid			1024x1024 Grid		
No. of Cores	**Time [s]**	**Speedup**	**Efficiency [%]**	**Time [s]**	**Speedup**	**Efficiency [%]**	**Time [s]**	**Speedup**	**Efficiency [%]**
1	38.906	1.000	100.000	675.656	1.000	100.000	10832.900	1.000	100.000
2	20.375	1.909	95.475	365.953	1.846	92.315	5873.210	1.844	92.223
3	15.093	2.578	85.925	287.515	2.350	78.333	4672.430	2.318	77.282
4	12.041	3.231	80.778	282.859	2.389	59.717	4428.150	2.446	61.159

(b)

	256x256 Grid			512x512 Grid			1024x1024 Grid		
No. of Cores	**Time [s]**	**Speedup**	**Efficiency [%]**	**Time [s]**	**Speedup**	**Efficiency [%]**	**Time [s]**	**Speedup**	**Efficiency [%]**
1	54.268	1.000	100.000	929.397	1.000	100.000	15540.900	1.000	100.000
2	24.069	2.255	112.374	474.744	1.958	97.884	7120.610	2.183	109.126
4	12.807	4.237	105.934	207.935	4.470	111.741	3774.850	4.117	102.924
6	10.452	5.192	86.535	151.757	6.124	102.071	3064.560	5.071	84.519
8	12.353	4.393	54.914	202.760	4.584	57.297	3237.450	4.800	60.004
10	9.766	5.557	55.568	152.805	6.082	60.822	2897.160	5.364	53.642
12	8.947	6.065	50.546	139.255	6.674	55.617	3025.110	5.137	42.811
14	8.549	6.348	45.342	122.818	7.567	54.052	2934.330	5.296	37.830
16	7.055	7.692	48.076	120.711	7.699	48.121	2915.710	5.330	33.313
18	7.048	7.700	42.777	101.184	9.185	51.029	2833.220	5.485	30.474
20	6.655	8.154	40.772	91.653	10.140	50.702	2949.510	5.269	26.345
22	6.324	8.581	39.006	85.284	10.898	49.535	2904.080	5.351	24.325
24	6.064	8.949	37.288	83.685	11.106	46.275	2996.370	5.187	21.611

Table 6.3: Equilibrium of a plain journal bearing under steady load: OpenMP simulations utilizing (a) an Intel i5-760, and (b) an Intel Xeon E5-2620 system.

	256x256 Grid			512x512 Grid			1024x1024 Grid		
No. of Cores	**Time [s]**	**Speedup**	**Efficiency [%]**	**Time [s]**	**Speedup**	**Efficiency [%]**	**Time [s]**	**Speedup**	**Efficiency [%]**
1	126.894	1.000	100.000	2481.970	1.000	100.000	41259.200	1.000	100.000
4	36.791	3.449	86.226	639.327	3.882	97.054	10859.700	3.799	94.982
8	21.155	5.998	74.979	339.520	7.310	91.378	6232.980	6.619	82.744
12	17.503	7.250	60.415	238.011	10.428	86.900	5740.350	7.188	59.896
16	12.987	9.771	61.068	188.663	13.156	82.222	6288.110	6.561	41.009
20	11.439	11.093	55.466	161.868	15.333	76.666	5378.730	7.671	38.354
24	10.702	11.857	49.404	132.328	18.756	78.151	5512.440	7.485	31.186
28	9.537	13.305	47.519	114.813	21.617	77.205	5316.000	7.761	27.719
32	7.566	16.772	52.411	98.897	25.097	78.427	5035.060	8.194	25.607
36	7.252	17.498	48.605	89.427	27.754	77.095	6682.690	6.174	17.150
40	6.631	19.136	47.841	89.549	27.716	69.291	4609.230	8.951	22.379
44	7.379	17.197	39.083	89.625	27.693	62.938	4734.600	8.714	19.805
48	7.272	17.450	36.353	82.558	30.063	62.632	5622.050	7.339	15.289
52	6.876	18.455	35.490	77.336	32.093	61.718	4136.420	9.975	19.182
56	7.780	16.310	29.126	72.553	34.209	61.088	3411.450	12.094	21.507
60	7.439	17.058	28.430	67.317	36.870	61.450	4151.000	9.940	16.566
64	7.152	17.742	27.723	69.686	35.616	55.651	3321.930	12.420	19.407

Table 6.4: Equilibrium of a plain journal bearing under steady load: OpenMP simulations utilizing an AMD Opteron 6276.

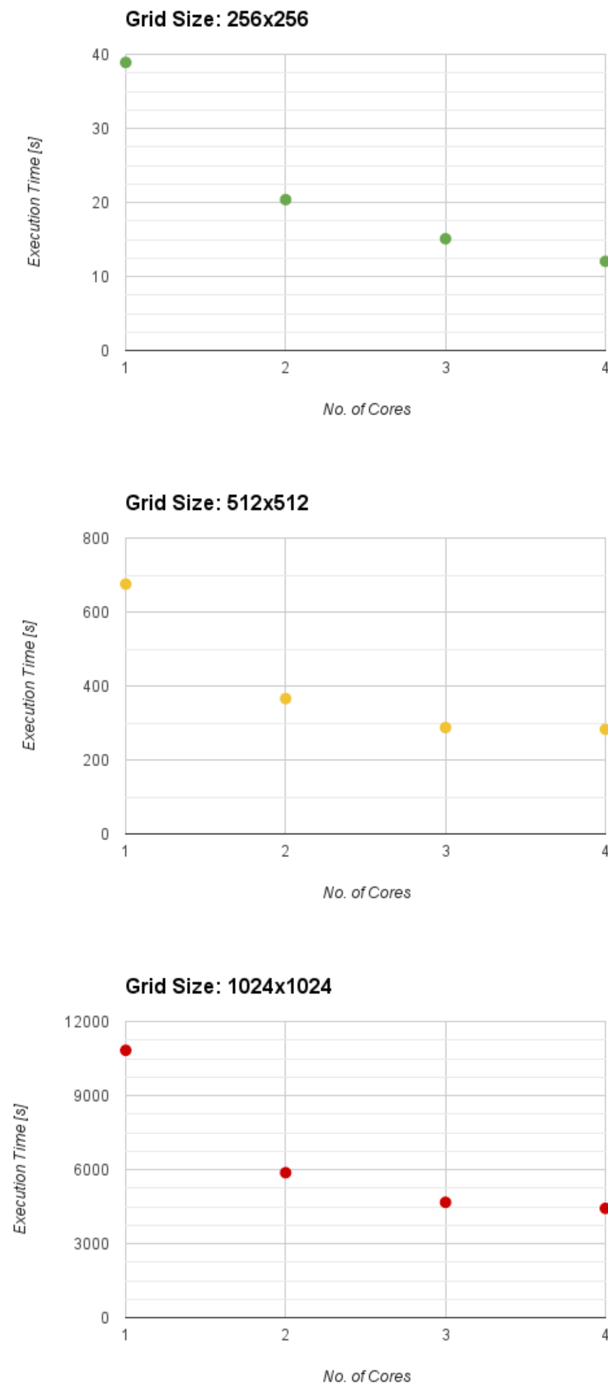


Figure 6.2: Equilibrium of a plain journal bearing under steady load: Execution times on an Intel i5-760 system for different number of utilized cores.

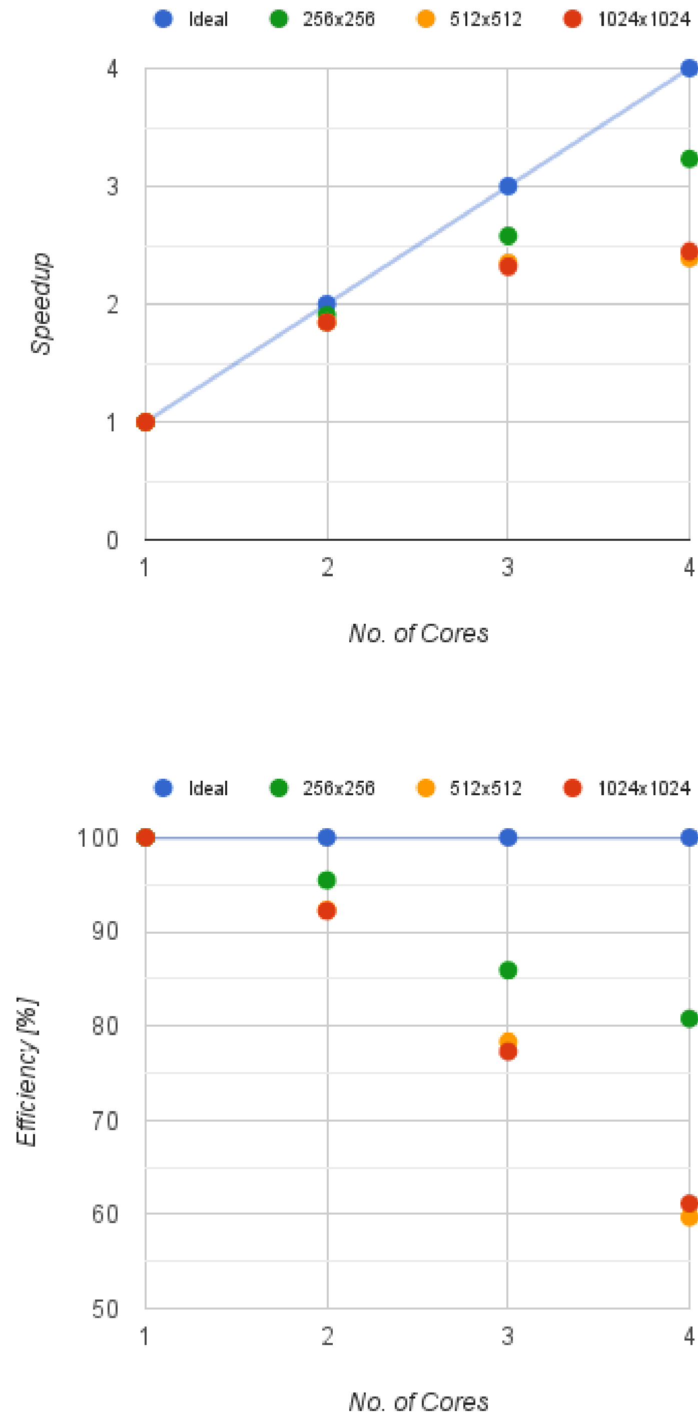


Figure 6.3: Equilibrium of a plain journal bearing under steady load: Speedup and parallel efficiency on an Intel i5-760 system for different number of utilized cores.

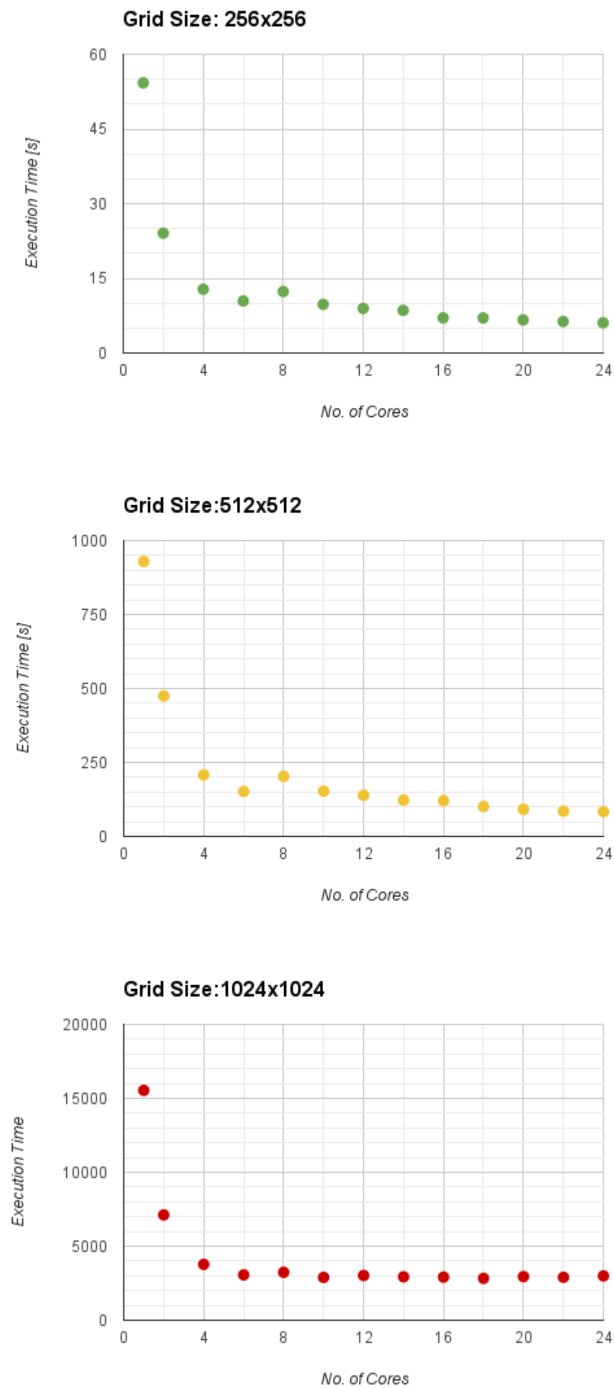


Figure 6.4: Equilibrium of a plain journal bearing under steady load: Execution times on an Intel Xeon E5-2620 system for different number of utilized cores.

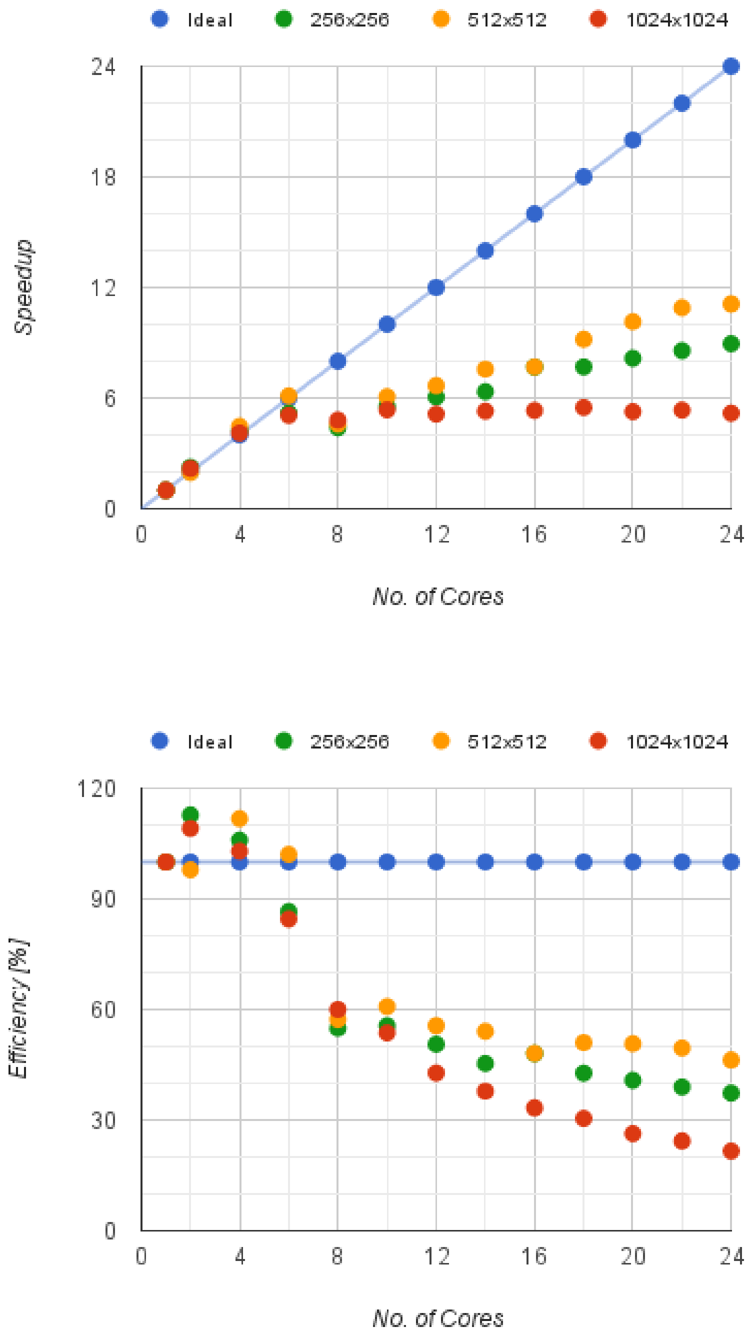


Figure 6.5: Equilibrium of a plain journal bearing under steady load: Speedup and parallel efficiency on an Intel Xeon E5-2620 system for different number of utilized cores.

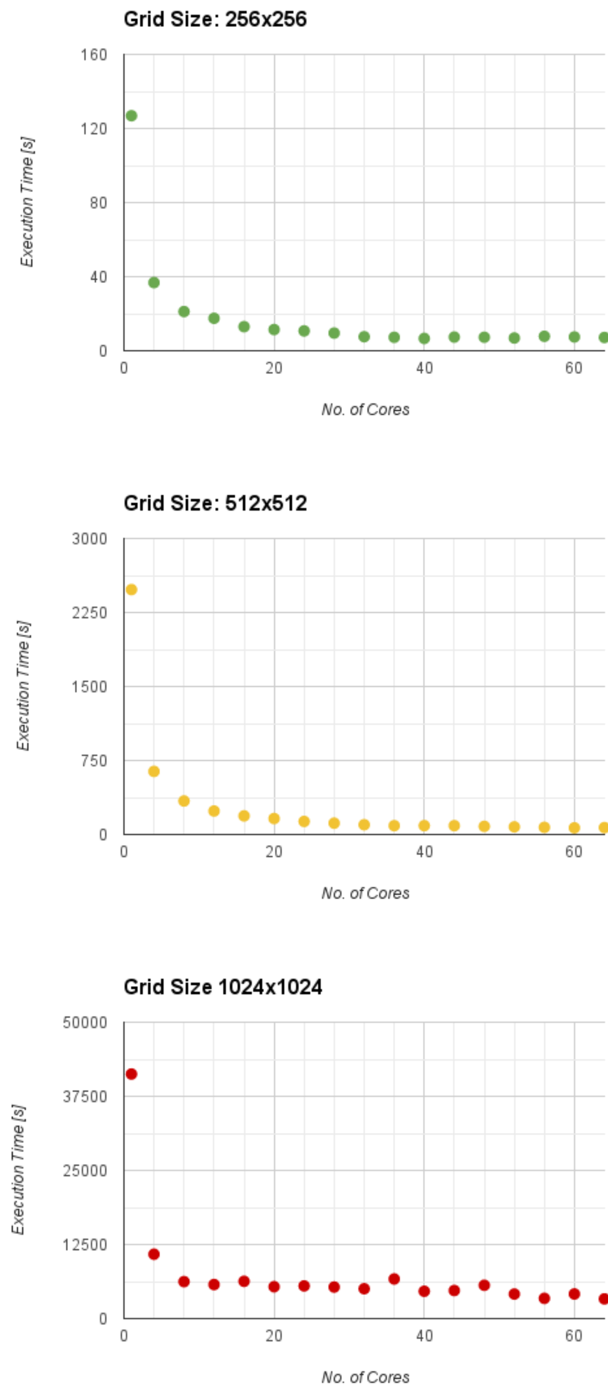


Figure 6.6: Equilibrium of a plain journal bearing under steady load: Execution times on an AMD Opteron 6276 system for different number of utilized cores.

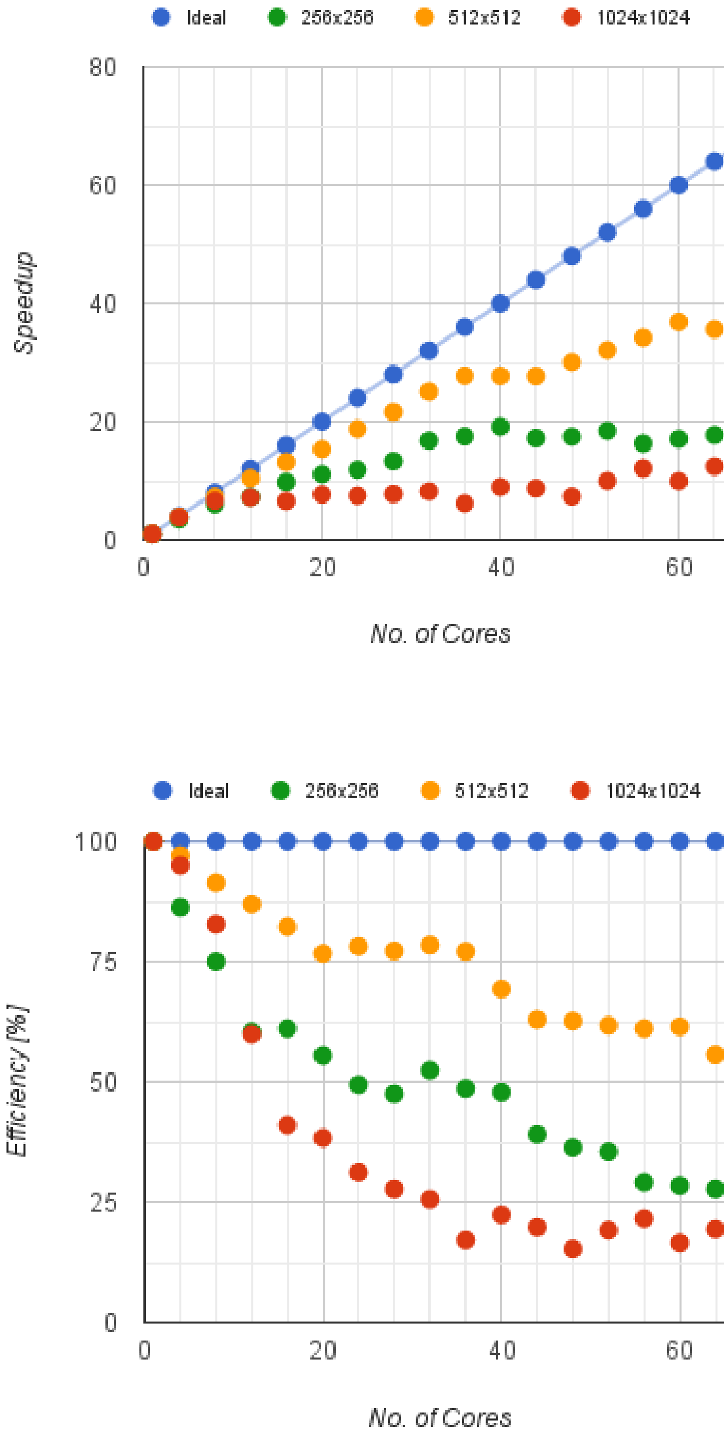


Figure 6.7: Equilibrium of a plain journal bearing under steady load: Speedup and parallel efficiency on an AMD Opteron 7276 system for different number of utilized cores.

The previous plots (Fig 6.3, Fig 6.5 and Fig 6.7) demonstrate that a linear scaling of computational performance cannot be maintained as the number of utilized cores increases. In particular, regarding the Intel i5-760 computer system, a speedup ratio of approximately 3.2 is observed for the 256x256 grid, when 4 cores are utilized. For larger grid sizes, the corresponding improvement is further reduced down to 2.4. For the Intel Xeon E5-2620 and the AMD Opteron 6276 computer systems, an almost linear increase of the speedup ratio is observed for computations up to 6 and 8 cores respectively. For larger numbers of utilized cores, speedup ratio decreases; for the 1024x1024 grid, a speedup ratio of 3.3 is achieved for utilization of 64 cores.

The observed behavior can be attributed mainly to the scheduler, a part of the system responsible for the synchronization barrier type. In more detail, when OpenMP is used to parallelize an iterative “for loop”, the workload is equally divided between the available threads. It is clear that it is not possible for all of the threads to terminate their task simultaneously, therefore some threads remain idle until the slower thread has finished processing its chunk of data. Consequently a big overhead in the performance is created, being even more intense when the number of threads increases. In general, two types of schedulers exist, the static and the dynamic. When static scheduling is used, each thread bears a predefined chunk of the total number of computations to process. In dynamic scheduling, after a thread has completed the computations it has undertaken, it takes over a chunk that is queued for processing. In the present work, the default static implementation has been used. It is further noted that, in order to accurately determine the actual processing time of the solution procedure, a “clean” workstation is necessary, meaning that no other process should be running in the system and that the thread pinning is enabled preventing thread migration. Moreover, the parallelized loop should be executed two times and only the second one to be taken into account, because, during the first time, the overhead due to the threading pool creation (slave threads) is also timed. In the present study, the performance was measured in default state workstations, with other processes also running, to replicate the average conditions of a personal computer.

Some additional remarks can be drawn for the Intel Xeon and AMD blade servers. First, when a larger number of threads is utilized, the core frequency is automatically lowered in order to prevent excessive heating, limiting computational performance. Another very important observation is related to the Non-Uniform Memory Access (NUMA) architecture of these systems (Fig 6.8). In this architecture, the processors communicate through several levels of memory with different access latencies. This architecture is called non-uniform because memory access time depends on the memory location relative to the processor. The Intel workstation is build with 2 Intel Xeon E5-2620 12-Core processors, whereas the AMD workstation with 4 AMD Opteron 6276 16-Core processors. These different processors communicate through a high latency shared memory, but each processor and each core have their own low latency cache memories. The present OpenMP benchmark provides ground for explaining the behavior of these processors. First, when a single processor is concerned we expect to

have a normal OpenMP speedup, as the NUMA architecture does not affect runtime. All variables are stored in the local memory of each processor and no overhead is caused. That is true if we activate up to 12 threads in the Intel Xeon machine and 16 threads in the AMD Opteron. But still the speedup is not linear for this amount of threads, as the results also show. That is, because these processors have 6 and 8 physical cores, respectively. In the case of the Intel Xeon processor, the extra cores are a result of Hyperthreading, whereas in the case of the AMD Opteron, the term "cores" can be misleading as in fact there are "modules" that share arithmetic logic units (ALUs) amongst them, and are capable of performing integer calculations for the operating system. As a result, with the Intel Xeon machine a good parallel efficiency is expected for up to 6 parallel threads, whereas with the AMD Opteron for up to 8 parallel threads. The above can be confirmed by the results of Fig. 6.5 and 6.7. Above these thresholds of threads, each additional thread that is enabled will contribute additional rise in performance, but not in an efficient manner. When more threads are enabled, more than one processor is needed, therefore the deficiencies of the NUMA architecture become inevitable. Data must be in a shared memory, "seen" by all the processors. However, this memory will be of very high latency and the speed at which each processor can access it, depends on their relative position. In these architectures, a thread affinity can be utilized to bind the OpenMP threads to the multiple processors. Depending on the topology of the system, thread affinity can have a dramatic impact on OpenMP performance. In the context of this work, it was not feasible to work on such levels of OpenMP optimization, but this constitutes a very interesting subject for future work.

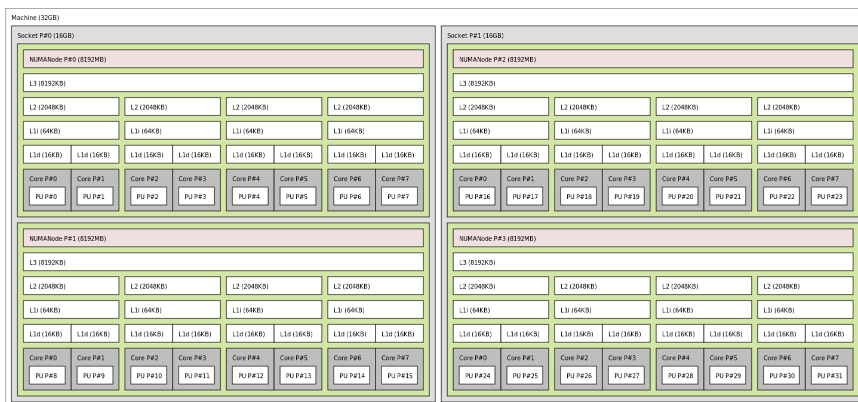


Figure 6.8: Topology of a ccNUMA Bulldozer server using hwloc.

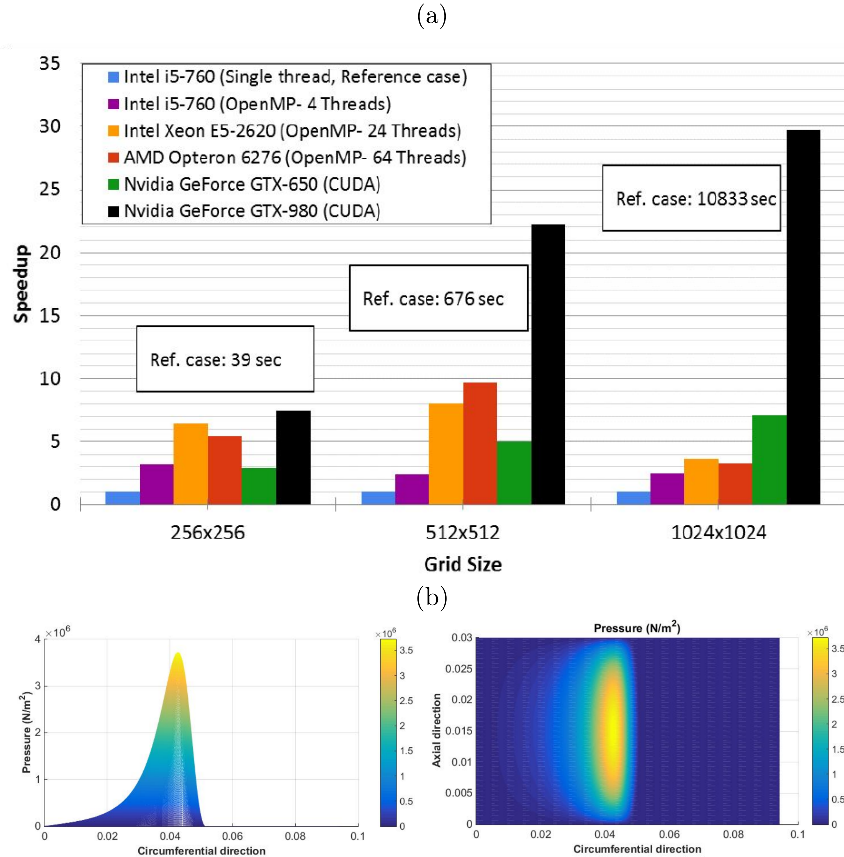


Figure 6.9: Steady state equilibrium of a plain journal bearing: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of pressure on the interior surface of the bearing.

In Figure 6.9, the performance of CUDA calculations is also presented. In the largest tested grid (1024x1024=1048576 nodes), a fairly basic GPU card (Nvidia GeForce GTX-650) surpassed the OpenMP computational systems approximately by a factor of 2. It is noted that the purchase cost Nvidia GeForce GTX-650 is 6 to 22 times less than that of the tested CPU workstations. On the other hand, the high-end GPU (Nvidia Geforce GTX-980) is dramatically faster than all other solutions. For the large computational grid, Geforce GTX-980 performs approximately 30 times faster than the best single-core CPU computation. Based on the present results, it becomes clear that for the utilized iterative methods and for large grid sizes, GPU computing is the optimum solution, in terms of both performance and acquisition cost.

6.4 Time-dependent response of a plain journal bearing

In the present section, the equilibrium of a plain journal bearing under under time-dependent, periodic load has been computed. Simulation is performed for two load cycles, considering 360 calculations per cycle. The geometric and operational parameters of the bearing are presented in Table 6.5, whereas time-dependent vertical and horizontal load of the bearing is presented in Figure 6.10(b). The bearing loads are representative of the crankshaft bearing loads of Diesel engines. Since derivative dh/dt is unknown at the beginning of simulation, two consecutive cycles need to be simulated. Here, three different computational grid sizes are studied, namely those of 256x256, 512x512 and 1024x1024 nodes. In Fig. 6.10(a), speedup ratios are summarized for the different grid sizes and computational systems. The results demonstrate that parallel computing can substantially improve the efficiency of time-dependent simulations, however, speedup is less pronounced, in comparison to the steady-state simulations of Section 6.3. OpenMP simulations on the i5-760 workstation displayed good parallel efficiency, demonstrating a speedup ratio of approximately 3 in all grid sizes. Simulations on the NUMA workstations (Intel Xeon E5-2620, AMD Opteron 6276) showed that reduction of computational time by a considerable factor is also feasible, especially in the medium sized grid. GPU simulations demonstrate improved efficiency, especially for big sized problems, reaching a speedup ratio of approximately 13, in comparison to single-core simulations.

Diameter [m]	0.3
Length [m]	0.3
Clearance [m]	0.00005
Lubricant Viscosity [Pas]	0.04
Vertical Load [N]	1000
Horizontal Load [N]	0
Rotational Speed [rpm]	1000

Table 6.5: Time-dependent response of a plain journal bearing: Geometric and operational parameters.

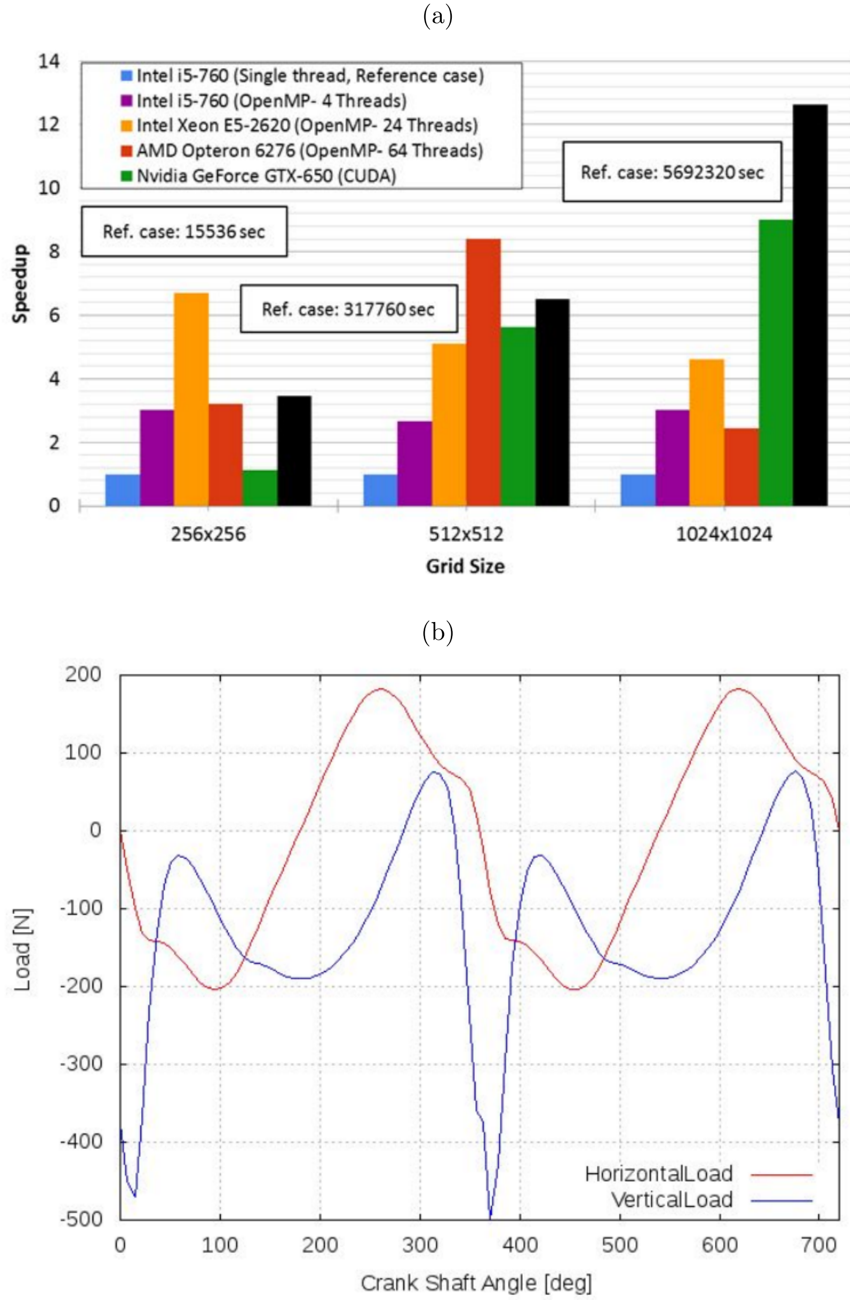


Figure 6.10: Time-dependent equilibrium of a plain journal bearing: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Plot of time-dependent vertical and horizontal load.

6.5 Steady-state response of a hydrophobic journal bearing

In the present section, the equilibrium of a plain hydrophobic journal bearing under steady load has been computed. Hydrophobicity is considered at part of the bearing surface, extending circumferentially from $\theta'_{h,start}$ to $\theta'_{h,end}$, and axially from $L_{h,start}$ to $L_{h,start}$ (see Fig. 6.11). The geometric and operational parameters of the bearing, as well as the equilibrium results, are presented in Table 6.6. Computations are performed for three different computational grid sizes, namely those of 256x256, 512x512 and 1024x1024 nodes.

Diameter [m]	0.3
Length [m]	0.3
Clearance [m]	0.00005
Lubricant Viscosity [Pas]	0.026
Vertical Load [N]	1000
Horizontal Load [N]	0
Rotational Speed [rpm]	1000

$\theta'_{h,start}$ **[degrees]**	20
$\theta'_{h,end}$ **[degrees]**	220
$L_{h,start}$ **[degrees]**	0.2L
$L_{h,end}$ **[degrees]**	0.8L
$b*$	10

Eccentricity [m]	0.495
Attitude Angle [degrees]	32.97
Minimum film thickness [μm]	25.2
Normalized friction coefficient	2.104
Maximum pressure [Pa]	1.44×10^6

Table 6.6: Steady-state equilibrium of a hydrophobic journal bearing: Geometric, operational and performance parameters.

In Fig. 6.12(a), the achieved speedup ratios for the different grid sizes and computational systems are presented. The results demonstrate that parallel computing can substantially improve the efficiency of simulations utilizing the modified Reynolds equation with hydrophobicity terms (Eq. (3.20)). OpenMP

simulations prove to be very efficient on the i5-760 workstation, displaying a speedup ratio of approximately 3, for all grid sizes. Regarding the NUMA workstations (Intel Xeon E5-2620, AMD Opteron 6276), improved performance is achieved for the small and medium sized grids, the latter displaying a speedup ratio of approximately 11. GPU simulations on the Nvidia Geforce GTX-980 demonstrate substantially improved efficiency, especially for medium and large sized problems, reaching a speedup ratio of approximately 15, in comparison to single-core simulations.

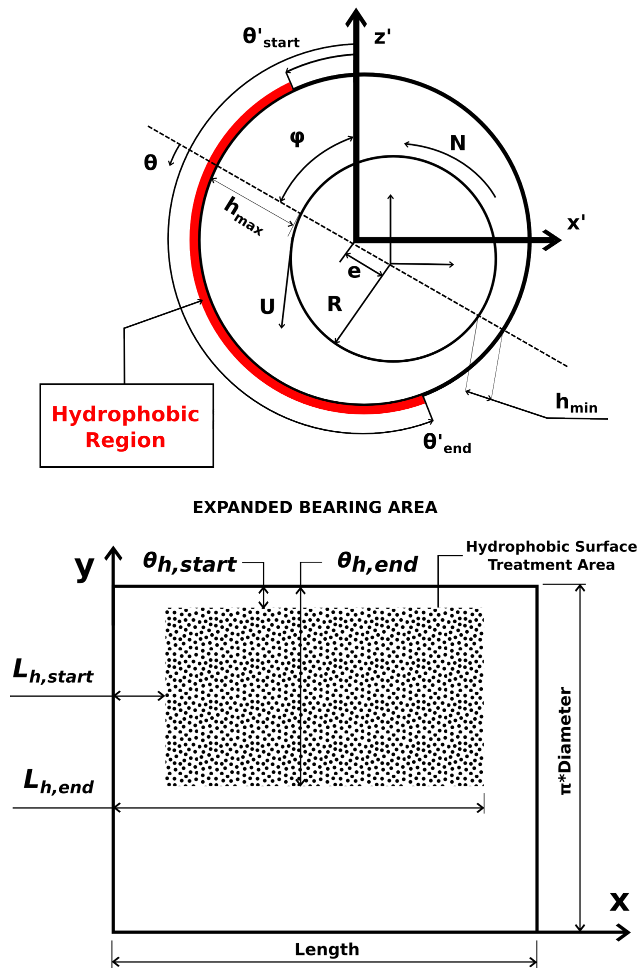


Figure 6.11: Geometry of the hydrophobic journal bearing.

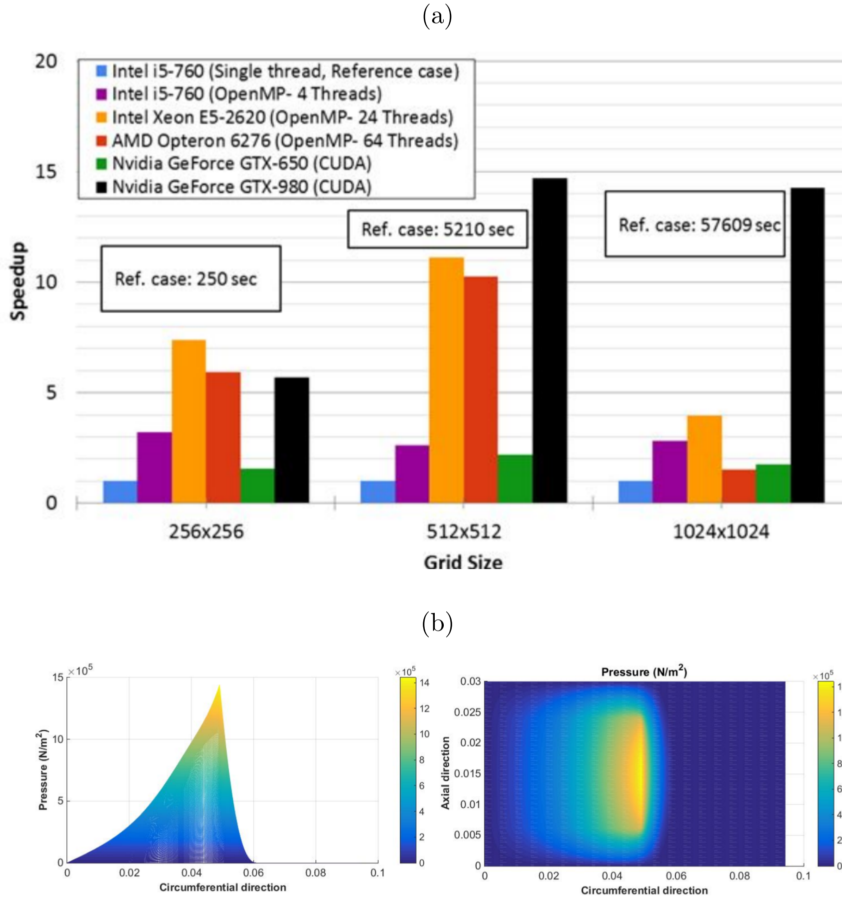


Figure 6.12: Steady state equilibrium of a hydrophobic journal bearing: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of pressure on the interior surface of the bearing.

6.6 Steady-state response of a textured journal bearing

Many recent research works have demonstrated that introduction of appropriate artificial surface texturing at part of the surface of bearings may lead to improved bearing performance. Numerical simulations of textured bearings with rectangular, cylindrical or spherical dimples require the generation of very fine meshes, therefore, this category of problems can greatly benefit from the utilization of parallel programming methods. In this section, the equilibrium of a partially

textured journal bearing under steady load has been computed. Artificial surface texturing in the form of rectangular dimples has been considered at part of the bearing surface, extending circumferentially from $\theta'_{t,start}$ to $\theta'_{t,end}$, and axially from $L_{t,start}$ to $L_{t,start}$, see Figure 6.13 and Table 6.7. The geometric and operational parameters of the bearing, as well as the main equilibrium results are also presented in Table 6.7. Computations are performed for three different computational grid sizes, namely those of 256x256, 512x512 and 1024x1024 nodes.

Diameter [m]	0.3
Length [m]	0.3
Clearance [m]	0.00005
Lubricant Viscosity [Pas]	0.026
Vertical Load [N]	1000
Horizontal Load [N]	0
Rotational Speed [rpm]	1000

$\theta'_{t,start}$ **[degrees]**	20
$\theta'_{t,end}$ **[degrees]**	200
$L_{t,start}$ **[m]**	0.15L
$L_{t,end}$ **[m]**	0.85L
$N_{d,\theta}$ **&** $N_{d,L}$	5
$d_{t,\theta}$ **&** $d_{t,L}$	0.7
h_d **[μm]**	15

Eccentricity [m]	0.625
Attitude Angle [degrees]	46.86
Minimum film thickness [μm]	18.74
Normalized friction coefficient	2.927
Maximum pressure [Pa]	1.434×10^6

Table 6.7: Steady-state equilibrium of a textured journal bearing: Geometric, operational and performance parameters.

In Fig. 6.14(a), the achieved speedup ratios for the different grid sizes and computational systems are presented. The results demonstrate that parallel computing can substantially improve the efficiency of simulations utilizing the Reynolds equation in geometries with increased complexity. OpenMP simula-

tions prove to be very efficient on the i5-760 workstation; a speedup ratio of approximately 3 is displayed for all grid sizes. Regarding the NUMA workstations (Intel Xeon E5-2620, AMD Opteron 6276), the results are similar to those corresponding to the hydrophobic bearing: Improved efficiently is achieved for the small and medium sized grids, the latter displaying a speedup ratio of approximately 9.5. GPU simulations on the Nvidia Geforce GTX-980 demonstrate substantially improved efficiency, especially for medium and large sized problems, reaching a speedup ratio of approximately 15, in comparison to single-core simulations. GPU simulations with the low-end Nvidia GTX-650 GPU display reduced performance for all the tested grid sizes.

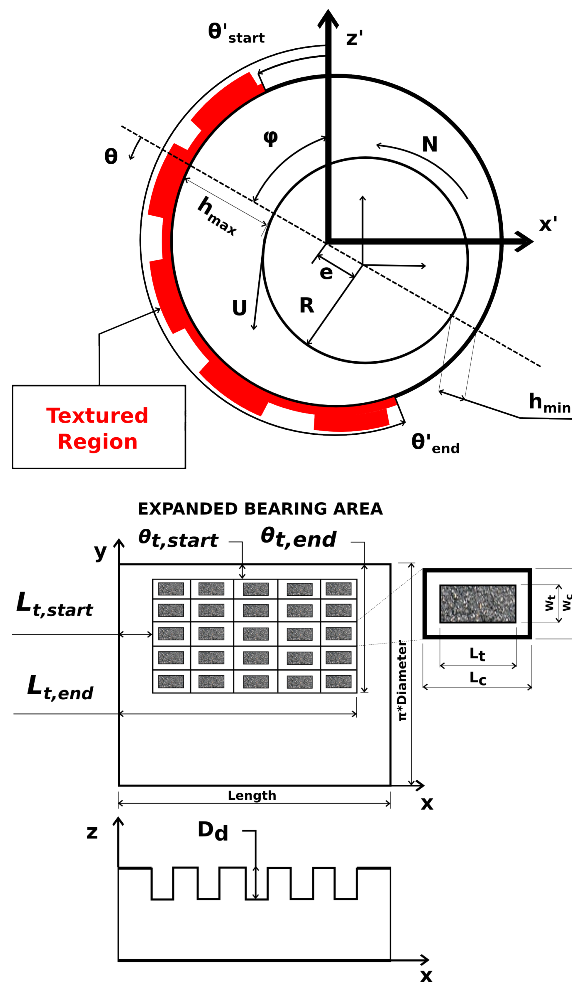


Figure 6.13: Geometry of the textured journal bearing.

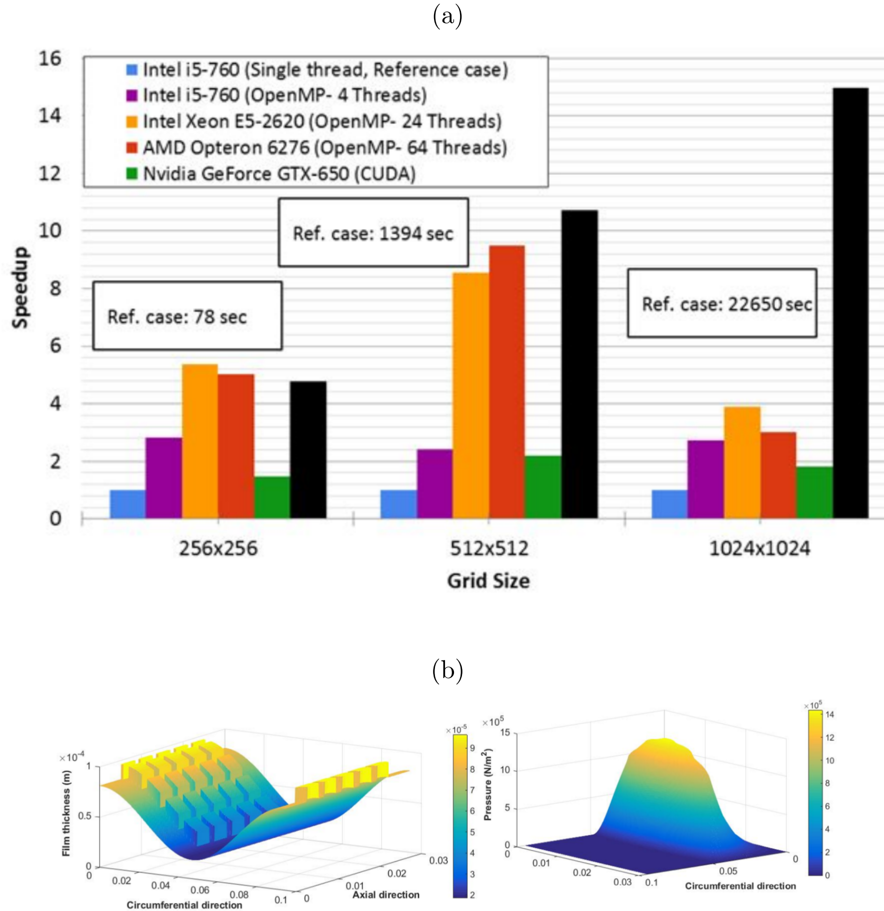


Figure 6.14: Steady state equilibrium of a textured journal bearing: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of film thickness and pressure on the interior surface of the bearing.

6.7 Steady-state equilibrium of a scratched journal bearing

Bearing performance can be substantially affected by the degradation of surface quality. Scratches constitute a usual form of bearing wear; they are observed at the lower part of the bearing bushing and their width and depth is usually of the order of a few tens of microns. Obviously, numerical simulations of scratched bearings require the generation of very fine meshes, therefore, this category of problems can also greatly benefit from utilization of parallel pro-

gramming methods. In this section, the equilibrium of a partially scratched journal bearing under steady load has been computed. The main bearing geometry and operational conditions follow the work of Dobrica and Fillon (Dobrica and Fillon, 2012), (Helene et al., 2013),see Table 6.8. Scratches in the form of circumferential grooves have been considered at part of the bearing surface, extending circumferentially from $\theta'_{s,start}$ to $\theta'_{s,end}$, and axially from $L_{s,start}$ to $L_{s,start}$, see Table 6.8 and Figure 6.15. The main equilibrium results of the bearing are summarized in Table 6.8 as well. Computations have been performed for a very fine grid of 512 x 4096 =2097152 nodes, ensuring 20 nodes in the direction of scratch width.

Diameter [m]	0.28
Length [m]	0.215
Clearance [m]	0.000224
Lubricant Viscosity [Pas]	0.011
Vertical Load [N]	210700
Horizontal Load [N]	0
Rotational Speed [rpm]	1500

$\theta'_{s,start}$ **[degrees]**	95
$\theta'_{s,end}$ **[degrees]**	265
$L_{s,start}$ **[m]**	0.15L
$L_{s,end}$ **[m]**	0.85L
$N_{s,L}$	16
h_s **[μm]**	112

Eccentricity [m]	0.967
Attitude Angle [degrees]	28.85
Minimum film thickness [μm]	7.44
Normalized friction coefficient	2.484
Maximum pressure [Pa]	20.13×10^6

Table 6.8: Steady-state equilibrium of a scratched journal bearing: Geometric, operational and performance parameters.

In Fig. 6.16(a), the achieved speedup ratios for the different computational systems are presented. The results demonstrate that parallel computing can improve computational efficiency by a factor of approximately 3. The i5-760 workstation provides the best efficiency, whereas the NUMA workstations display slightly slower execution times, although they utilize larger number of cores. GPU simulations on the Nvidia Geforce GTX-980 demonstrate substantially improved efficiency, of the order of 7.5, in comparison to single-core simulations. It is noted that simulation time can be reduced from 2 days and 1 hours (single-core solution on Intel i5-760) to roughly 8 hours (Geforce GTX-980). The low-end GTX-650 GPU was not able to substantially reduce computational time.

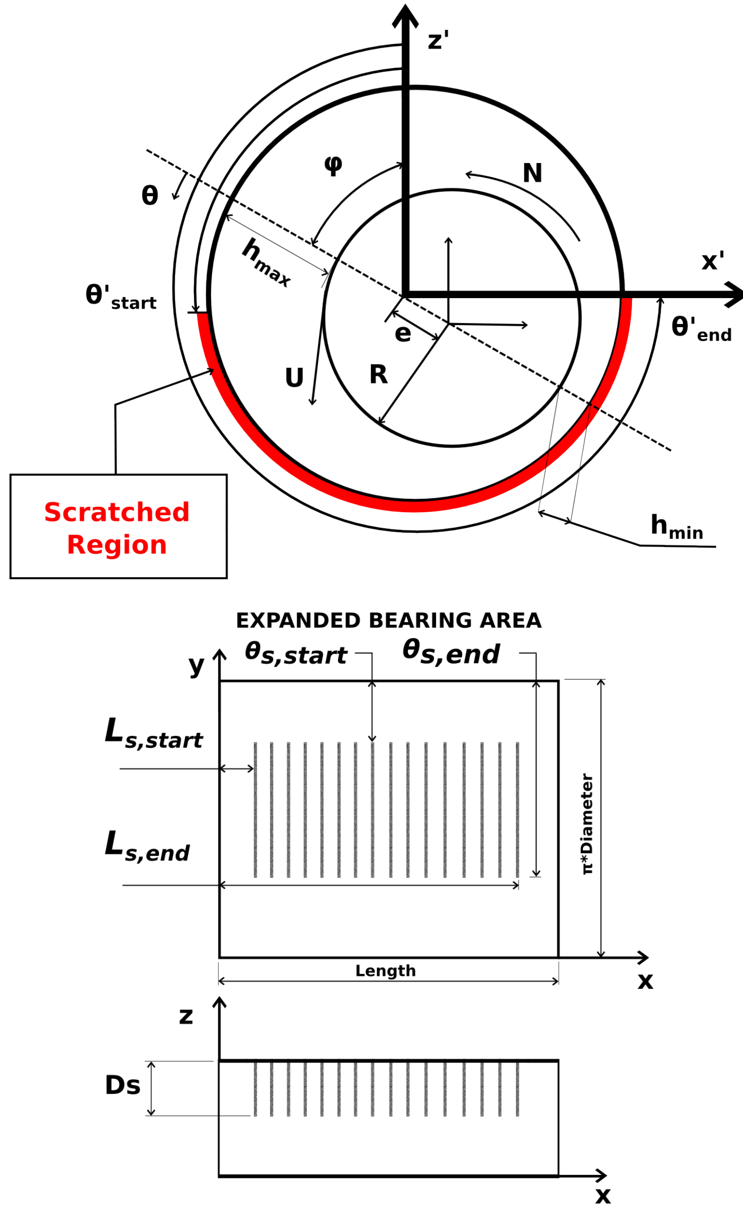


Figure 6.15: Geometry of the scratched journal bearing.

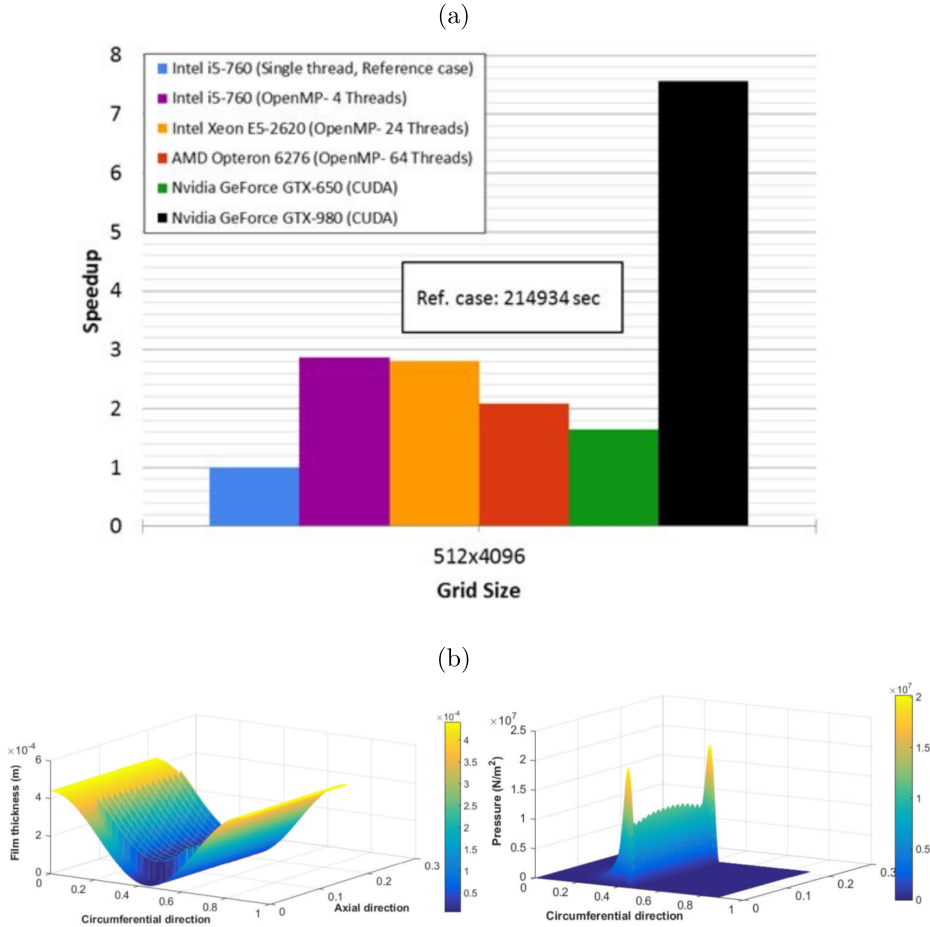


Figure 6.16: Steady state equilibrium of a scratched journal bearing: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of film thickness and pressure on the interior surface of the bearing.

6.8 Steady-state equilibrium of a plain journal bearing with oil-groove

In the present section, the equilibrium of a plain journal bearing under steady load has been computed using the Reynolds equation with the Elrod-Adams cavitation algorithm Eq. (3.22). The geometric and operational parameters of the bearing are presented in Table 6.9. Here, continuity condition in the circumferential direction is taken into account, whereas lubricating oil is assumed to enter the domain from a rectangular oil hole, extending circumferentially from

$\theta' = -10$ to $\theta' = 10$ deg and axially from $0.45L$ to $0.55L$. Oil groove pressure is assumed constant and equal to 0.2 MPa. The main equilibrium results of the bearing are summarized in Table 6.9. Two different computational grid sizes are studied, namely those of 64x64 and 128x128 nodes.

Diameter [m]	0.3
Length [m]	0.3
Clearance [m]	0.00005
Lubricant Viscosity [Pas]	0.04
Bulk Modulus [Pa]	0.0000001
Vertical Load [N]	500
Horizontal Load [N]	0
Rotational Speed [rpm]	1000

Eccentricity [m]	0.644
Attitude Angle [degrees]	47.3
Minimum film thickness [μm]	17.82
Normalized friction coefficient	2.613
Maximum pressure [Pa]	1.388×10^{6}

Table 6.9: Steady-state equilibrium of a plain journal bearing with oil groove: Geometric, operational and performance parameters.

Solution of the Reynolds equation with the Elrod-Adams constitutes the most complex and computationally intensive algorithm of the present study. The largest obstacle in parallelizing this algorithm is the domain decomposition in two different zones. The if statements separating the nodes in the cavitating region from those in the active region lead to load imbalance, as different instructions of the algorithm are executed based on the region each node belongs to. Regardless of the parallel model, the faster threads must remain idle and wait for the slower thread to finish. In Fig. 6.17(a), the achieved speedup ratios for the different grid sizes and computational systems are presented. Results show, that the OpenMP NUMA workstations were less efficient compared to the more powerful intel i5-760 processor. In the larger grid size, the GPUs dominate the OpenMP implementations, with the high-end GTX-980 being the best solution for this type of problems, reaching a speedup ratio of approximately 7.5. In Fig. 6.17(b), color-coded contours of pressure in the full-film region and fractional film content in the cavitating region of the bearing are presented.

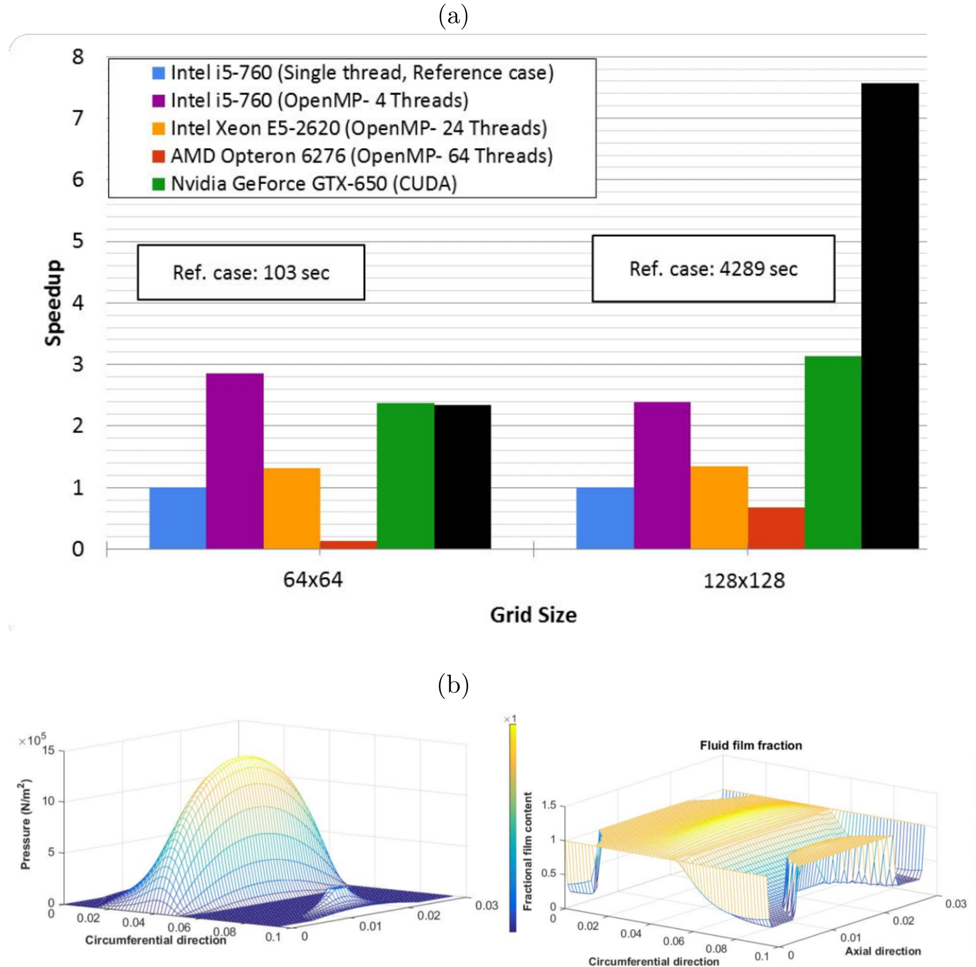


Figure 6.17: Steady state equilibrium of a plain journal bearing with oil groove: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of pressure and lubricant density on the interior surface of the bearing.

6.9 Steady-state equilibrium of a textured journal bearing with oil-groove

In the present section, the equilibrium of a textured journal bearing under steady load has been computed using the Reynolds equation with the Elrod-Adams cavitation algorithm. The bearing is equivalent to that of Section 6.8 (see Table 6.9). Artificial surface texturing in the form of rectangular dimples has been

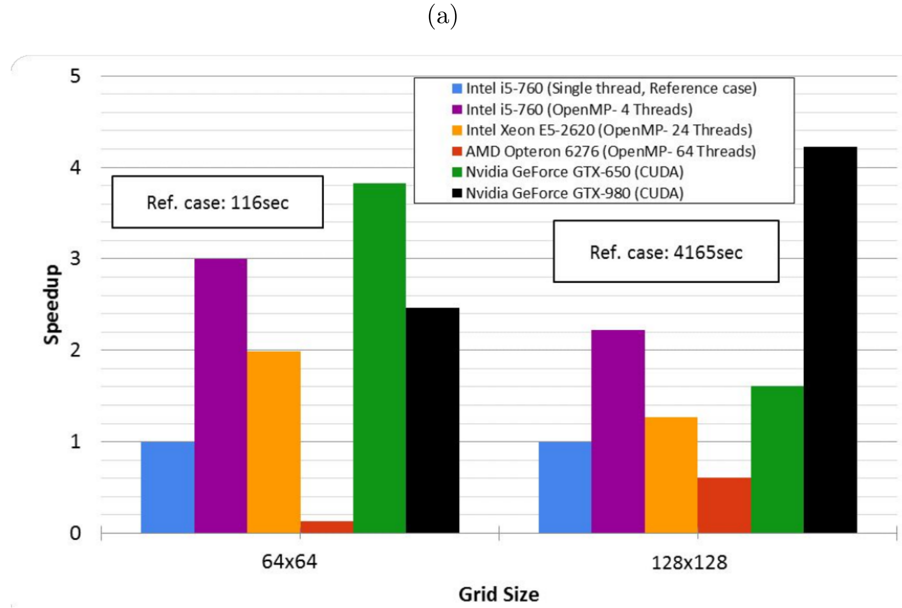
considered at part of the bearing surface. Texturing parameters are identical to those considered in Section 6.6 (see Table 6.7 for geometrical details). The main equilibrium results of the bearing are summarized in Table 6.10. Computations are performed for two different computational grid sizes, namely those of 64x64 and 128x128 nodes.

Eccentricity [m]	0.644
Attitude Angle [degrees]	40.66
Minimum film thickness [μm]	17.78
Normalized friction coefficient	2.537
Maximum pressure [Pa]	1.399×10^6

Table 6.10: Steady-state equilibrium of a textured journal bearing with oil groove: Performance parameters.

This case combines a complex algorithmic implementation and a complex bearing geometry. Parallel programming methods display the least efficiency amongst all the studied cases. Here, the i5-760 OpenMP provides acceptable efficiency, displaying a speedup ratio of approximately 3 for the small grid size, and 2.2 for the large grid size, in comparison to single-core calculations (See Fig. 6.18(a)). The efficiency of the NUMA workstations is less than single-core simulations utilizing the Intel i5-760 processor. In the small grid size, the low-end GTX-650 GPU is faster than the high-end GTX-980. The contrary is true regarding the large grid size, where GTX-980 displays a speedup ratio of approximately 4.2. For grid sizes larger than 128x128, the developed parallel algorithms were proven unstable, therefore larger grids were not not tested in the context of the present work.

(a)



(b)

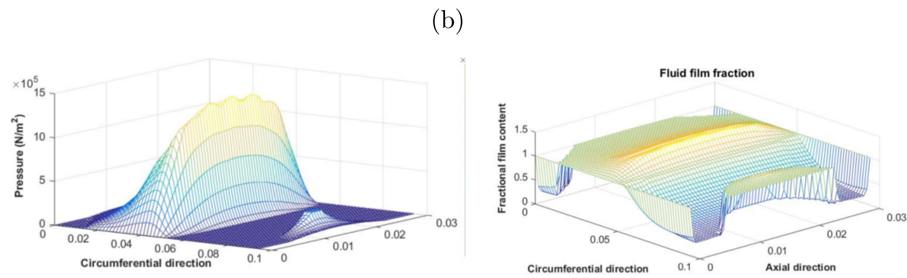


Figure 6.18: Steady state equilibrium of a textured journal bearing with oil groove: (a) Comparison of speedup ratios for the different computational systems of the present study. (b) Color-coded contours of pressure and lubricant density on the interior surface of the bearing.

Chapter 7

Conclusions and Future Work

In the present thesis, OpenMP and CUDA parallel programming algorithms have been developed for solving the Reynolds equation in two-dimensional steady and time-dependent hydrodynamic lubrication problems. Different forms of the Reynolds equation have been utilized for taking into consideration hydrophobic surfaces of the rotor and stator, and cavitation in the diverging parts of the lubricant domain. Here, plain, textured, hydrophobic and scratched journal bearings, have been considered. For each studied case, a detailed comparison of speedup amongst the different parallel algorithms and different CPU/GPU systems has been presented. Three different CPUs (Intel i5-760 with 4 cores, Intel Xeon E5-2620 with 24 cores and AMD Opteron 6276 with 64 cores) and two different GPU processing units (Nvidia GeForce GTX-650, Nvidia GeForce GTX-980) have been tested. Computational times were compared to reference single-core simulations on Intel i5-760 CPU. The results demonstrate that parallel computing can significantly reduce computational time for the considered hydrodynamic lubrication problems. Regarding CPU computations, the developed OpenMP implementation demonstrates, in most of the cases, acceptable efficiency. In particular, the Intel i5-760 achieves a mean speedup ratio of approximately 3, whereas the NUMA based Intel Xeon E5-2620 and AMD Opteron 6276 workstations, exhibit very good performance for utilization of up to 6 and 8 cores, respectively. Inconsistent performance is observed when more than one of the physical processors of the NUMA systems are used. In such cases, deeper OpenMP optimization is required to fully exploit their computational capacity. Nonetheless, in several cases, values of speedup ratio up to 11 have been recorded. The Graphics Processing Units have proven substantially more efficient, especially in large grid sizes, but require low-level tuning in the development phase. The performance of the (inexpensive) low-end GPU GTX-650 is comparable to that of the CPU workstations, exhibiting better performance on larger grid sizes. The high-end GPU GTX-980 is characterized by superior performance on large grid sizes, outperforming by a wide margin all the other tested hardware. The maximum speedup ratio achieved with GTX-980 is 29.7.

Although the results presented here have demonstrated the effectiveness of parallel computing methods in hydrodynamic lubrication problems, it could be further developed in numerous ways, such as:

- OpenMP optimization for standard and NUMA architectures
- CUDA memory optimization using texture memory and tiling
- MPI implementations for distributed memory cluster systems
- Hybrid implementations utilizing MPI+OpenMP or MPI+CUDA
- Multi-GPU implementations for several GPUs in a single workstation
- Parallel implementations of Thermohydrodynamic (THD) analysis and Elastothermohydrodynamic (ETHD) analysis of bearings using finite volumes
- Optimization studies on bearing design and operational behavior using parallel computing

Bibliography

Clausen, N. (2014). Marine diesel engines- how efficient can a two-stroke engine be?, MAN Diesel A/S Copenhagen/Denmark.

Comfort, A. (2003). An Introduction to Heavy-Duty Diesel Engine Frictional Losses And Lubricant Properties Affecting Fuel Economy- Part I, U.S Army Research, Engineering and Development Command.

Czichos, H. (1978). Tribology: a systems approach to the science and technology of friction, lubrication and wear. *Elsevier, Amsterdam*, pages 11–12.

Dobrica, M. and Fillon, M. (2012). Performance degradation in scratched journal bearings. *Tribology International*, 51.

Elrod, H. (1981). A cavitation algorithm. *ASME Journ. of Lubrication Technology*, 3:350–354.

Elrod, H. and Adams, M. (1974). A computer program for cavitation and starvation problems. *New York: Mechanical Engineering Publications*, pages 37–41.

Etsion, I., Halperin, G., Brizmer, V., and Kligerman, Y. (2004). Experimental investigation of laser surface textured parallel thrust bearings. *Tribol. Lett.*, 17:295–300.

Fatu, A., Maspeyrot, P., and Hajjam, M. (2011). Wall slip effects in (elasto)hydrodynamic journal bearings. *Tribology International*, doi:10.1016/j.triboint.2011.03.003.

Fesanghary, M. and Khonsari, M. (2011). A modification of the switch function in the elrod cavitation algorithm. *ASME Journ. of Tribology*, 133:1–4.

Giacopini, M., Fowell, M., Dini, D., and Strozzi, A. (2010). A mass conserving complementarity formulation to study lubricant films in the presence of cavitation. *ASME Journ. of Tribology*, 132:1–12.

GuoJun, M., ChengWei, and Ping, Z. (2007). Hydrodynamics of slip wedge and optimization of surface slip property. *Sci China-Physics Mech Astron*, 50, no.3:321–330.

Helene, M., Beaurain, J., Raud, X., and Fillon, M. (2013). Impact of scratches in tilting pad journal bearings- influenece of the geometrical characteristics of scratches. *Proceedings of the 12th EDF-Prime Workshop, Poitiers*.

Hori, Y. (2006). *Hydrodynamic Lubrication*. Springer.

Khonsari, M. and Booser, R. (2008). *Appied Tribology: Bearing Design and Lubrication*. John Wiley and Sons, 2nd edition.

Konstantinidis, E. and Cotronis, Y. (2012). *Accelerating the Red/Black SOR Method Using GPUs with CUDA*, volume Parallel Processing and Applied Mathematics Lecture Notes in Computer Science.

Koukoulopoulos, L. (2014). *Software Development for the Solution of Hydrodynamic Lubrication Problems in Piston Rings of Two-Stroke Marine Diesel Engines, Sch. of Naval Architecture and Marine Engineering, NTUA*. Sch. of Naval Architecture and Marine Engineering, NTUA.

Liu, J., Ma, Z., Li, S., and Zhao, Z. (2011). A gpu accelerated red-black sor algorithm for computational fluid dynamics problems. *Advanced Materials Research Vol. 320 (2011) pp 335-340*.

NVIDIA, Corporation (2007). *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*.

OpenMP (2015). Syntax and Documentation. http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf. Accessed: 07/2015.

OpenMPI (2015). Documentation. http://www.open-mpi.org/doc. Accessed: 07/2015.

Papadopoulos, C., Nikolakopoulos, P., and Kaiktsis, L. (2011). Evolutionary optimization of micro-thrust bearings with periodic trapezoidal surface texturing. *J. Eng, Gas Turbines Power*, 133:1–10.

Raptis, L. (2014). *Software Development for the Solution of Hydrodynamic Lubrication Problems in Main Bearings of Marine Diesel Engines, Sch. of Naval Architecture and Marine Engineering, NTUA*. Sch. of Naval Architecture and Marine Engineering, NTUA.

Stachowiak, G. and Batchelor, A. (2001). *Hydrodynamic Lubrication*. Butterworth Heinemann.

Vijayaraghavan, D. and Keith, T. (1989). Development and evaluation of a cavitation algorithm. *STLE Tribology Transactions*, 32:225–233.

Volund, A. (2003). Measurement and calculation of frictional loss in large two-stroke engines, Department of Mechanical Engineering, Technical University of Denmark, DTU-Tryk.

Wang, N., Chan, C., and Cha, K. (2012). Workstation computing of discretized reynolds equations. *Tribology Transactions*, 55, Issue 3, 2012:288–296.

Wang, N., Chih-Ming, T., and Kuo-Chinag, C. (2009). Optimum design of externally pressurized air-bearing using cluster openmp. *Tribology International*, 42:1180–1186.