Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Optimization Techniques for Task-based Parallel Programming Models

Διπλωματική Εργασία

του

Αθανάσιου – Άκανθου Χασάπη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εργαστήριο Υπολογιστικών Συστημάτων

Αθήνα, Ιούλιος 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Optimization Techniques for Task-based Parallel Programming Models

Διπλωματική Εργασία

του

Αθανάσιου – Άκανθου Χασάπη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Ιουλίου, 2014.

.....	.....	.....
Νεκτάριος Κοζύρης	Αριστείδης Παγουρτζής	Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.	Επικ. Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2014

.....
Αθανάσιος – Άκανθος Χασάπης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © – All rights reserved Αθανάσιος – Άκανθος Χασάπης,
2014. Με επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Ένα από τα πιο απαιτητικά προβλήματα στα σύγχρονα παράλληλα υπολογιστικά συστήματα είναι η εκμετάλλευση του μεγάλου αριθμού των νημάτων/πυρήνων που προσφέρει το σύγχρονο υλικό, με σκοπό την βελτίωση της αποδοτικότητας εφαρμογών που εκτελούν κομμάτια κώδικα παράλληλα. Στην βιβλιογραφία και την βιομηχανία έχουν προταθεί διάφορα προγραμματιστικά μοντέλα για αυτό τον σκοπό, στα οποία περιλαμβάνεται και το μοντέλο με παράλληλες εργασίες. Στο συγκεκριμένο μοντέλο, που έχει σκοπό την απλοποίηση του παράλληλου προγραμματισμού, ο προγραμματιστής εκφράζει τον παραλληλισμό της εφαρμογής ως εργασίες που μπορούν να εκτελεστούν παράλληλα και το σύστημα εκτέλεσης αποφασίζει πως αυτές οι εργασίες θα ανατεθούν σε νήματα του λειτουργικού συστήματος προς εκτέλεση.

Στόχος της παρούσας εργασίας είναι να εξερευνήσει και να βελτιστοποιήσει τους εσωτερικούς μηχανισμούς της βιβλιοθήκης Intel TBB κάτω από συγκεκριμένους αρχιτεκτονικούς περιορισμούς. Αρχικά εξετάζουμε τον scheduler εργασιών της βιβλιοθήκης, με έμφαση στον μηχανισμό «κλοπής εργασιών», ώστε να αναγνωριστούν οι βασικές λειτουργίες του και εκτελούμε profiling για να μετρήσουμε την επιβάρυνση που επιφέρει η καθεμία. Εν συνεχεία, γίνεται προσπάθεια να βελτιστοποιήσουμε τον μηχανισμό τυχαίας κλοπής προσθέτοντας πληροφορίες που αφορούν την αρχιτεκτονική, κυρίως την ιεραρχία κρυφών μνημών και την διαμόρφωση των packages. Υλοποιούμε έναν μηχανισμό κλοπής εργασιών που ακολουθεί δύο πολιτικές: 1) κλοπή από τους κοντινότερους πυρήνες (σε απόσταση ιεραρχίας μνήμης), 2) κλοπή από τον πιο φορτωμένο με εργασίες πυρήνα. Η πρώτη πολιτική έχει στόχο να μεγιστοποιήσει την επαναχρησιμοποίηση δεδομένων που μοιράζονται πυρήνες στην ιεραρχία μνήμης, μείωση της μόλυνσης της κρυφής μνήμης με μη σχετικά δεδομένα (μείωση των conflict/coherence misses), ενθαρρύνοντας την πρόσβαση δεδομένων σε τοπικό αρχιτεκτονικό επίπεδο. Η δεύτερη πολιτική έχει στόχο την βελτίωση της εξισορρόπησης φορτίου μεταξύ των πυρήνων. Για την αξιολόγηση των παραπάνω παρουσιάζουμε πειραματικά αποτελέσματα που αφορούν την βελτίωση της απόδοσης διάφορων εφαρμογών σε μία SMP πλατφόρμα 24 πυρήνων, μία NUMA πλατφόρμα 12 πυρήνων και μία NUMA πλατφόρμα 32 πυρήνων (με πολυνηματισμό).

Λέξεις-Κλειδιά: Intel TBB, παράλληλα προγραμματιστικά μοντέλα βασισμένα σε εργασίες, εξισορρόπηση φορτίου, ιεραρχία κρυφών μνημών, κλοπή εργασιών, τοπικότητα δεδομένων

Abstract

One of the most challenging problems in modern parallel processing systems is to exploit the large number of cores/threads available in modern hardware, in order to improve the efficiency of applications by executing pieces of code in parallel. Various programming models have been proposed for this purpose, among which the task programming model. This model aims at simplifying parallel programming. In this model, the programmer expresses parallelism as tasks to be executed in parallel and the runtime system decides how these tasks are assigned to system threads.

The goal of this thesis is to explore and optimize the internals of the Intel TBB Library under certain architectural conditions. Initially we examine the library task scheduler, focusing on the task stealing mechanism, in order to identify its basic functions and we run some profiling to verify the task stealing functionality and to measure the overheads of each basic function. Subsequently we attempt to optimize the architecture agnostic random stealing function by adding architecture information, mainly about the cache hierarchy and the socket configuration. We implement a stealing mechanism that adopts certain policies: i) stealing from the closest (in terms of cache/NUMA locality) core, ii) stealing from the most loaded core. The first policy aims to maximize the reuse of data shared between cores, reduce cache pollution due to irrelevant data (i.e. minimize conflict/coherence misses), and promote data accesses from local NUMA memory nodes. The second policy tries to achieve better load balancing among the cores. To that end, we present experimental results on performance improvement by measuring the speedup of several applications on a 24-core SMP and a 12-core (with hyperthreading) NUMA multicore machine.

Keywords: Intel TBB, task-based parallel programming models, load balancing, cache hierarchy, work stealing, data locality

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του Καθηγητή Νεκτάριου Κοζύρη.

Θα ήθελα καταρχήν να ευχαριστήσω τον καθηγητή μου κ. Νεκτάριο Κοζύρη για τη επόπτεία του κατά την εκπόνηση της εργασίας μου, για τις γνώσεις και την έμπνευση που μου προσέφερε με την διδασκαλία του, αλλά και για την ευκαιρία που μου έδωσε να ασχοληθώ με ένα θέμα εξαιρετικά ενδιαφέρον στο άριστο περιβάλλον του εργαστηρίου του.

Θα ήθελα να εκφράσω την ιδιαίτερη ευγνωμοσύνη μου στον μεταδιδακτορικό ερευνητή κ. Νίκο Αναστόπουλο για την συνεχή του καθοδήγηση στην διάρκεια εκπόνησης της διπλωματικής αυτής εργασίας, ο οποίος με την ανεξάντλητη υπομονή του με ενθάρρυνε συνεχώς και με γέμιζε αισιοδοξία για την πορεία της εργασίας, καθώς και για τις γνώσεις και την εμπειρία που μου μεταλαμπάδευσε μέσω των συζητήσεών μας γύρω από το θέμα της εργασίας.

Θα ήθελα ακόμη να ευχαριστήσω τους φίλους και συναδέλφους μου για την στήριξη που μου προσέφεραν και τις εμπειρίες που μοιραστήκαμε.

Τέλος, θα ήθελα να ευχαριστήσω την σύντροφό μου, η οποία με στήριζε ψυχολογικά σε όλη την ακαδημαϊκή μου πορεία μέχρι τώρα, δείχνοντας εμπιστοσύνη στις επιλογές μου και στις δυνάμεις μου, και την οικογένειά μου, η οποία με στήριξε με υπομονή στην πορεία μου στο ΕΜΠ τόσα χρόνια, αποτελώντας την σταθερότερη αξία στην ζωή μου.

Table of Contents

Περίληψη 5
Abstract 6
Ευχαριστίες 7
Table of Contents 10
List of Figures 13
List of Listings 20
Chapter 1 Introduction 21
1.1 Overview 21
1.2 Multi-socket, multi-processor systems 22
1.2.1 Shared Memory Architectures 22
1.2.2 Distributed Memory Architectures 23
1.2.3 Hybrid Architectures 24
1.2.4 A Multi-Socket Multi-Core Machine 25
1.3 Parallel Programming, Amdahl's Law, scalability 25
1.4 Parallel Programming Models 27
1.4.1 Shared Memory programming model 27
1.4.2 Distributed Memory programming model 28
1.4.3 Hybrid Programming model 29
1.5 Overview of Key Features for Performance 30
1.6 Problems and pitfalls of parallel programming that should be avoided 31
1.7 Desired Properties of Parallel Programming Models 33
Chapter 2 Motivation – Overview of the Problem 35
2.1 Regular vs. Irregular &Nested/Recursive Parallelism – Oversubscription - Implicit vs. Explicit Parallelism 35
2.2 The TBB Library 37
2.2.1 Overview of the library 37
2.2.2 How it satisfies the desired properties 39
2.3 TBB Scheduler 39
2.3.1 Overview, Basic Architectures and Components, Basic Functionalities 39

2.3.2 Executing Tasks - Work Stealing Mechanism & Load Balancing Algorithms 40
2.3.3 Cache Coherence Protocols and Problems with Work Stealing 44
2.4 Profiling of basic functionalities – Characterization of overhead scalability 44
2.4.1 Systems Used 45
2.4.2 TBB Scheduler Basics 47
2.4.3 Basic TBB Functionalities 47
2.4.4 Applications used for characterization 48
Chapter 3 Techniques Used 63
3.1 Optimization targets 63
3.2 Stealing from the nearest neighbor 63
3.2.1 Technique description 63
3.2.2 Implementation details 64
3.3 Stealing from the most loaded processor 67
3.3.1 Technique description 67
3.3.2 Implementation details 68
Chapter 4 Evaluation 71
4.1 Physical Systems 71
4.2 Stealing from the nearest neighbor 72
4.2.1 Benchmarks Used 72
4.2.2 Results 72
4.2.3 Remarks 77
4.3 Load Balancing 78
4.3.1 Benchmarks Used 78
4.3.2 Results – Finding Max 78
4.3.3 Results–Global vs Local Sorted List 82
Chapter 5 Epilogue – Conclusions & Future Work 87
5.1 Conclusions 87
5.2 Related Work 87
5.3 Future Work 88
Chapter 6 Bibliography - References 89
Chapter 7 Appendix A – Profiling Results 90
7.1 Quicksort 90

7.2 Swaptions 93
7.3 Matrix Multiplication 96
7.4 Convex-hull 99
Chapter 8 Appendix B – Evaluation Results 103
8.1 Cache-Aware Techniques 103
8.1.1 Heat 103
8.1.2 Gauss elimination 103
8.1.3 Floyd-Warshall 106
8.1.4 Quicksort 108
8.1.5 Matrix Multiplication 111
8.2 Load Balancing Techniques 113
8.2.1 Searching for the heaviest once in five steals 113
8.2.2 Global vs Local Sorted List 115

List of Figures

Figure 1. Shared Memory Architecture....23
Figure 2. Distributed Memory Architecture....24
Figure 3. Hybrid Architecture....25
Figure 4. Amdahl's Law....27
Figure 5. TBB Components....38
Figure 6. Scheduler Architecture Overview....39
Figure 7. Recursive Splitting & Work Stealing 1....42
Figure 8. Recursive Splitting & Work Stealing 2....43
Figure 9. 24-core "Dunnington" SMP Platform....46
Figure 10. 12-core "Termi" NUMA Platform....46
Figure 11. Blackscholes speedup on SMP....50
Figure 12. Blackscholes speedup on NUMA....50
Figure 13. Blackscholes User-Library time on SMP for each thread count....50
Figure 14. Blackscholes User-Library time on NUMA for each thread count....50
Figure 15. Blackscholes basic functionalities breakdown on SMP for each thread count....51
Figure 16. Blackscholes basic functionalities breakdown on NUMA for each thread count....51
Figure 17. Blakscholes basic functionalities' scalability on SMP for each thread count....51
Figure 18. Blackscholes basic functionalities' scalability on NUMA for each thread count....51
Figure 19. Blackscholes stealing components breakdown on SMP for each thread count 52
Figure 20. Blackscholes stealing components breakdown on NUMA for each thread count....52
Figure 21. Blackocholes scalability of stealing components on SMP for each thread count....52
Figure 22. Blackscholes scalability of stealing components on NUMA for each thread count....52
Figure 23. Fluidanimate speedup on SMP....53
Figure 24. Fluidanimate speedup on NUMA....53
Figure 25. FluidanimateUser-Library time on SMP for each thread count....53
Figure 26. FluidanimateUser-Library time on NUMA for each thread count....53
Figure 27. Fluidanimate basic functionalities breakdown on SMP for each thread count54
Figure 28. Fluidanimate basic functionalities breakdown on NUMA for each thread count....54
Figure 29. Fluidanimate basic functionalities' scalability on SMP for each thread count 54
Figure 30. Fluidanimate basic functionalities' scalability on NUMA for each thread count....54
Figure 31. Fluidanimate stealing components breakdown on SMP for each thread count....55

Figure 32. Fluidanimate stealing components breakdown on NUMA for each thread count 55
Figure 33. Fluidanimate scalability of stealing components on SMP for each thread count 55
Figure 34. Fluidanimate scalability of stealing components on NUMA for each thread count 55
Figure 35. Strassen speedup on SMP 56
Figure 36. Strassen speedup on NUMA 56
Figure 37. StrassenUser-Library time on SMP for each thread count 56
Figure 38. Strassen User-Library time on NUMA for each thread count 56
Figure 39. Strassen basic functionalities breakdown on SMP for each thread count 57
Figure 40. Strassen basic functionalities breakdown on NUMA for each thread count 57
Figure 41. Strassen basic functionalities' scalability on SMP for each thread count 57
Figure 42. Strassen basic functionalities' scalability on NUMA for each thread count 57
Figure 43. Strassen stealing components breakdown on SMP for each thread count 58
Figure 44. Strassen stealing components breakdown on NUMA for each thread count 58
Figure 45. Strassen scalability of stealing components on SMP for each thread count 58
Figure 46. Strassen scalability of stealing components on NUMA for each thread count 58
Figure 47. Streamcluster speedup on SMP 59
Figure 48. Streamcluster speedup on NUMA 59
Figure 49. Streamcluster User-Library time on SMP for each thread count 59
Figure 50. Streamcluster User-Library time on NUMA for each thread count 59
Figure 51. Streamcluster basic functionalities breakdown on SMP for each thread count 60
Figure 52. Streamcluster basic functionalities breakdown on NUMA for each thread count 60
Figure 53. Streamcluster basic functionalities' scalability on SMP for each thread count 60
Figure 54. Streamcluster basic functionalities' scalability on NUMA for each thread count 60
Figure 55. Streamcluster stealing components breakdown on SMP for each thread count 61
Figure 56. Streamcluster stealing components breakdown on NUMA for each thread count 61
Figure 57. Streamcluster scalability of stealing components on SMP for each thread count 61
Figure 58. Streamcluster scalability of stealing components on NUMA for each thread count 61
Figure 59. The numbers represent the cpu ids the OS assigns to each core on Dunnington 65
Figure 60. Cpu-id distribution to arena's slots 66
Figure 61. 32-core "Sandman" NUMA Platform 71

Figure 62. Speedup of 5-point Heat on Termi (Cache-aware) 73
Figure 63. Execution times of 5-point Heat on Termi (Cache-aware) 73
Figure 64. Performance gains of 5-point Heat on Termi (Cache-aware)....73
Figure 65. Speedup of 5-point Heat on Sandman (Cache-aware) 74
Figure 66. Execution times of 5-point Heat on Sandman (Cache-aware) 74
Figure 67. Performance gains of 5-point Heat on Sandman (Cache-aware)....74
Figure 68. Word Count speedup on Sandman (Cache-aware)....75
Figure 69. Word Count execution times on Sandman (Cache-aware) 75
Figure 70. Word Count performance gains on Sandman (Cache-aware) 76
Figure 71. Word Count speedup on Dunnington (Cache-aware)....76
Figure 72. Word Countexecution times on Dunnington (Cache-aware) 76
Figure 73. Word Count speedup on Termi (Cache-aware)....77
Figure 74. Word Count execution times on Termi (Cache-aware) 77
Figure 75. Streamcluster speedup on Dunnington (Just pick max)....78
Figure 76. Streamcluster execution times on Dunnington (Just pick max) 78
Figure 77. Streamcluster speedup on Termi (Just pick max)....79
Figure 78. Streamcluster execution times on Termi (Just pick max) 79
Figure 79. Streamcluster once in five steals speedup on Dunnington (Once in five)....80
Figure 80. Streamcluster once in five execution times on Dunnington (Once in five)....80
Figure 81. Streamcluster once in five Dunnington % Performance Improvement (Once in five)....80
Figure 82. Streamcluster once in five Sandman Speedup (Once in five)....81
Figure 83. Streamcluster once in five Sandman Execution Times (Once in five)....81
Figure 84. Streamcluster once in five Sandman % Performance Improvement (Once in five)....82
Figure 85. Streamcluster speedup on Dunnington (sorted-list) 83
Figure 86. Streamcluster performance gains on Dunnington (sorted list)....83
Figure 87. Quicksort speedup on Dunnington (sorted list)....84
Figure 88. Quicksort performance gains on Dunnington (sorted list) 84
Figure 89. Matrix multiplication speedup on Sandman (sorted list) 85
Figure 90. Matrix multiplication performance gains on Sandman (sorted list)....85
Figure 91. Quicksort speedup on SMP....90
Figure 92. Quicksort speedup on NUMA 90
Figure 93. Quicksort User-Library time on SMP for each thread count....91
Figure 94. Quicksort User-Library time on NUMA for each thread count 91
Figure 95. Quicksort basic functionalities breakdown on SMP for each thread count 91
Figure 96. Quicksort basic functionalities breakdown on NUMA for each thread count..91
Figure 97. Quicksort basic functionalities' scalability on SMP for each thread count 92
Figure 98. Quicksort basic functionalities' scalability on NUMA for each thread count..92
Figure 99. Quicksort stealing components breakdown on SMP for each thread count 92
Figure 100. Quicksort stealing components breakdown on NUMA for each thread count 92

Figure 101. Quicksort scalability of stealing components on SMP for each thread count 93
Figure 102. Quicksort scalability of stealing components on NUMA for each thread count 93
Figure 103. Swaptions speedup on SMP 93
Figure 104. Swaptions speedup on NUMA 93
Figure 105. Swaptions User-Library time on SMP for each thread count 94
Figure 106. Swaptions User-Library time on NUMA for each thread count 94
Figure 107. Swaptions basic functionalities breakdown on SMP for each thread count .. 94
Figure 108. Swaptions basic functionalities breakdown on NUMA for each thread count 94
Figure 109. Swaptions basic functionalities' scalability on SMP for each thread count ... 95
Figure 110. Swaptions basic functionalities' scalability on NUMA for each thread count 95
Figure 111. Swaptions stealing components breakdown on SMP for each thread count .. 95
Figure 112. Swaptions stealing components breakdown on NUMA for each thread count 95
Figure 113. Swaptions scalability of stealing components on SMP for each thread count 96
Figure 114. Swaptions scalability of stealing components on NUMA for each thread count 96
Figure 115. Matrix multiplication speedup on SMP 96
Figure 116. Matrix multiplication speedup on NUMA 96
Figure 117. Matrix multiplication User-Library time on SMP for each thread count 97
Figure 118. Matrix multiplication User-Library time on NUMA for each thread count .. 97
Figure 119. Matrix multiplication basic functionalities breakdown on SMP for each thread count 97
Figure 120. Matrix multiplication basic functionalities breakdown on NUMA for each thread count 97
Figure 121. Matrix multiplication basic functionalities' scalability on SMP for each thread count 98
Figure 122. Matrix multiplication basic functionalities' scalability on NUMA for each thread count 98
Figure 123. Matrix multiplication stealing components breakdown on SMP for each thread count 98
Figure 124. Matrix multiplication stealing components breakdown on NUMA for each thread count 98
Figure 125. Matrix multiplication scalability of stealing components on SMP for each thread count 99
Figure 126. Matrix multiplication scalability of stealing components on NUMA for each thread count 99
Figure 127. Convex Hull speedup on SMP 99
Figure 128. Convex Hull speedup on NUMA 99
Figure 129. Convex Hull User-Library time on SMP for each thread count 100

Figure 130. Convex Hull User-Library time on NUMA for each thread count 100
Figure 131. Convex Hull basic functionalities breakdown on SMP for each thread count 100
Figure 132. Convex Hull basic functionalities breakdown on NUMA for each thread count 100
Figure 133. Convex Hull basic functionalities' scalability on SMP for each thread count 101
Figure 134. Convex Hull basic functionalities' scalability on NUMA for each thread count 101
Figure 135. Convex Hull stealing components breakdown on SMP for each thread count 101
Figure 136. Convex Hull stealing components breakdown on NUMA for each thread count 101
Figure 137. Convex Hull scalability of stealing components on SMP for each thread count 102
Figure 138. Convex Hull scalability of stealing components on NUMA for each thread count 102
Figure 139. Speedup of 5-point Heat on Dunnington(Cache-aware) 103
Figure 140. Execution Times of 5-point Heat on Dunnington(Cache-aware) 103
Figure 141. Gauss elimination Speedup on Dunnington(Cache-aware) 104
Figure 142. Gauss elimination execution times on Dunnington(Cache-aware) 104
Figure 143. Gauss elimination speedup on Termi(Cache-aware) 104
Figure 144. Gauss elimination execution times on Termi(Cache-aware) 105
Figure 145. Gauss elimination speedup on Sandman(Cache-aware) 105
Figure 146. Gauss elimination execution times on Sandman(Cache-aware) 105
Figure 147. Gauss elimination % performance improvement on Sandman(Cache-aware) 106
Figure 148. Floyd-Warshall speedup on Sandman(Cache-aware) 106
Figure 149. Floyd-Warshall execution times on Sandman(Cache-aware) 106
Figure 150. Floyd-Warshall performance gains on Sandman(Cache-aware) 107
Figure 151. Floyd-Warshall speedup on Termi(Cache-aware) 107
Figure 152. Floyd-Warshall execution times on Termi(Cache-aware) 107
Figure 153. Floyd-Warshall speedup on Dunnington(Cache-aware) 108
Figure 154. Floyd-Warshall execution times on Dunnington(Cache-aware) 108
Figure 155. Quicksort speedup on Dunnington(Cache-aware) 109
Figure 156. Quicksort execution times on Dunnington(Cache-aware) 109
Figure 157. Quicksort speedup on Termi(Cache-aware) 109
Figure 158. Quicksort execution times on Termi(Cache-aware) 110
Figure 159. Quicksort speedup on Sandman(Cache-aware) 110
Figure 160. Quicksort execution times on Sandman(Cache-aware) 110
Figure 161. Quicksort performance benefits on Sandman(Cache-aware) 111
Figure 162. Matrix multiplication speedup on Dunnington(Cache-aware) 111

Figure 163. Matrix multiplication execution times on Dunnington(Cache-aware)..... 111
Figure 164. Matrix multiplication speedup on Termi(Cache-aware) 112
Figure 165. Matrix multiplication execution times on Termi (Cache-aware) 112
Figure 166. Matrix multiplication speedup on Sandman (Cache-aware) 112
Figure 167. Matrix multiplication execution times on Sandman (Cache-aware) 113
Figure 168. Blackscholes speedup on Dunnington (once in five) 113
Figure 169. Blackscholes execution times on Dunnington (once in five) 114
Figure 170. Blackscholes performance gains on Dunnington (once in five) 114
Figure 171. Streamcluster speedup on Termi (sorted list) 115
Figure 172. Streamcluster performance gains on Termi (sorted list)..... 115
Figure 173. Streamcluster speedup on Sandman (sorted list)..... 116
Figure 174. Streamcluster performance gains on Sandman (sorted list) 116
Figure 175. Quicksort speedup on Termi (sorted list)..... 117
Figure 176. Quicksort performance gains on Termi (sorted list) 117
Figure 177. Quicksort speedup on Sandman (sorted list)..... 117
Figure 178. Quicksort performance gains on Sandman (sorted list) 118
Figure 179. Matrix multiplication speedup on Dunnington (sorted list) 118
Figure 180. Matrix multiplication performance gains on Dunnington (sorted list)..... 118
Figure 181. Matrix multiplication speedup on Termi (sorted list) 119
Figure 182. Matrix multiplication performance gains on Termi (sorted list)..... 119
Figure 183. Strassen speedup on Dunnington (sorted list)..... 119
Figure 184. Strassen performance gains on Dunnington (sorted list) 120
Figure 185. Strassen speedup on Termi (sorted list)..... 120
Figure 186. Strassen performance gains on Termi (sorted list) 120
Figure 187. Strassen speedup on Sandman (sorted list)..... 121
Figure 188. Strassen performance gains on Sandman (sorted list) 121
Figure 189. Blackscholes speedup on Dunnington (sorted list) 122
Figure 190. Blackscholes performance gains on Dunnington (sorted list)..... 122
Figure 191. Blackscholes speedup on Termi (sorted list) 122
Figure 192. Blackscholes peformance gains on Termi (sorted list) 123
Figure 193. Blackscholes speedup on Sandman (sorted list) 123
Figure 194. Blackscholes performance gains on Sandman (sorted list)..... 123
Figure 195. Swaptions speedup on Dunnington (sorted list)..... 124
Figure 196. Swaptions performance gains on Dunnington (sorted list) 124
Figure 197. Swaptions speedup on Termi (sorted list)..... 125
Figure 198. Swaptions performance gains on Termi (sorted list) 125
Figure 199. Swaptions speedup on Sandman (sorted list)..... 126
Figure 200. Swaptions performance gains on Sandman (sorted list) 126

List of Listings

Listing 1. Floyd-Warshall algorithm 35
Listing 2. Algorithm for computing the n-th Fibonacci number 36
Listing 3. Basic task dispatch loop 47
Listing 4. Worker stealing loop 67
Listing 5. Sorted-list technique algorithm 70

Chapter 1
Introduction

1.1 Overview

In 1965 Gordon E. Moore published a paper [1] that affected the pace in which microprocessors evolved. In this paper he stated that the number of transistors on integrated circuits would continue to double every two years, a trend confirmed by observations on computer hardware history at the time. This prediction undoubtedly continues to affect the computer hardware industry to this day, sometimes being the push behind modern efforts for increased performance. Since the appearance of the first microprocessor IBM chip in 1971, uniprocessor chips have dominated the computing industry for three long decades. During this period, the increase in transistor density was exploited mainly by an increased clock frequency, execution optimizations and caches. Increasing the clock speed is more or less about running the same work faster. Optimizing execution flow tried to make the instructions flow better and faster, squeezing the most work out of each clock cycle by reducing latency and maximizing the work accomplished per clock cycle. Finally, increasing the size of on-chip cache is about putting the most useful data closer to the processor, as main memory continues to be so much slower than the CPU. It is important to point out that all these improvements aimed at making sequential programs run faster.

Due to physical limitations, CPU performance growth hit a wall around 2003. The clock race between manufacturers has led up to 3,8 GHz, where it became harder and harder to exploit higher clock speeds. Heat dissipation, power consumption and current leakage problems are the main obstacles yet to overcome. Thus, in order to exploit the still increasing transistor density, industry has shifted towards multicore architectures. This shift signaled the end of the free-lunch era [2], where improvement in performance was offered freely by the architectural improvements, without any effort by the programmer. Applications would no longer benefit from performance gains without significant redesign. Multicore architectures have unraveled a new world for Computer Science, where introducing new runtime environments and programming models are essential to exploit the new hardware.

With almost a decade having passed, multicore systems and parallel applications have become the standard. Operating Systems and Compilers have evolved to support the new hardware, desktop applications use multiple parallel threads and even traditionally serial algorithms have been replaced by parallel and distributed alternatives, promising better scaling and improved performance in multicore environments. The prevalent class of applications to be benefited from multicore consists of computation-intensive applications.

Except for scientific applications, which are traditionally computationally demanding, customer-oriented applications, including computer graphics, database management and machine learning, do have increasing demands in computational resources, in an effort to manage unprecedentedly large datasets and reduce response time.

1.2 Multi-socket, multi-processor systems

Before explaining the common parallel architectures of our interest, we shall introduce Flynn's taxonomy[3] of computer architectures, according to the level of parallelism they employ to process instructions and data streams.

- SISD: Single instruction, Single Data
  A sequential (or uniprocessor) computer. No parallelism employed.
- SIMD: Single Instruction, Multiple Data
  A computer which concurrently processes multiple data streams with a single instruction stream, to perform operations that may be parallelized.
- MISD: Multiple Instruction, Single Data
  Uncommon, non-commercial architecture, used only for scientific purposes, as fault tolerance.
- MIMD: Multiple Instruction, Multiple Data
  Each processor executes its own instruction stream and processes its own data stream. This architecture supports multiple threads (thread-level parallelism). Multicore processors and clusters are examples of MIMD architectures.

Parallel computers are based on MIMD architectures, which can be further classified according to their memory organization, into shared-memory architectures, distributed-memory architectures and hybrid architectures and are profoundly analyzed below.

1.2.1 Shared Memory Architectures

A Shared Memory Architecture is a memory organization scheme that offers a shared memory address space to the programmer. Communication in this scheme is carried out using variables in memory, which are accessed and modified using loads and stores. Each processor has its own cache hierarchy. A typical Shared Memory Architecture is shown in figure 1. Shared Memory Architectures can provide *Uniform Memory Access* (UMA), where accesses from any processor to any memory address take the same amount of time, or *Non-Uniform Memory Access* (NUMA), where memory access time varies among different memory addresses, depending on the processor and the topology. Obviously, NUMA architectures offer very low latencies for nearby memory accesses and lower memory bus congestion when used correctly, introduce though challenges in program development, due to their complicated topological peculiarity, a tradeoff that should be taken into consideration.

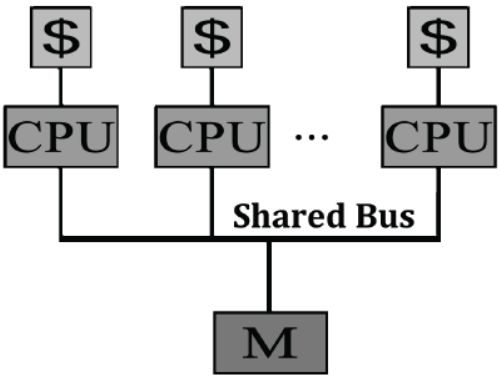

Figure 1. Shared Memory Architecture

Shared Memory Architectures offer ease of programmability, since parallel programs can operate on the same collections of data, which are present in memory only once. However, such an approach hides a lot of pitfalls, as concurrent modification of the same data by different processors can lead to inconsistent data, thus requiring a synchronization mechanism to ensure the validity of data. Such mechanisms are usually locks or mutexes, so as to ensure that only one processor enters a critical section of the code at a time. Moreover, cache coherence protocols are implemented to impose a universal sequence of access to the main memory.
Although attractive for parallel programming, Shared Memory Architectures can be used for connecting only small numbers of processors, up to a few dozens, since such architectures don't scale well. The reason for that is that all processors compete for the same bus and memory system, which have limited bandwidth, leading to a saturation after adding more than 30-40 processors.

1.2.2 Distributed Memory Architectures

A Distributed Memory Architecture is a memory scheme consisting of a network of separate processing elements, that are offered no shared memory address space and each node has access only to its own private memory address space. Each processor has its own cache hierarchy and processors are connected using an interconnection network, with different implementations varying in characteristics, such as latency, throughput and scalability. A typical Distributed Memory Architecture is shown in figure 2. Computational tasks can only operate on local data and if remote data is required, the computational task must communicate with one or more remote processors to serve its request. Communication in Distributed Memory Architectures is carried out using explicit send and receive routines to send and receive data.

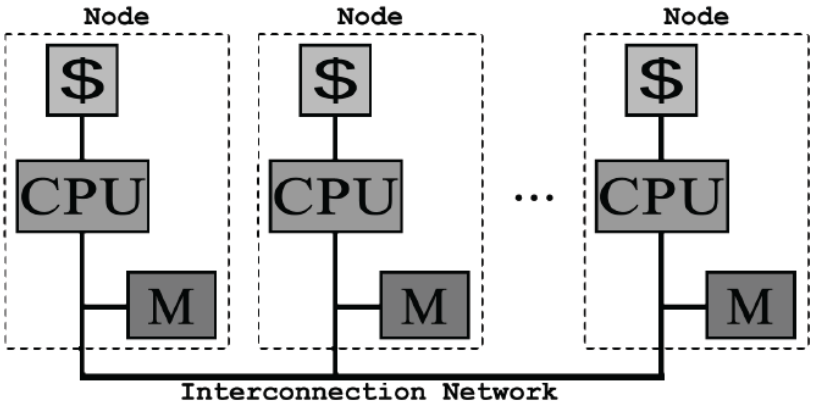

Figure 2. Distributed Memory Architecture

Clusters are usually built of commodity computers, using the same operating system, physically connected through cables and switches, following some network topology. Software gets involved to manage communication between non-neighboring nodes. To decouple communication operations from the processors, direct memory access controllers (DMA) and routers are employed, which both enable data transfer directly from the local memory.

One important drawback of clusters is their management cost. Managing a cluster consisting of n nodes equals to the cost of managing n computers, while the cost of managing a multiprocessor of n cores equals to the cost of managing a single computer. Furthermore, the interconnection network adds extra latency to the communication process, compared to a memory bus, which increases with the number of nodes. On the other hand, a cluster is a low-cost solution to gain high performance. Scalability comes naturally by adding more independent nodes to the network, enabling modern supercomputers to have thousands of nodes, which can be maintained or replaced with no functioning effect on the system.

Programming on a Distributed Memory Architecture is a far more challenging issue that on a Shared Memory Architecture, since communication and data transfer overheads have to be identified in advance and implemented explicitly. On the other hand, Distributed Memory Architectures scale up to thousands of nodes, since they are constructed using independent nodes and interconnection networks, avoiding the bottlenecks that appear in Shared Memory Architectures.

1.2.3 Hybrid Architectures

A Hybrid Memory Architecture is a memory organization scheme that follows the Distributed Memory scheme, where a symmetric multiprocessor (SMP) has taken the place of each single processor node. Each node has its own private memory address space and shared memory parallel programming techniques can be employed within it, whereas the system scales up in the same way as a distributed memory system, simply by connecting more SMPs to the network. A typical Hybrid Memory Architecture is shown in figure 3. This architecture tries to combine the benefits of both memory architectures and is the typical architecture of modern clusters and supercomputers.

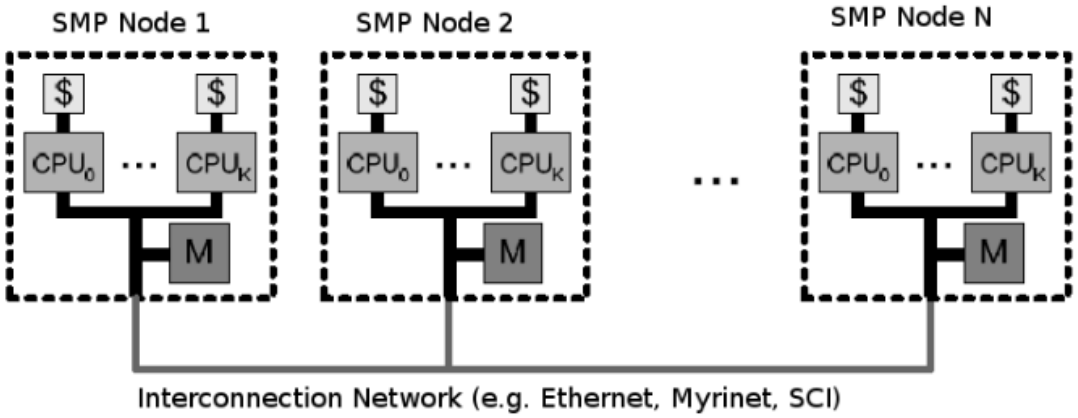

Figure 3. Hybrid Architecture

1.2.4 A Multi-Socket Multi-Core Machine

A Multi-Socket Multi-Core Machine refers to a shared memory architecture consisting of several multicores on the same machine, each residing on a socket. Each multiprocessor has its own cache hierarchy, and each cache memory level can be shared between two or more processors.

1.3 Parallel Programming, Amdahl's Law, scalability

Parallel Programming makes sense when it enables the programmer to achieve speedup of his application execution time. Despite being the main objective, no explicit formula exists for the parallelization of sequential algorithms and programs. Thus, the programmer bears the burden of exploring the potential parallelism of an algorithm, with respect to its semantics, and resolving issues that directly affect the execution time of the parallel program.

We will elaborate on these issues later on, but before that we define some main performance metrics of parallel programming, before we introduce the main parallel programming models:

- $T_p(n)$: the parallel runtime of a program of size n on p processors
- $S_p(n) = \frac{T^*(n)}{T_p}$: the speedup in execution time that a parallel program achieves, compared to the sequential equivalent. $T^*(n)$ is the runtime of the fastest sequential program. In essence, it is the relative saving of execution time that can be obtained by using a parallel execution on p processors compared to the best sequential program. If the inequality $S_p(n) \leq p$ holds, then the parallel implementation is efficient. If $S_p(n) = p$, the speedup is *linear*.
- $Efficiency = \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p}$: it measures return on hardware investment. Ideal efficiency is 1 (often reported as 100%), which corresponds to linear speedup.

Linear speedup is rare in practice, since there is extra work involved in distributing work to processors and coordinating them. In addition, and optimal serial algorithm may be able to do less work overall than an optimal parallel algorithm for certain problems, so the achievable speedup may be sublinear, even on theoretical ideal machines.
Interestingly, superlinear speedup (or efficiency greater that 100%) can be achieved. Some common cases of superlinear speedup include:

- Restructuring a program for parallel execution can cause it to use cache memory better, even when run on a single processor.
- The program's performance is strongly dependent on having a sufficient amount of cache memory, and no single processor has access to that amount. If multiple processors bring that amount to bear, because they do not all share the same cache, absolute speedup can be superlinear.
- The parallel algorithm may be more efficient than the equivalent serial algorithm, since it may be able to avoid work that its serialization would be forced to do. For example, in tree search problems, searching multiple branches in parallel sometimes permits chopping off branches sooner than would occur in the serial code.

If the cost of the best sequential program is unknown or varies depending on the data set, then speedup is often computed by using a sequential version of the parallel implementation.
In the early years of high performance computing, Gene M. Amdahl [4] first denoted some inherent constraints in the process of parallel programming. There is a fraction of computational load in every application, associated with data management, which cannot be executed in parallel with other computations and other acts as a constant overhead to the runtime. To model this restriction, Amdahl introduced his famous law, which sets the limit of the speedup potential of the program, according to the following formula

$$S_p(n) = \frac{1}{r_s + \frac{r_p}{p}}$$

where r_p is the portion of the program that can be parallelized and r_s the serial portion of the program ($r_p = 1 - r_s$).
Amdahl's law is a useful measure of the best case execution time for a parallel program. If the number of processors p goes to infinity, the total speedup goes to $1/r_s$. If the parallelizable part of the program is relatively small, its speedup would be respectively small, regardless to the number of the processing units. Figure XX depicts the influence of Amdahl's law in parallel executions of different sequential fractions. In terms of programming recipes, Amdahl's law should be interpreted as follows: the programmer should try to parallelize (or optimize) the parts of the code that consume the greatest fraction of time. He also should try to parallelize all parts of the program (initialization faces, memory allocation etc.), because if for example there is a 10% serial fraction of code in our program, the maximum speedup potential is only 10.

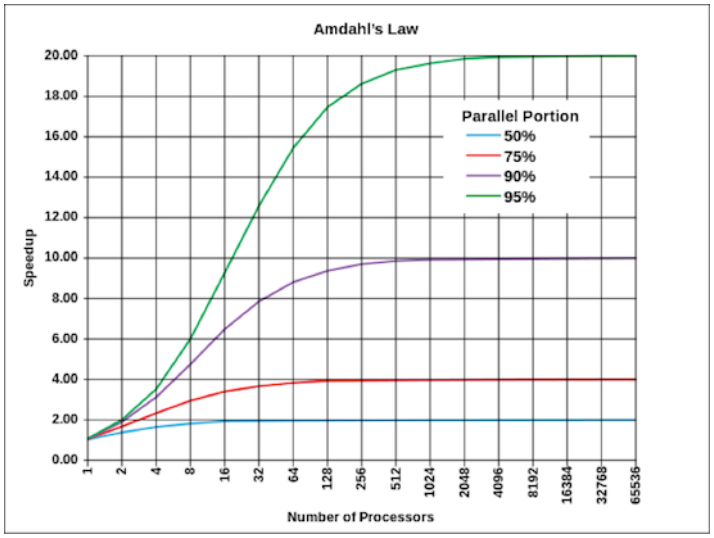

Figure 4. Amdahl's Law

1.4 Parallel Programming Models

Parallel programming models are the interface between the hardware and the programmer, offering an abstraction level to facilitate parallel programming on the diverse parallel architectures. It enables the expression of parallel programs which can be compiled and executed. Thus, there exist three general parallel programming models, with respect to the aforementioned architectures: the shared memory model, the message passing model and the hybrid model. They can be implemented as language extensions, runtime libraries of programming languages or even autonomous execution models. They map the more or less high level programming constructs to lower level primitives, which are provided by the underlying system. The mapping may make use of hardware provided tools (like specific machine instructions) or operating systems constructs (like threads).

1.4.1 Shared Memory programming model

A shared memory programming model enables the programmer to partition a programming task into multiple threads which run in parallel on the cores of a multiprocessor with a shared address space. Communication between threads is handled via load and store operations on the shared address space, bearing in mind that whenever a processor writes to a shared memory access, all processors accessing the same address will be aware of the change. In order to maintain data consistency, synchronization mechanisms are needed, such as barriers and locks, preventing race conditions from affecting the parallel program's correctness.

Shared memory programming models offer ease of programmability, as they facilitate data exchange and communication through a simple annotation of a variable as shared, thus visible and accessible to all processing units. Furthermore, these programming mod-

els supply the programmers with several parallel constructs, easily applicable to sequential programs for their parallelization, such as parallel-for loops. The price for these conveniences is the complexity of identifying and resolving race conditions even for a highly skilled programmer. Managing shared data often leads to subtle and not easily traceable bugs, which makes large-scale parallel software development very error prone, tricky and time-consuming, affecting the productivity.
It is rather straightforward and efficient to implement shared memory programming models for shared memory platforms, carrying though the disadvantage of their limited scalability. On the other hand, implementation on distributed memory platforms, although feasible, requires special performance degrading software layers and costly hardware support.
Commonly used shared memory programming models are PThreads, OpenMP, CilkPlus and Intel TBBs. The latter will be thoroughly discussed in the following chapter.

1.4.2 Distributed Memory programming model

In a message passing programming model, the program consists of a set of independent processes, where the same instructions may reside on distinct computing nodes or computers. Each process owns a local private address space and sends and receives messages to and from other processes to achieve inter-process communication and data exchange. Message passing is executed by the operating system or by function calls to the runtime library that activates low level operations. In a trivial approach of this model, a send operation involves a local buffer where the message to be sent is stored and a receiving process, whereas its complementary receive operation involves a local buffer where the message to be received will be stored. Modern approaches have though a little different implementation. The message sent is copied into an internal system buffer of the runtime system, thus the sending process can continue after the copying operation is completed, while the receiving process copies the data from the internal buffer of the runtime.
A more general classification of the communication in a message passing model is based on whether it is performed in a synchronous or asynchronous manner. *Synchronous* (also known as *blocking*) message passing refers to the case where both the sending and the receiving process block all their other operations until data exchange is accomplished. The message is immediately stored in the receiving process's local memory and no synchronization mechanism is required, as both processes involved are synchronized at the end of the communication. In *asynchronous* or *non-blocking* message passing, the message is sent by the sending process without waiting for the receiving process to be ready to receive. Both processes may continue with their tasks until lower-level operations deliver the message. A disadvantage of asynchronous communication is that it involves an internal buffer, which, if full, may lead to a deadlock.
Communication can also be categorized as *point-to-point* or *collective*, depending on the number of processes that exchange data. Point-to-point is when a single process sends

data to e single receiving process, while collective communication involves more than two processes, with multiple sending and receiving points.
Naturally, message passing models serve better parallel programming on distributed memory computing systems, which appeared long before shared memory parallel systems. The de facto model for message passing in clusters is the MPI (Message Passing Interface) standard library. The performance of such models on clusters is determined by the communication efficiency, which relies on the interconnection network. With increasing number of nodes, a significant overhead is added to message passing delays, which cannot be modeled with Amdahl's law. MPI is applicable to shared memory architectures as well, though for reasons of performance the interconnection network is bypassed and message passing is served by shared memory operations.
Programming with message passing models can be a challenging job. The programmer has to design the parallel program from scratch, make decisions about data distribution, message passing patterns and synchronization points. MPI is other than that error prone, as it involves employing the MPI routines that match to the aforementioned decisions, which can be cumbersome and non-trivial for the average programmer, requiring time-consuming debugging processes. The non-trivial programmability though, when it leads to a fine parallel implementation, has rewarding results, as the program can be highly efficient and scalable, compared to its shared memory alternative.
Another popular implementation of message passing model is the *Actor Model*, implemented in programming languages like Erlang and Scala. In the actor model, each object is an actor. This is an entity that has a mailbox and a behavior. Messages can be exchanged between actors, which will be buffered in the mailbox. Upon receiving a message, the behavior of the actor is executed, upon which the actor can: send a number of messages to other actors, create a number of actors and assume new behavior for the next message to be received. All communications are performed asynchronously. This implies that the sender does not wait for a message to be received upon sending it, it immediately continues its execution. There are no guarantees in which order messages will be received by the recipient, but they will eventually be delivered.

1.4.3 Hybrid Programming model

The hybrid programming model is a combination of a shared memory and message passing model. A common hybrid model is the joint use of MPI and OpenMP. This model is suited for hybrid architectures, as described above, where the shared memory is used to parallelize a program at the interior of a node of an SMP cluster and the message passing model is used for the communication between processes residing on distinct nodes. Except for OpenMP, other shared memory implementations can be used, such as Intel TBBs.

1.5 Overview of Key Features for Performance

Although optimizing code cannot be dealt with in a generic manner, mainly because it depends highly on the specific architectural characteristics of the underlying machine, modern architectures have been designed with two major key assumptions: Data Locality and Parallel Slack.
Data Locality refers to reusing data from nearby locations with regard to time or space. So algorithms should be designed having in mind some of the following rules:

- Chunking the work in order to fit in cache. If the working set doesn't fit in cache, there will be a certain performance degradation due to capacity cache misses.
- Data structures and memory accesses should be organized to reuse data locally when possible. Especially, unnecessary memory accesses far apart in memory or simultaneous access to multiple memory locations located a power of two apart should be avoided.
- Accessing too many pages at once could cause unnecessary TLB misses.
- It is very important to align data with cache line boundaries. Unrelated data accesses from different cores to the same cache lines should be avoided, as they may cause false sharing.

Avoiding some of the above may require changes to data layout, including reordering items and adding padding to achieve (or avoid) alignments with the hardware architecture. It is noteworthy that breaking up the work into chunks and getting good alignment with cache is also beneficial to single-core architectures.

Following the above rules assumes knowledge of cache line sizes, cache organization, or the total size of the caches, which are not a given when writing portable code. In this case, the memory allocation routines should be customized so that they select the chunk size in a dynamic manner, either by hand-tuning them when porting to a new machine, or by writing auto-tuning routines. Using cache oblivious algorithms, that is, algorithms using recursive decomposition, is another approach to auto-tuning.

In this thesis, we will be referring to shared memory architectures. These architectures have the property that groups of cores compete for the usage of a single memory bus. In this case, another important factor that affects performance is arithmetic intensity, the ratio of computation to communication. Given the fact that on-chip compute performance is still rising with the number of transistors, but off-chip bandwidth is not rising as fast, in order to achieve scalability a large number of on-chip computations should be performed for every off-chip memory access. This can be achieved through a range of optimizations, including fusion and tiling. As a rule of thumb large enough chunks of work that fit in cache should bring in practice the best performance. However, larger chunks of work reduce the available parallelism since it will reduce the total number of work units.

Parallel Slack refers to the amount of extra parallelism available above the minimum necessary to keep the parallel hardware resources utilized. Specifying an amount of po-

tential parallelism higher than the actual parallelism offered by the hardware gives the underlying software and hardware schedulers more flexibility to exploit machine resources.

The ideal strategy would be to choose work units of size that reasonably amortizes the overhead of partitioning and scheduling them and offer good arithmetic intensity. Breaking the problem down to the exact amount of hardware parallelism may sound tempting, it isn't though the best strategy. In case a task delays for some reason (for example an operating system interrupt), it will inevitably delay the entire program.

1.6 Problems and pitfalls of parallel programming that should be avoided

- Race Conditions: they occur when concurrent tasks perform actions on the same memory location without proper synchronization. When entering a critical section of a parallel program, shared data that are accessed can cause unpredictable behavior without synchronization. That can be caused because there is no guarantee about the order that the operations are going to be executed by the hardware, so the outcome is likely to be corrupted data. If you are unlucky, a program with data races can work fine during tasting but fail once it is in the customer's hands. Even considering the possible interleaving of instructions isn't enough to predict data races, because modern hardware usually if not sequentially consistent. That means that hardware and the compiler may produce different reordering between operations. Avoiding races using special hardware features is a solution, though not a good one, as it kills portability. For this reason, the parallel programming model used and the programming language should offer a memory model that enables avoiding data races independently from the hardware details. Races are not limited to memory locations. They can happen with files and I/O too.
- Mutual Exclusion and Locks: Locks are a low-level way to eliminate races. Mutual exclusion can be achieved in many situations using a lock. The locking and unlocking are implemented using hardware instructions, in order to ensure atomicity. An important point about locks is that they should protect logical invariants and not specific memory locations. For example, in the case of a complex data structure as the linked list, a lock might protect the invariant "the *next* field of each element points to the next element in the list". Any time a task traverses a list, it must first take the lock, otherwise it might walk next fields under construction by another task. If a lock protects a specific memory location, the invariant may be temporarily violated inside the critical section, leading to unpredictable behaviors.
- Deadlock: it occurs when two concurrent tasks wait for each other, not being able to resume until the other task proceeds. This can happen when they try to ac-

quire more than one lock at the same time, in a way that creates cyclic dependencies. If for example task A tries to acquire locks L1 and L2, and task B tries to acquire locks L2 and L1 at the same time, it is possible that A acquires L1, B acquires L2, and then they wait each other. There are several ways to avoid deadlocks:

i. Holding at most one lock at a time: Never call other people's code while holding a lock, unless you are sure that the other code never acquires a lock.
ii. Always acquire multiple locks in the same order: a specific ordering to lock acquiring avoids deadlocks.
iii. Avoid locks when possible
iv. Backoff: when trying to acquire a lock, if it cannot be immediately acquired, release all locks already acquired. This approach requires a "try lock" operation that immediately returns if the lock cannot be acquired.

- Strangled Scaling: Locks serialize the program execution by nature, causing Amdahl bottlenecks to the overall computation. When tasks contend for the same lock, the impact on scaling can be severe, even worse than if the protected code was serial. Except for the bottleneck to execution, the status of the protected memory locations must be communicated between cores, thus adding communication costs not paid by the serial equivalent, which can be very costly when the underlying machine is multisocket.

  The locking can be either fine-grained or coarse-grained. Usually, fine-grained locking replaces a single highly contended lock with many uncontended locks, thus improving the scalability. Nevertheless, fine-grained locking can be tricky to implement.
- Load Imbalance: it refers to uneven distribution of work across workers. The time of the longest running task contributes to the span, which limits how fast the parallelized portion of the program can run. Load imbalance can be avoided by decomposing the work to small parallel chunks, thus making it easier to distribute to the workers available.
- Lack of locality: Locality of data can be either temporal or spatial. Temporal locality refers to using the same data in the near future, while spatial locality refers to using nearby data. In modern architectures, which use many levels of caches, either types of locality can lead to speedup. Communication is very expensive in these systems, while computation is very cheap. Thus, it is often preferable to increase the work in exchange for reducing communication.

  True and false sharing overhead caused by the cache coherence protocols can be very high in multisocket architectures, due to the data exchange through the intersocket interconnect. Also, a cache miss can take up to the order of a hundred cycles. So having good locality but also avoiding unnecessary sharing between cores are two requirements that both should be fulfilled, although in some cases they may contradict each other.

- Overhead of parallelization: the programmer should have in mind that launching and synchronizing parallel tasks introduces overhead, which increases the total amount of work to be done. Making tasks very small can help with load balancing, but it can cause very large overhead of managing them. Ideally, the decomposition of the work to parallel tasks should allow balancing the load while still making tasks large enough to amortize synchronization overhead and maximize arithmetic intensity. Launching and synchronizing the tasks in a tree structure can lead to a time overhead that is logarithmic to the number of the workers, instead of linear if all the parallel tasks were launched from one task.

1.7 Desired Properties of Parallel Programming Models

With the existing codebase consisting mainly of serial code, it is necessary to extend existing programming practices and tools to support parallelism. Broadly speaking, while enabling dependable results, parallel programming models should have the following properties:

- Performance: using the parallel programming model should be possible to predictably achieve good performance. Moreover, the performance should be easily tunable for different systems and should scale easily to larger systems.
- Productivity: Programming models should be highly expressive, debuggable and maintainable. A very important aspect that greatly contributes to the achievement of these requirements is composability, which will be further discussed later.
- Safety/Determinism: An inherent complication of parallel computation is nondeterminism. Determinism implies that running the same program multiple times produces the same result. Due to the randomness of thread scheduling, for reasons outside the control of the application, the order of operation of different threads may be interleaved in an arbitrary order. If the threads modify shared data (in a shared memory programming model), it is possible that different runs of a program may produce different results even with the same input. Although nondeterminism is not necessarily bad, many approaches to application testing assume determinism. In many cases, non-determinism is an error, as it leads to possible corruption of shared data. The problem of safety is how to ensure that only correct orderings occur.
- Portability of functionality: Being able to run code on a wide variety of platforms, regardless of operating systems, processors and compilers, is desirable.
- Portability of performance: Portability of performance is a serious concern. It is reassuring for the programmer to know that his code will continue to perform well on new machines and on machines he may not have tested it on. Ideally and application that is tuned to run within 80%of the peak performance of a machine should not suddenly run at 30% of the peak performance on another machine. This can be achieved only with more abstract programming models. Abstract models are removed enough from the hardware design to allow programs to map

to a wide variety of hardware without requiring code changes, while delivering reasonable performance relative to the machine's capability.

- Composability: it is the ability to use a feature without regard to other features being used elsewhere in the program. Ideally, every feature in a programming language is composable with every other. For example, if this property didn't hold for an if statement, then linking a library where any if statement was used would mean for statements would be disallowed throughout the rest of the application. As absurd as it may sound, similar situations exist in some parallel programming models or combinations of programming models. Incompatibility between programming models or constructs can lead to failure even if parallel regions do not directly invoke each other. Such situations can arise, for example, by inconsistent use of local thread memory. Another principal issue is the inability to support hierarchical composition. This commonly occurs when a program that is parallelized using a parallel programming model calls a library function which is parallelized using a different parallel programming model. To avoid this danger the programmer should know inner details of the library, which violates some fundamental principles of software engineering, such as information hiding and separation of concerns. So if the library is serial, and the next version becomes parallelized, upgrading to the newest version, although the binary interface is the same, might break the code with which it is combined.

Chapter 2
Motivation – Overview of the Problem

2.1 Regular vs. Irregular &Nested/Recursive Parallelism – Oversubscription - Implicit vs. Explicit Parallelism

In the previous chapter we presented an overview of the basic keys for performance in parallel programming, as well as common pitfalls and the desired properties of parallel programming.

In this section we will address two more aspects of parallel programming that often occur when parallelizing programs, namely *regular vs irregular* parallelism and *nested parallelism.*

Regular parallelism refers to the parallelization of an algorithm, which acts in a predictable and static manner on data, which means that the computations are already known to the programmer statically. An example of this kind of algorithms is the Floyd-Warshall algorithm.

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each vertex v
  dist[v][v] ← 0
for each edge (u,v)
  dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
for k from 1 to |V|
  for i from 1 to |V|
    for j from 1 to |V|
      if dist[i][k] + dist[k][j] < dist[i][j] then
        dist[i][j] ← dist[i][k] + dist[k][j]
```

Listing 1. Floyd-Warshall algorithm

In this case the programmer knows statically the order of computations, thus being able to identify the loops that can be parallelized, divide the work and data between threads and estimate very precisely the amount of work that each thread will have to do, compute the work span and devise the way to parallelize it effectively.

On the contrary, an irregular algorithm is one that the amount of work depends on the instance of the problem and is unpredictable by nature. For example the BFS or the A* algorithm has this kind of "irregularity", as the number of neighboring nodes that will be explored for every node on the search front cannot be foreseen and they depend on the topology of the graph that is explored. In this case, the programmer cannot simply divide

the work to a number of threads, because it is very likely that some of these will have to do very little work, while others may face exponential increase in the work to be done. If for example a thread follows a linear path on the graph, it will just explore subsequent nodes, while another thread could face a tree-like structure of path alternatives that should explore. That would lead to work imbalance, making the critical path ridiculously long, resulting to huge performance degradation.

Nested parallelism refers to the situation when a programmer wants to parallelize a piece of code which is nested in an outer piece of code that is already parallelized. A common example is the parallelization of nested loops. As it is not usually done on purpose, another more non-trivial example could be the following: Suppose an algorithm f is parallelized, by creating 15 extra threads to assist the calling thread, and each thread calls a library routine g. If the implementer of g applies the same logic, now there are 16x15 threads running concurrently. A special case of nested parallelism is recursive parallelism. This occurs when the algorithm to parallelize is recursive by nature and is difficult to transform it to its iterative equivalent. Even if it is not that difficult, it sure loses the elegance, readability and maintainability of the recursive formula. Also, a recursive form of an algorithm could be cache oblivious, thus enabling better use of the cache memory. A classic example of recursive algorithm is the computation of the nth Fibonacci number, given by the following algorithm:

```
int Fib(int n)
{
        if (n <= 1)
                return 1;
        else
                return Fib(n - 1) + Fib(n - 2);
}
```

Listing 2. Algorithm for computing the n-th Fibonacci number

If the programmer tries to spawn a thread for each of the two recursive calls, it could result in a huge number of threads being spawned, which is undesirable. In either cases, creating a very large number of threads could cause oversubscription, which means having more threads than the available parallelism the hardware offers, or even more than the system can handle.

When using OS threading interfaces, such as POSIX threads, too much *actual parallelism* can be detrimental. These threads have mandatory semantics, which means they must run in parallel. So the OS must time-slice execution among these threads, incurring large overhead for context switching and reloading items into cache. Mandatory, or explicit, parallelism doesn't also support the idea of hierarchical decomposition of program modules that was discussed earlier, which makes writing large-scale parallel software that uses libraries and modules that can also be parallelized a real pain. That introduces the

need for a parallel programming model that offers the ability to express optional parallelism. Instead of using threads as our main parallelizing component, we use tasks.
A task is a piece of optional parallelism, which is implicitly scheduled on software threads. Scheduling software threads on hardware threads by the OS is usually preemptive; it can happen at any time. In contrast, scheduling tasks on software threads is non-preemptive; a thread only switches tasks at predictable switch points. Non-preemptive scheduling enables significantly lower overhead and stronger reasoning about space and time requirements than OS threads. Tasks are a more intuitive way of expressing parallelism in general and offer a significant parallel slack, which gives more flexibility to exploit machine resources. For example, having more potential parallelism that cores can help performance when the cores support multithreading. If, for instance, code must inevitably chase pointers using independent memory reads, additional parallelism can enable hardware-multithreading to hide the latency of the memory reads. Moreover, hierarchically decomposing software modules, composability and nested parallelism become non-issues, as they all end up just expressing more optional parallelism, which will be scheduled on the right number of software threads with care, without oversubscribing software threads to the system with the detrimental results discussed.

2.2 The TBB Library

In this thesis we examine the Intel Threading Building Blocks (TBB) multithreading library. Its programming model supports parallelism based on a tasking model. TBB is a library, not a language extension, and thus can be used with any compiler supporting ISO C++, so it is portable across platforms, operating systems and processors. It uses C++ features, such as function objects, to implement its syntax.

2.2.1 Overview of the library

TBB is written following the generic programming philosophy used by the C++ Standard Template Library (STL). It relies heavily on C++ templates to provide generic parallel programming patterns, such as parallel_for or parallel_reduce, with the fewest possible assumptions about data structures and data types that they will be used on.
An overview of the components of Intel TBB are presented in figure5. They include the following:

- Task: The most primitive and low level representation of a task, as parallel work abstraction. Tasks are chunks of work to be done, following the philosophy of potential parallelism. It is designed primarily for efficient execution, rather than convenience. It serves as a foundation for every tool for parallel computation that is offered by the library, and thus should impose minimal performance penalty. Task groups run an arbitrary number of tasks in parallel.
- Parallel Algorithms: These are higher level templates, which provide convenient interfaces for tasks, enabling the programmer to express parallelism using some popu-

lar algorithms and patterns, such as parallel_for, parallel_reduce, parallel_scan, work pile pattern, pipeline pattern, flow graph etc. As already mentioned, their foundation is tasks. This means that if, for instance, a parallel_for is invoked, the template will spawn the required number of tasks implicitly, chunking the work that was given according to specific requirements. These requirements are given in many cases by the blocked_range class and the different partitioners.

- Synchronization: the library includes primitive synchronization components such as atomic variables, mutexes etc., enabling the programmer to have full control of the program execution flow.
- Concurrent containers: popular and useful containers that are designed to be scalable and generic, following the philosophy of STL.
- Memory allocation: the library offers scalable memory allocators that offer cache alignment for false sharing avoidance and thread-local storage.
- Utility: cross-thread accurate timers

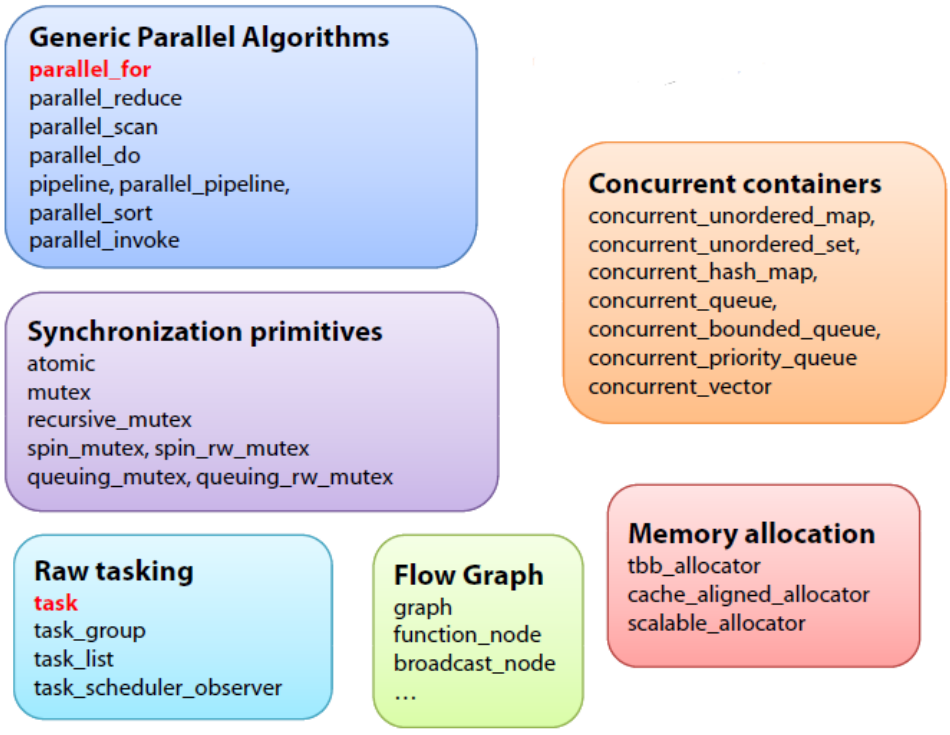

Figure 5. TBB Components

All the above functionalities are fully composable, not only with one another, but also with other popular parallel programming models such as OpenMP and MPI. TBB could be used to provide scalable parallelization at node level, and MPI to provide message passing style parallelism at system level. The abstract implementations on containers and algorithms that TBB offers boost productivity by enabling high code reuse while the non-preemptive scheduling of tasks enables better time and space overhead estimations.

2.2.2 How it satisfies the desired properties

The aforementioned must have made it clear that Intel TBB aims at high performance in shared memory architectures, provides a portable solution (it runs with any ISO C++ compiler) that can cooperate with other parallel programming models, boosts productivity by providing high code reuse with its generic components, which are fully composable.

2.3 TBB Scheduler

2.3.1 Overview, Basic Architectures and Components, Basic Functionalities

At the heart of the TBB runtime library exists the TBB task scheduler. This piece of code has the responsibility to schedule tasks on software threads in a non-preemptive manner. An overview of the basic components that constitute the TBB scheduler is presented in figure 6.

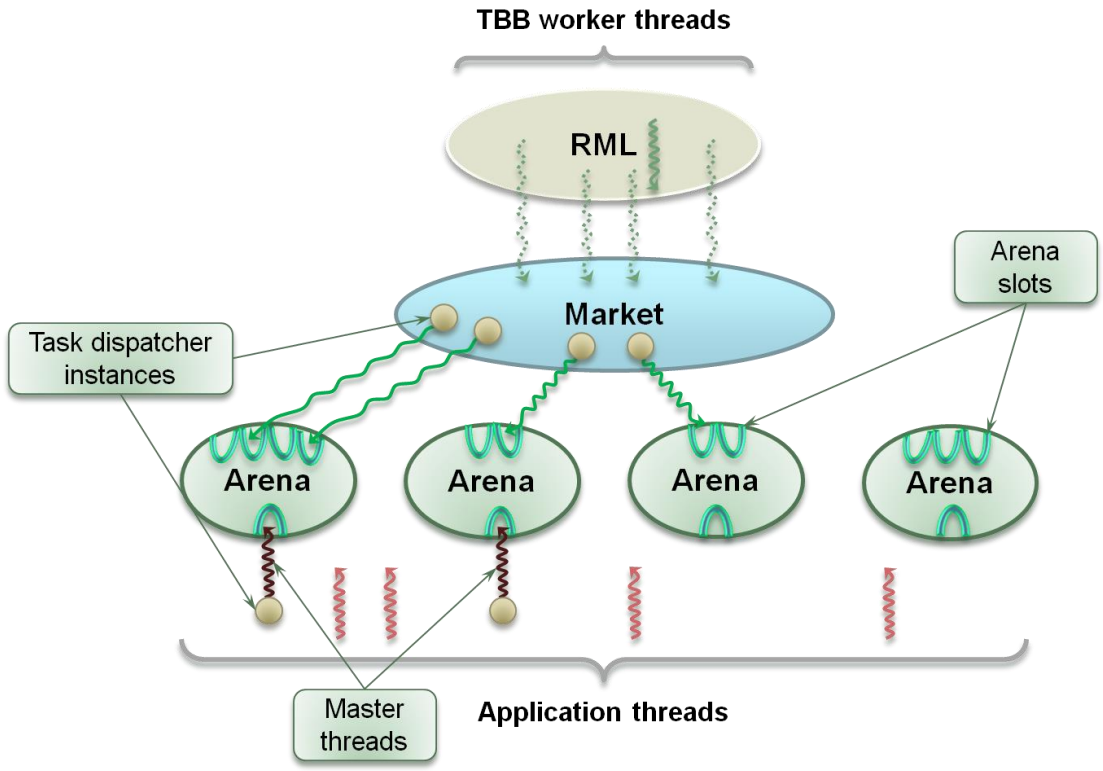

Figure 6. Scheduler Architecture Overview

When a thread creates for the first time an instance of the class *task_scheduler_init*, it is considered as the master thread and the following structures are initialized:

- RML (Resource Management Layer): this structure keeps the workers, which are OS threads in essence. At the initialization time no actual OS threads are created. The number of virtual workers that are created corresponds to the requested concurrency by the application. The concurrency offered by the TBB runtime is the largest between the number of threads that the first application that initial-

ized the scheduler requested and the concurrency offered by the underlying machine. This amount cannot be changed, unless the scheduler is destroyed and re-initialized requesting different concurrency. Actual OS threads are created lazily, for example when a parallel algorithm is invoked, up to the maximum number of virtual workers that RML was initialized with.

- Market: this structure has the responsibility to provide the arenas with workers. The maximum number of workers that can be offered to all master threads is defined by the concurrency that the first application that constructed the scheduler requested. If many master threads ask for workers, it is possible to exceed the limit of the virtual workers of the RML. In this case the market tries to dispatch a fraction of the total workers, according to the following rule:

$$RML\_workers * \frac{master\_demand}{total\_demand}$$

- Arena: this structure is associated with the master thread. Each master thread has its own arena, in which this thread along with the workers provided by the market work on the tasks. Each arena registers to the market. The slots of each arena correspond to the maximum number of workers that can be assigned by the market and can be used by the master thread to execute tasks. Their number is equal or less than the concurrency requested by the master thread that first initialized the task scheduler.
- Task dispatcher: this structure is the local scheduler of each worker that has the responsibility to acquire and execute tasks. Local instances of the task dispatcher, which corresponds to the class custom_scheduler, register to each arena slot. The functionality of this class is described below.

2.3.2 Executing Tasks - Work Stealing Mechanism & Load Balancing Algorithms

When a parallel algorithm is invoked from a master thread, for example a parallel_for, an initial root task is spawned at the local task pool of the master thread. The market provides the arena with the maximum number of workers it can and each one of them keeps his own local task pool, initially empty. Each task pool is a LIFO data structure, which means that the owner can only take for execution the task at the beginning of the task pool, and can spawn new tasks also at the beginning. The reason for that will become obvious later on.

TBBs use two load balancing techniques, *recursive splitting* and *work stealing*. Recursive splitting refers to the situation where a worker has to execute a body of code on a range of data, for example a parallel_for on an array of N elements. In this case the worker starts to recursively split the range to equal subranges, creating and spawning the corresponding tasks. This continues in a depth first manner, as in figure 7, until a range becomes less or equal to a specific grainsize. Then the worker calls the body code on the subrange, executing the work needed. The other workers who initially have an empty local task pool try to steal work from the workers that have tasks available. When a

worker (including the master) runs out of work, he tries to steal work from a victim worker, which is chosen randomly. The latter mechanism is called work stealing and it ensures that no worker will stay idle if there is still work available and no other worker executes it. If the chunk of work he stole needs further splitting, he continues the splitting the same way the master thread did it, creating subtasks that are available to be stolen by other idle workers. Although there is the factor of randomness in choosing the victim, it has been proven in practice that this simple mechanism is very efficient for load balancing, minimizing the idleness of the workers.

The critical point about work stealing is that when a worker tries to steal a victim, he always steals from the end of its task pool, in a first-in first-out manner (FIFO). The reason for that is that because of the depth-first recursive splitting that treats the task pool as a last-in first-out (LIFO), the beginning of each local task pool has the smaller chunks of work, while the end has the largest chunks. As a result, splitting in a depth-first manner while stealing in a breadth-first manner leads to the best possible load balance.

2.3.2.1 Partitioners and Grainsize

The partitioner is responsible for splitting a range to subranges. Each partitioner guarantees that the recursive splitting will continue until the chunks of work become greater than G/2 and smaller that G, where G is the grainsize. There are three kinds of partitioners, the *simple partitioner*, the *auto-partitioner* and the *affinity partitioner*:

- Simple Partitioner: In the case of the simple partitioner, the programmer explicitly defines the grainsize, with the default value being 1. The partitioning of a range r is simply continues until r.is_divisible() becomes false.
- Auto-partitioner: The grainsize is automatically estimated by a heuristic. The mechanism includes two values, V and K, which are both 4 in the current implementation. A variable n is initialized with the value P*K for the top level range, where P is the number of processors available. Each time the range is split, it gets half of the original n. If a range is stolen, its n is forced to be at least V. When n reaches 1 the range is not further split, although is_divisible() may still return true (controlled by the grainsize that was hinted to the partitioner). The intuition behind this mechanism is that the range should be split to a number of equal chunks that equals to the number of available processors, but there should be a little more splitting so that more than one tasks are pending for execution at each processor, to help better load balancing and compensate for the randomness of stealing. If the range is two-dimensional, it can be divisible along one or both axes. If divisible along both dimensions, the two-dimensional range chooses the split that yields pieces with an aspect ratio similar to aspect ratio of the grain size.
- Affinity partitioner: In certain applications, such as numerical relaxation and time-stepping marches for partial differential equations, cache affinity is crucial

for performance. The randomness of stealing causes loop algorithms to have poor affinity between successive sweeps over the same range. The affinity partitioner tries to tackle this problem, by hinting that the same or similar subranges are assigned to the same workers between successive sweeps, hoping that the data of each subrange will still be in the cache for the next iterations. This is though a hint, meaning that subranges may still be migrated to other threads to rebalance load. Moreover, it is not guaranteed that the thread that corresponds to a specific worker will not be migrated to a different physical core by the operating system. Nevertheless, the OS does not often migrate threads for no reason, and assuming that no other resource demanding process is running concurrently, the OS migration should not cause any performance issues. The grainsize in this case is selected as in the case of auto-partitioner.

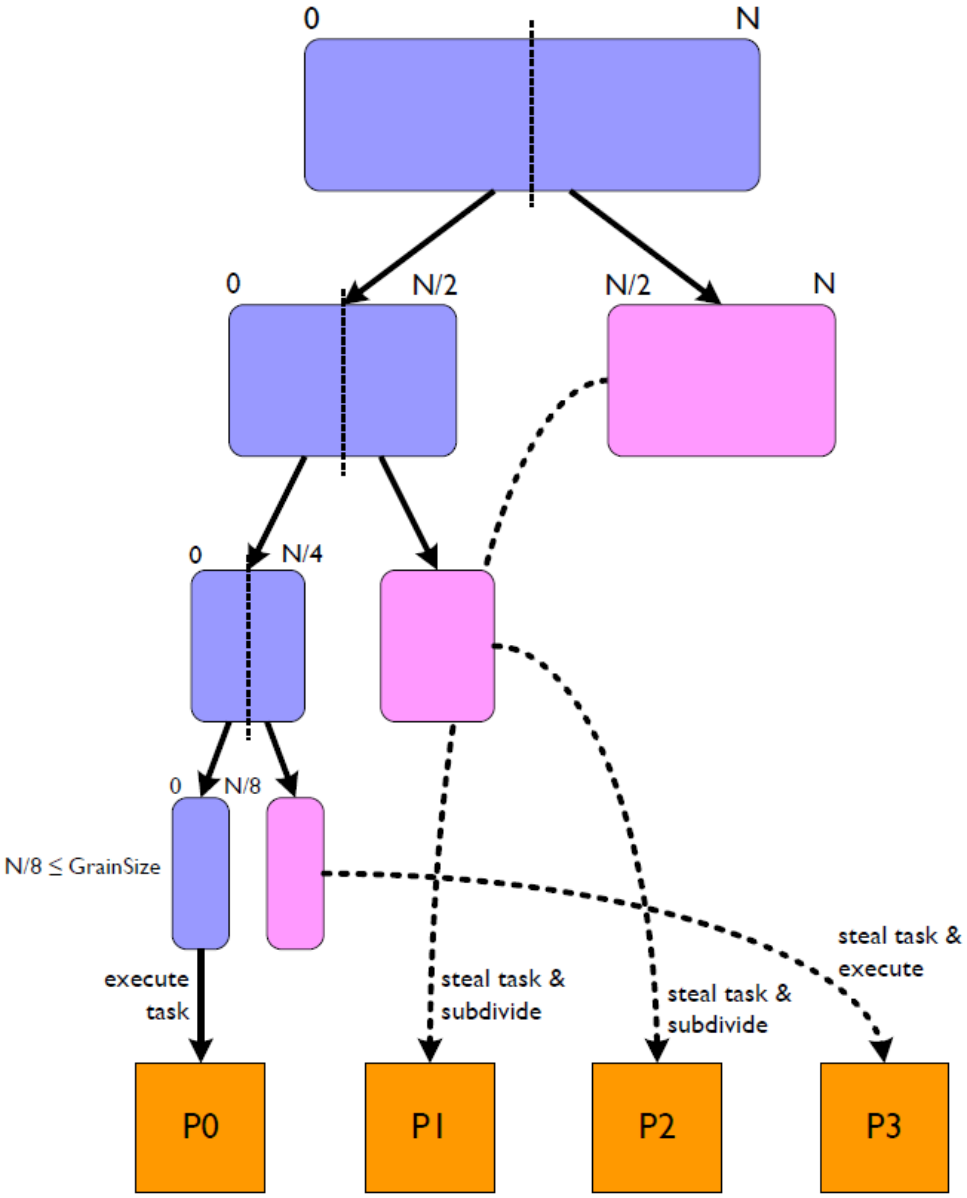

Figure 7. Recursive Splitting & Work Stealing 1

Figure8 shows two possible steps of a parallel_for computation on the range 0 to N. First the master thread has subdivided the range at his local task pool, putting the smallest chunks of work at the front. Meanwhile, workers 1 and 2 tried to steal work from master. Worker 1 stole first the rightmost chunk of work and subsequently Worker 2 stole the second rightmost chunk. Worker 3 has not still stolen anything. At the next step, the master thread still executes his smaller chunk of work, while Worker 1 and 2 have already splitted their stolen chunks a few times. Worker 3 now kicks in and tries to steal the rightmost chunk from Worker 1.

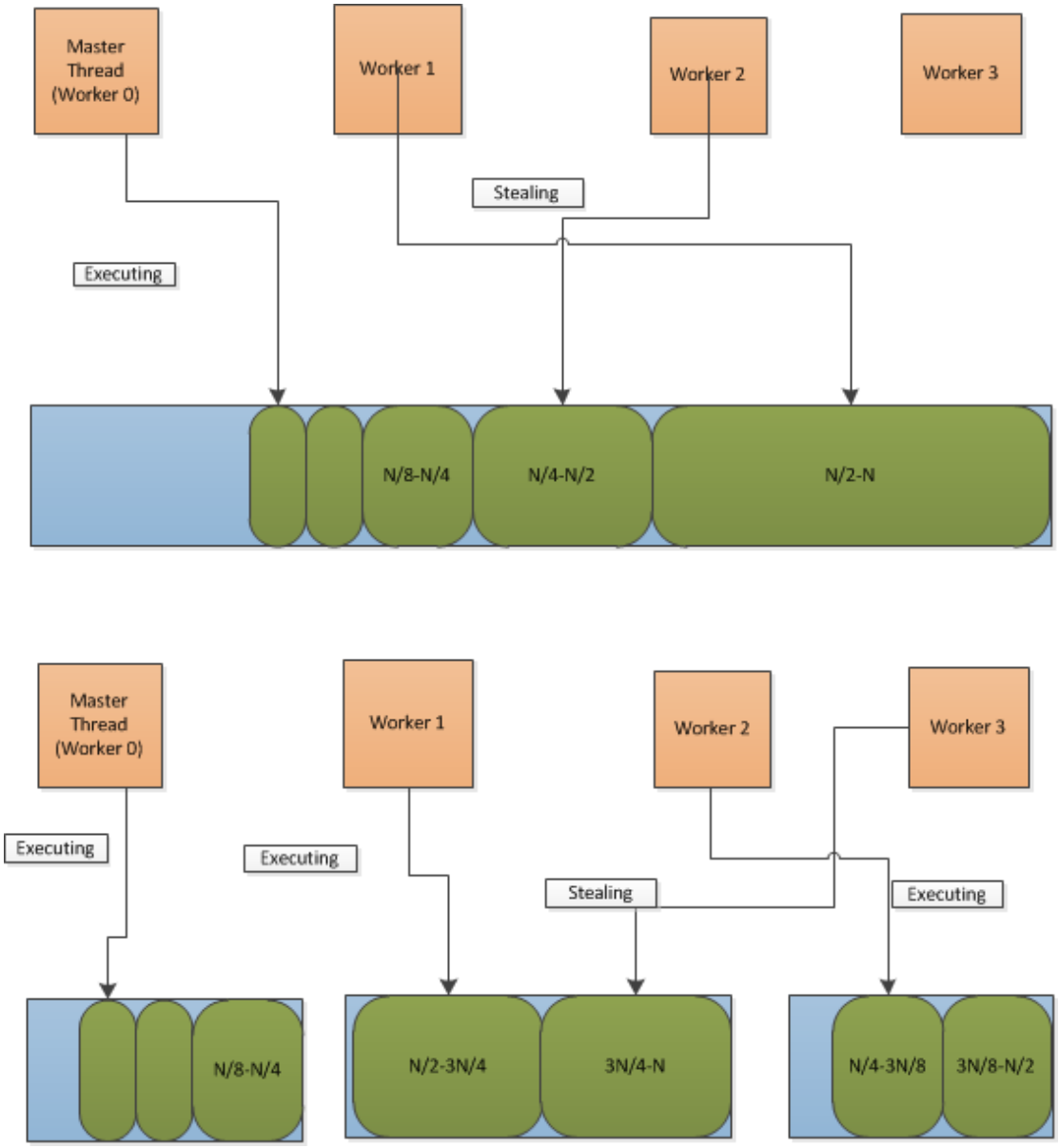

Figure 8. Recursive Splitting & Work Stealing 2

When the master thread decides to leave the arena, it destructs the task_scheduler_init object. Other workers though may still be working on tasks, so complete destruction of

components is postponed. When a worker repeatedly fails to steal work from others after finishing his own, he scans the arena to make sure he is left alone and that the other workers have already left, leaving no further work to be done. If that holds, he leaves the arena and destructs the relevant components.

2.3.3 Cache Coherence Protocols and Problems with Work Stealing

While work stealing is a simple and efficient mechanism to balance the load, several issues may arise during runtime.

2.3.3.1 Cache pollution

Stealing a task from a random victim implies that the thief must operate on data, such as a specific subrange of a loop, which would otherwise be handled by the victim. The randomness of stealing could pollute thief's cache with data that are completely irrelevant to the current data, and are far apart from each other in memory. This could cause serious performance degradation and several cache misses, compulsory as well as capacity misses, because the new data may evict data that will be needed in the future.

2.3.3.2 Data sharing

When a thief brings data from a victim into his cache memory, some of them may be adjacent to data in memory that are used by the victim. This leads to false-sharing between the two cores, which can cause a serious bottleneck in runtime, because of the cache coherence protocols that will start to invalidate data to each other, causing a ping-pong effect. That could be avoided if the two cores shared some cache level. The lower the better, but the randomness in stealing cannot guarantee it.

2.3.3.3 Locking

Work stealing includes some locking mechanisms, in case a clash between the thief and the victim occurs. Locking also occurs when multiple thieves try to steal from the same victim. In either cases, when the thread count increases, locking could become a bottleneck to the system because of the randomness of stealing. If the stealing mechanism limited the stealing between specific threads, this locking mechanism could scale better and fewer clashes would occur.

2.4 Profiling of basic functionalities – Characterization of overhead scalability

The first step of this study is to examine the overhead introduced by the TBB runtime library on parallel applications, broken down to basic functions as well as total user-library time. The profiling has multiple targets: first it aims to confirm the structure of the library and identify the functionalities into the code. Second, we try to measure the amount of stealing that occurs during runtime and how much it can affect the overall performance. Also we try to expose the scalability of each basic function of the library, including stealing, in order to identify possible bottlenecks of the task stealing mechanism and overall performance.

2.4.1 Systems Used

We use the Intel Threading Building Blocks 3.4 library (tbb40_20120408oss), which at the start of our study was the most up-to-date release available. Although most recent releases have taken place since then, the basic functionality of the task scheduler has not been changed, so these changes do not affect the outcome of our results. We compile TBB using GCC 4.6.3 and used the optimized "release" library.

For our profiling we used the systems described in the following subsections:

2.4.1.1 "Dunnington" SMP Platform

The first physical system used for the profiling is a 24-core Dunnington-based SMP with the following characteristics (shown in figure 9):

- 4 package(Intel(R) Xeon(R) X7460 @ 2.67GHz)
- 6 cores per package
- no Hyperthreading
- 32KB L1 cache per core
- 3MB L2 cache per 2 cores
- 16MB L3 cache per package (6 cores)
- 28.136 MB RAM

2.4.1.2 "Termi" NUMA Platform

The second physical system used is a 12-core Termi-based NUMA with the following characteristics (shown in figure 10):

- 2 Packages (Intel(R) Xeon(R) X5650 @ 2.67GHz)
- 6 cores per Package
- Hyperthreading (24 threads in total)
- 32KB L1 cache per core (2 threads)
- 256KB L2 cache per core (2 threads)
- 12MB L3 per package (6 cores, 12 threads)
- 48.295 MB RAM

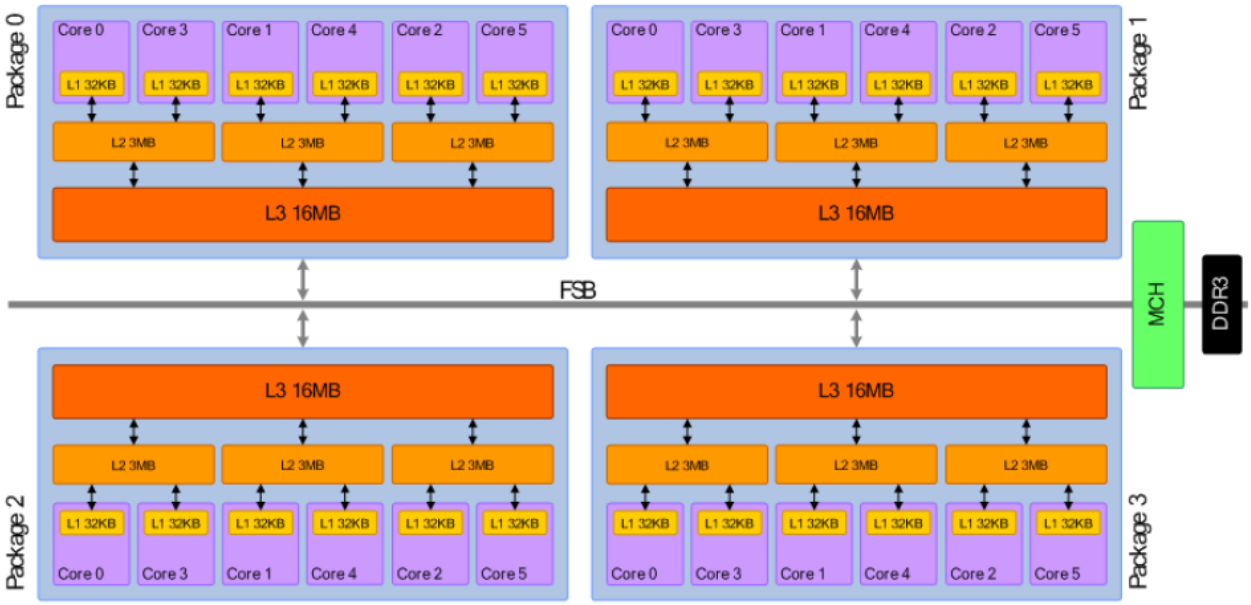

Figure 9. 24-core "Dunnington" SMP Platform

Figure 10. 12-core "Termi" NUMA Platform

2.4.2 TBB Scheduler Basics

The TBB scheduler consists mainly of a basic dispatch loop, a simplified version of which is presented below. The wait_for_all() procedure is the main scheduling loop. It consists of nested loops that attempt to obtain work through three different ways: explicit task specification, local task queue and random task stealing. In the innermost loop the task for execution is specified by the current task, which returns a pointer to the next task. If the current task does not return a task to execute, the do-while loop tries to acquire a task from the local task pool. If there is no work left in the local task pool, the thread tries to steal a task from a random victim. If that is unsuccessful, the thread waits for a fixed amount of time and tries again. If too many unsuccessful attempts occur, the thread gives up and waits until the main thread wakes it by generating more tasks.

```
wait_for_all(task *child) {
        task = child;
        loop until root is alive {
          do {
                while task available {
                        next_task = task->execute();
                        Decrease ref_count for parent of task
                        If ref_count == 0
                                next_task = parent of task
                }
                task = get_task();
          } while (task);
        task = steal_task(random());
        if steal unsuccessful {
          Wait for a fixed amount of time
          If waited for too long, wait for master thread to produce new work
        }
}
```

Listing 3. Basic task dispatch loop

2.4.3 Basic TBB Functionalities

The basic TBB Scheduler functionalities are:

- spawn: When a worker is created, it is associated with a local task pool, as already mentioned. Tasks are explicitly enqueued into a task pool when their corresponding worker calls the spawn method. This can happen many times when executing a task, as in the case of the Fibonacci computation, where each task (which corresponds to the computation of the n-th number) spawns two more tasks (which correspond to the computation of the (n-1)-th and (n-2)-th numbers). It takes a pointer to a task object and enqueues it to the local task pool of the calling thread.

- get_task: This method is called by the dispatcher loop when the completion of the previous task returns no task for execution. It tries to retrieve a task from the local task pool. When unsuccessful, it returns NULL.
- receive_or_steal_task: This method is called when there is no work left in the local task pool. Except for stealing tasks from other threads, it may also retrieve tasks that have been mailed to it. It can be subdivided into tree functions:
    - lock_task_pool: It basically tries to lock the victim's local task pool so that no collision will happen with the owner. Increasing number of threads may cause several collisions and as any locking mechanism, it can have scalability issues.
    - stealing: It is the act of retrieving a task class description from the victim's task pool, after locking it.
    - steal_wait: It is the time spent on functions other than actual stealing inside the receive_or_steal_task, which is essentially the time waiting between successive attempts to steal.
- acquire_queue: This method is called by spawn and get_task to lock the thread's local task pool, in order to either enqueue a new task or retrieve a task ready for execution.
- lib_wait: this value represents the time spent by the wait_for_all dispatch loop on functions other than the above. It is essentially the difference between the total library timer (the wait_for_all loop) and the sum of the individual timers of each aforementioned function (except for acquire_queue which is included in spawn and get_task timers).

2.4.4 Applications used for characterization

We study the impact of the TBB runtime library on parallel applications by using some well-known benchmarks: blackscholes, fluidanimate, streamcluster and swaptions, which are part of PARSEC benchmark suite[5], and convex_hull, matrix_multiply, quicksort, strassen, which are in-house developed benchmarks. These applications were chosen because they create a substantial amount of tasks and use many of the templated algorithms offered by the library, as well as the low-level way of task creation by hand, enabling us to examine the behavior and scalability of TBB's basic functionalities in a wide range of circumstances.

Algorithm	Description	Input size
Blackscholes	Option pricing with Black-Scholes Partial Differential Equation	64K
Fluidanimate	Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method	500K
Streamcluster	Online clustering of an input stream	16384 data points
Swaptions	Pricing of a portfolio of swaptions	64 swaptions, 20000 simulations
Convex Hull	Smallest convex that contains a set of points	40M points

Matrix_multiply	Matrix multiplication	1500x1500
Quicksort	Quicksort sorting method	100M
Strassen	Matrix multiplication	256x256

In the following subsections we present the results acquired during the profiling. For each application there is a brief description followed by 1) the speedup achieved, 2) a User-Library time breakdown for each thread count, 3) a breakdown diagram of the library time to the basic functionalities of the scheduler, 4) the scalability of each functionality separately. We present the most notable and characteristic results. The rest can be found in Appendix A.

2.4.4.1 Blackscholes

First, the speedup of the parallel area is given. We see that on the SMP machine the scalability is not as good as on the NUMA, a fact that we relate to the potential bottleneck on the memory bus in combination with the increased overhead in case of inter-socket communication. The following graphs give a clearer image of the overheads of each library component.

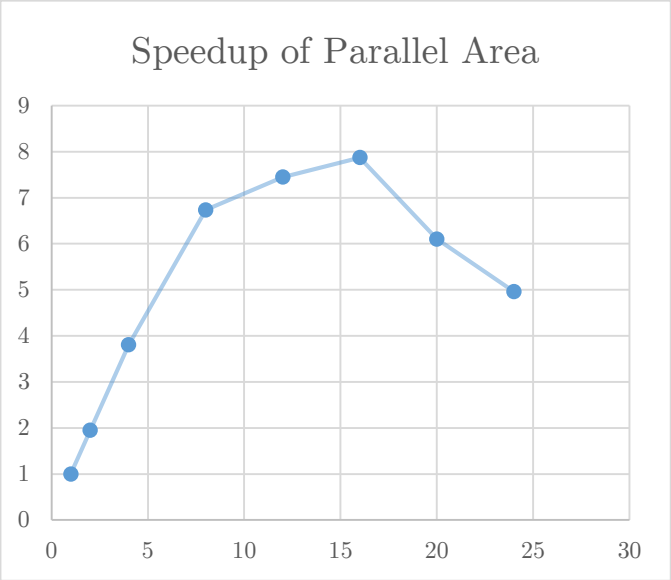

Figure 11. Blackscholes speedup on SMP

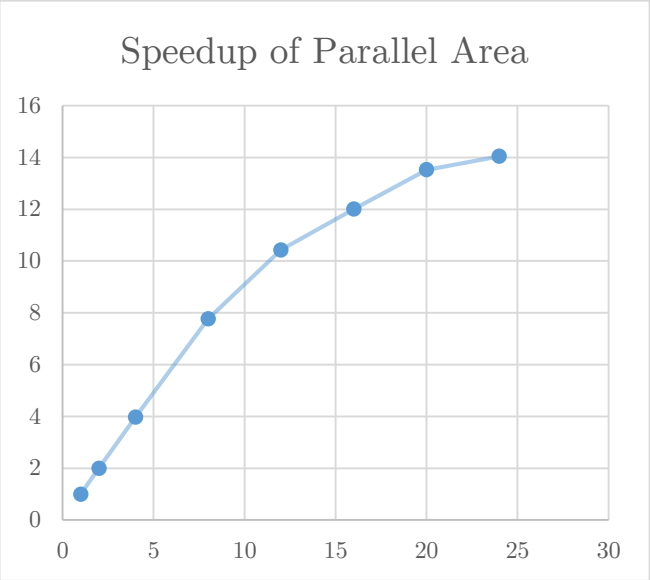

Figure 12. Blackscholes speedup on NUMA

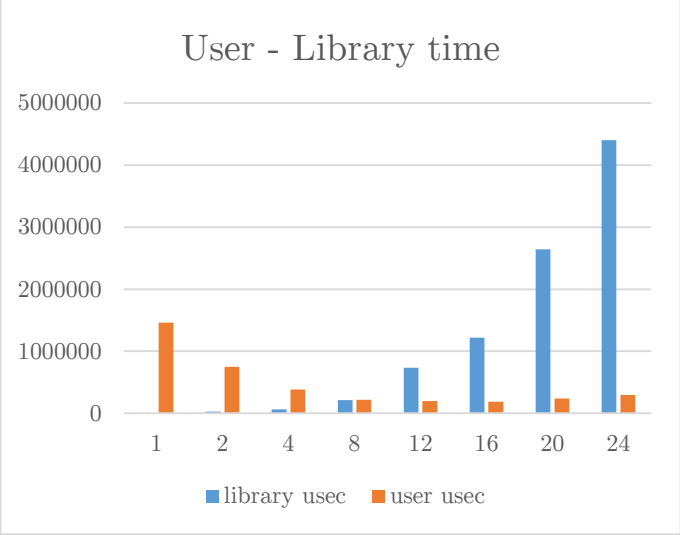

Figure 13. Blackscholes User-Library time on SMP for each thread count

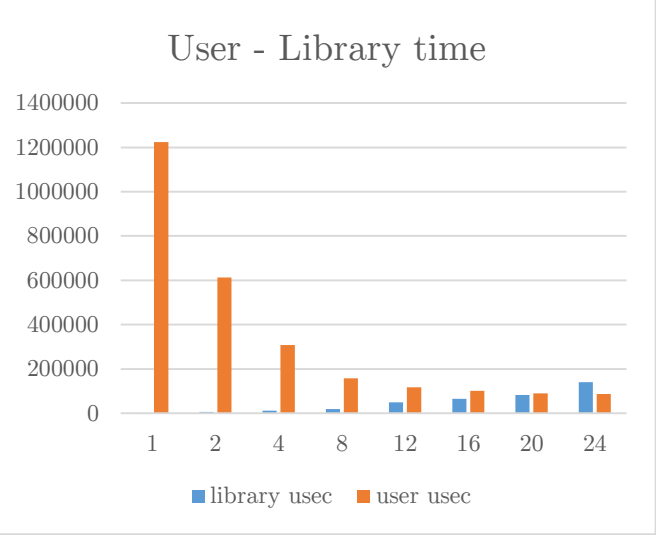

Figure 14. Blackscholes User-Library time on NUMA for each thread count

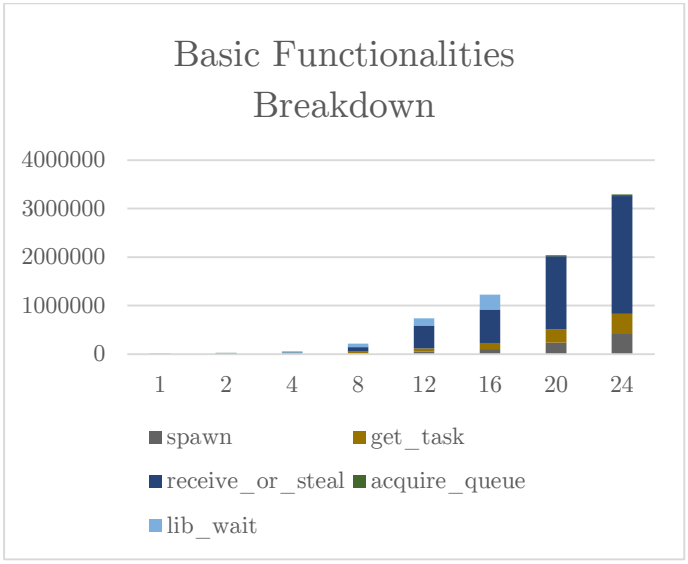

Figure 15. Blackscholes basic functionalities breakdown on SMP for each thread count

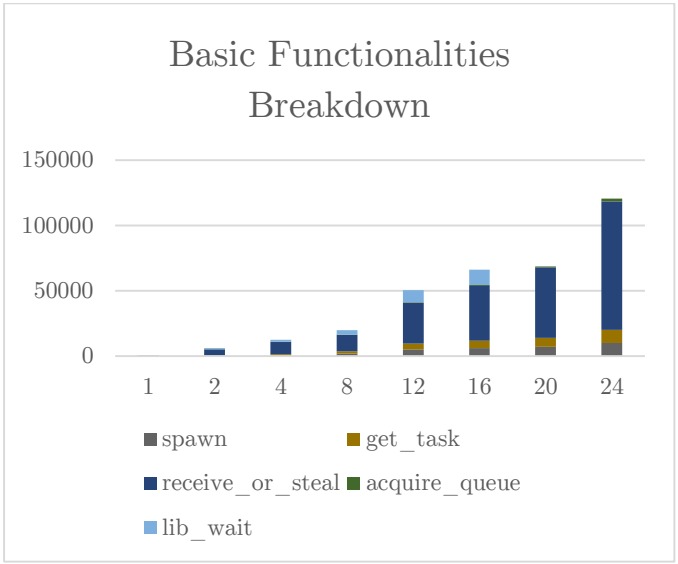

Figure 16. Blackscholes basic functionalities breakdown on NUMA for each thread count

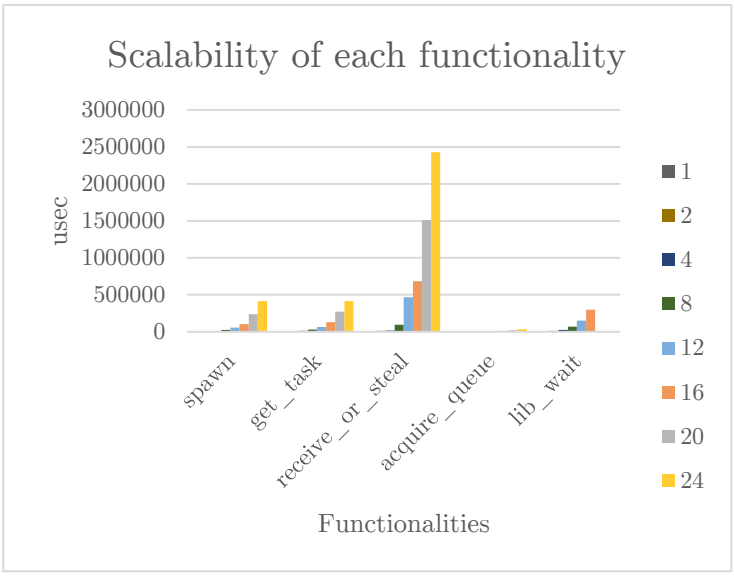

Figure 17. Blakscholes basic functionalities' scalability on SMP for each thread count

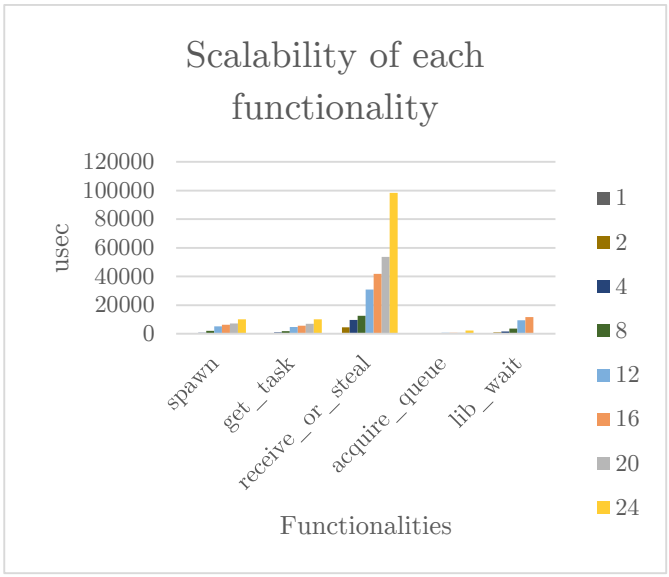

Figure 18. Blackscholes basic functionalities' scalability on NUMA for each thread count

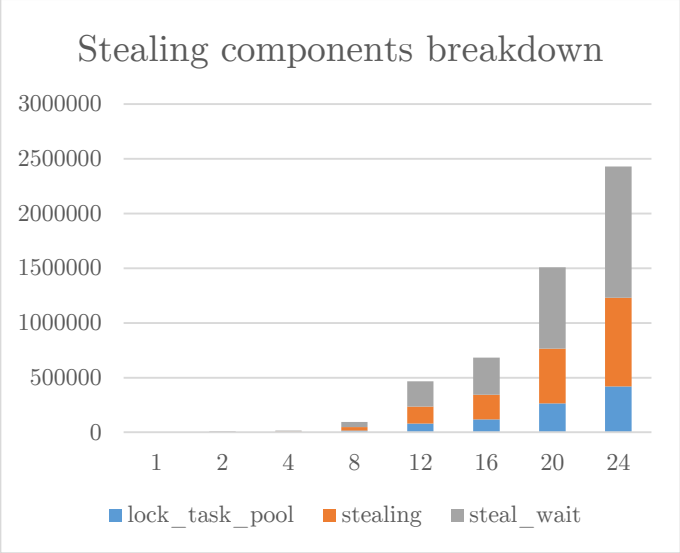

Figure 19. Blackscholes stealing components breakdown on SMP for each thread count

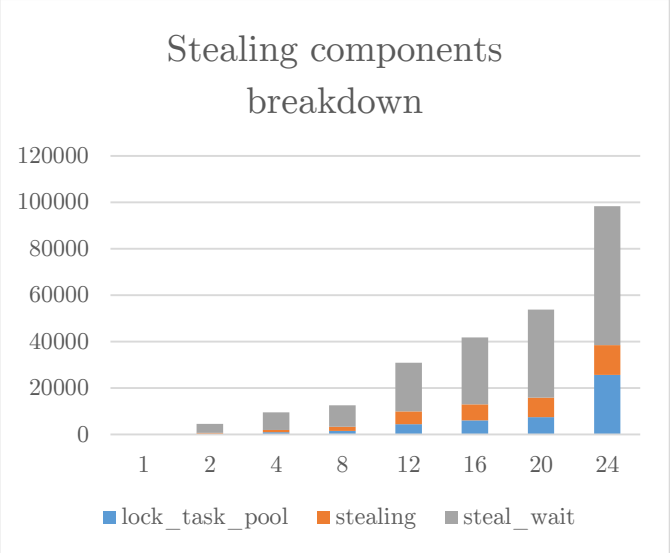

Figure 20. Blackscholes stealing components breakdown on NUMA for each thread count

Figure 21. Blackocholes scalability of stealing components on SMP for each thread count

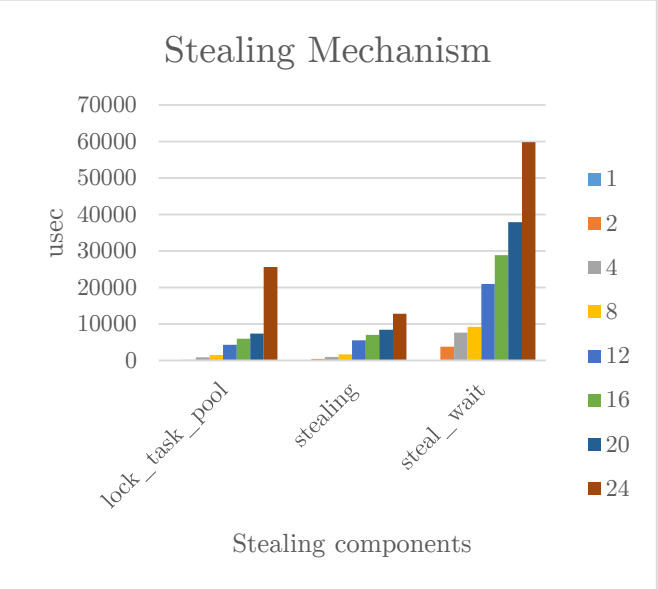

Figure 22. Blackscholes scalability of stealing components on NUMA for each thread count

2.4.4.2 *Fluidanimate*

The same graphs are presented for the *fluidanimate* application. This application does not scale very well and the library time is dominated by the stealing function.

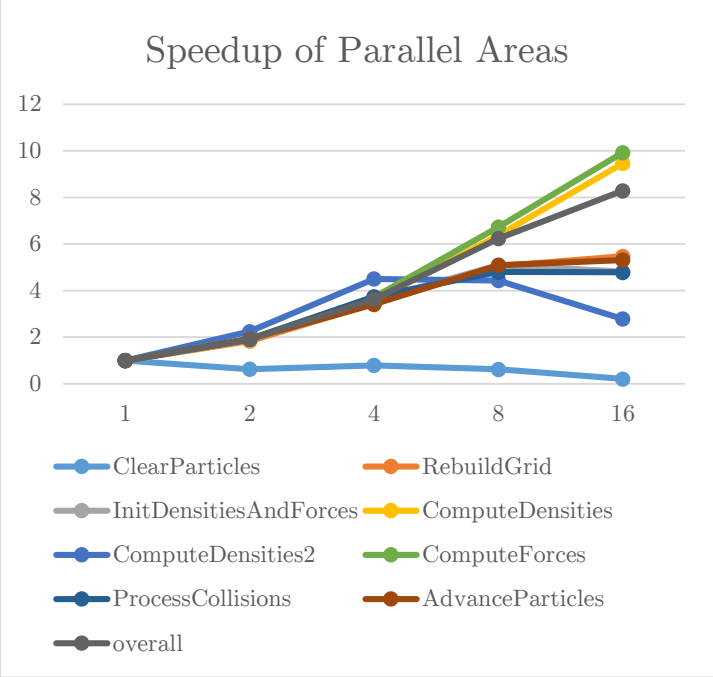

Figure 23. Fluidanimate speedup on SMP

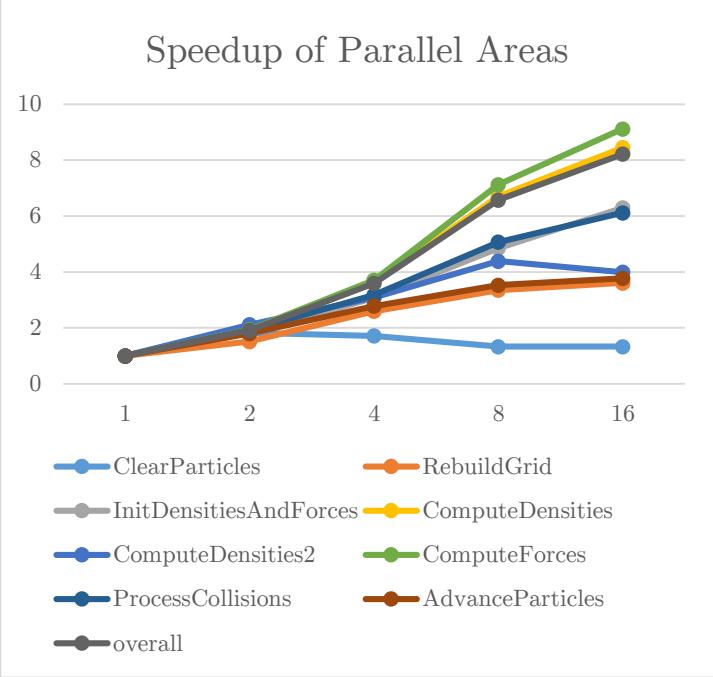

Figure 24. Fluidanimate speedup on NUMA

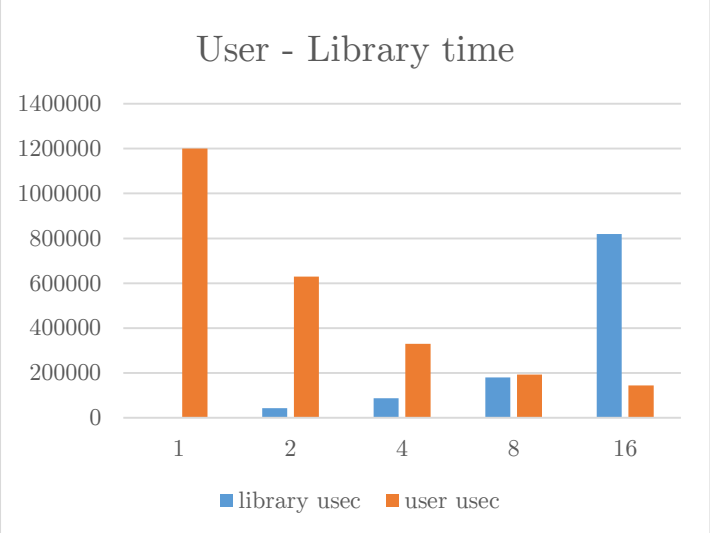

Figure 25. FluidanimateUser-Library time on SMP for each thread count

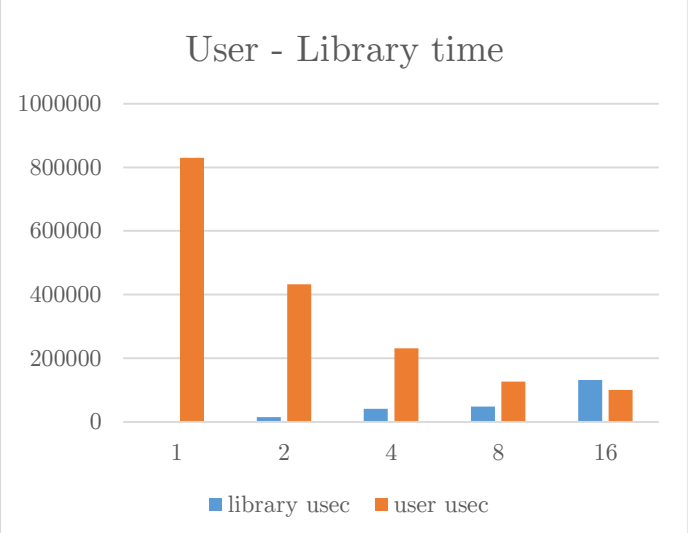

Figure 26. FluidanimateUser-Library time on NUMA for each thread count

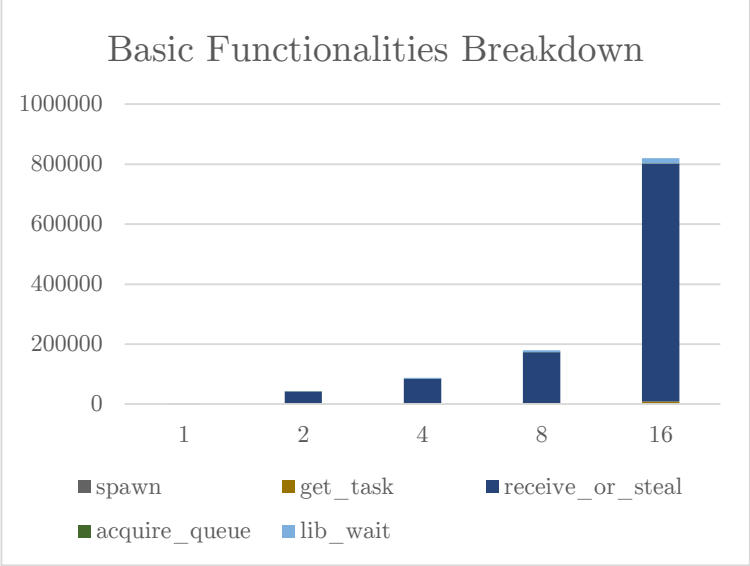

Figure 27. Fluidanimate basic functionalities breakdown on SMP for each thread count

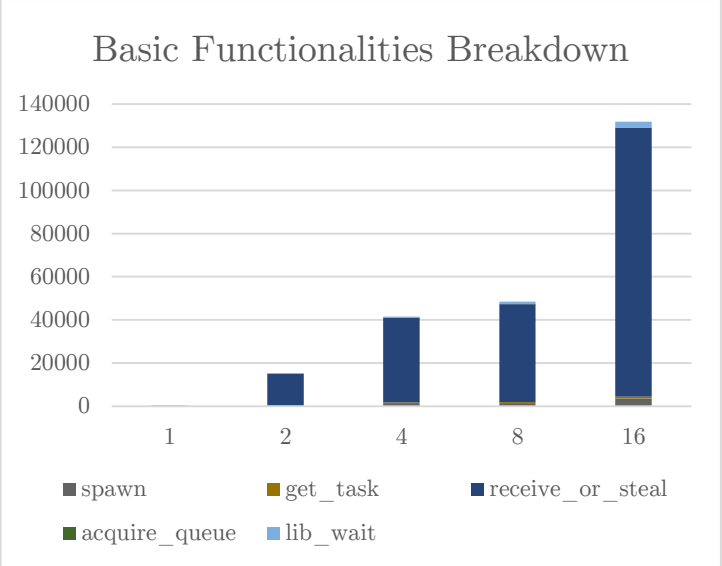

Figure 28. Fluidanimate basic functionalities breakdown on NUMA for each thread count

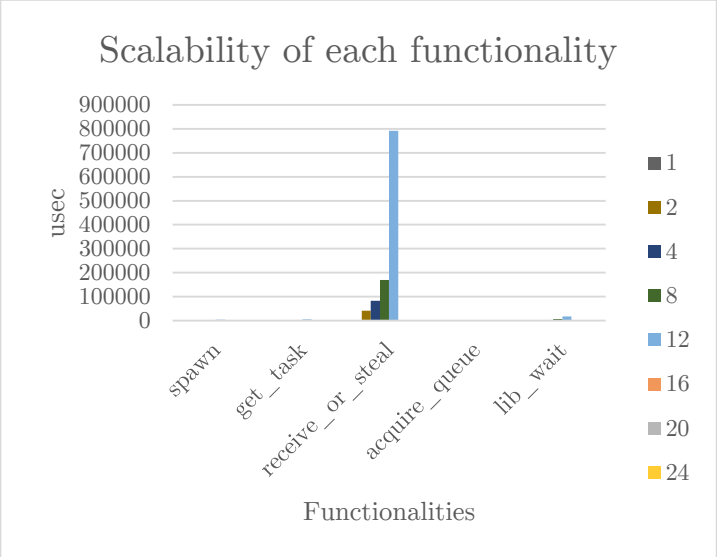

Figure 29. Fluidanimate basic functionalities' scalability on SMP for each thread count

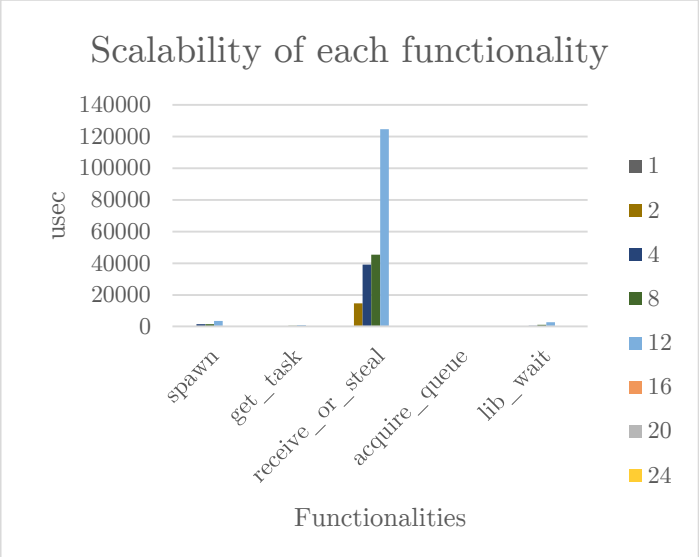

Figure 30. Fluidanimate basic functionalities' scalability on NUMA for each thread count

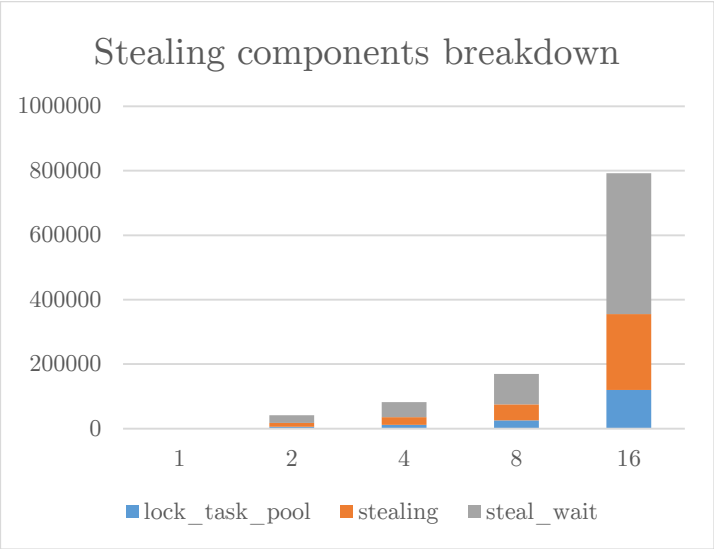

Figure 31. Fluidanimate stealing components breakdown on SMP for each thread count

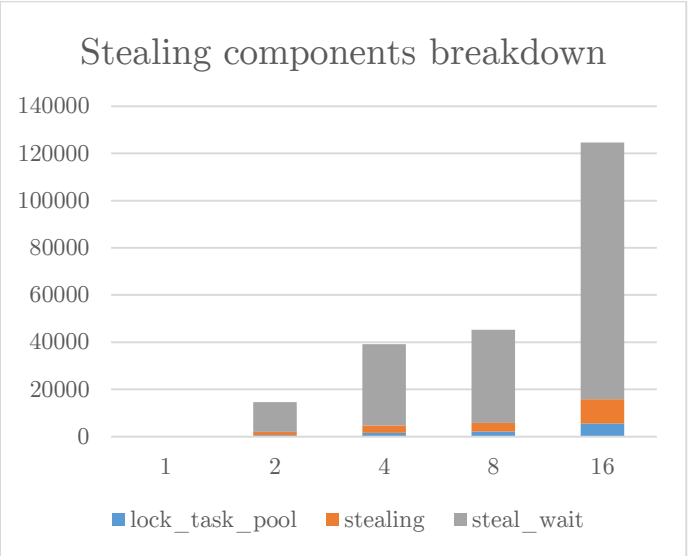

Figure 32. Fluidanimate stealing components breakdown on NUMA for each thread count

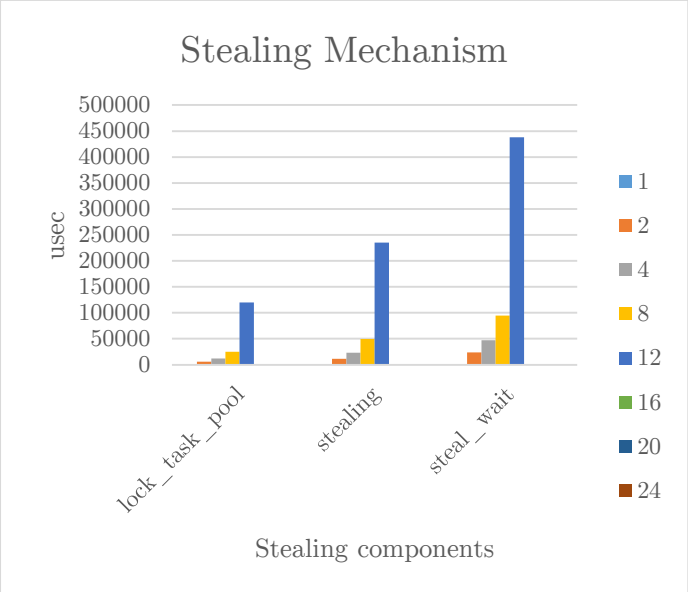

Figure 33. Fluidanimate scalability of stealing components on SMP for each thread count

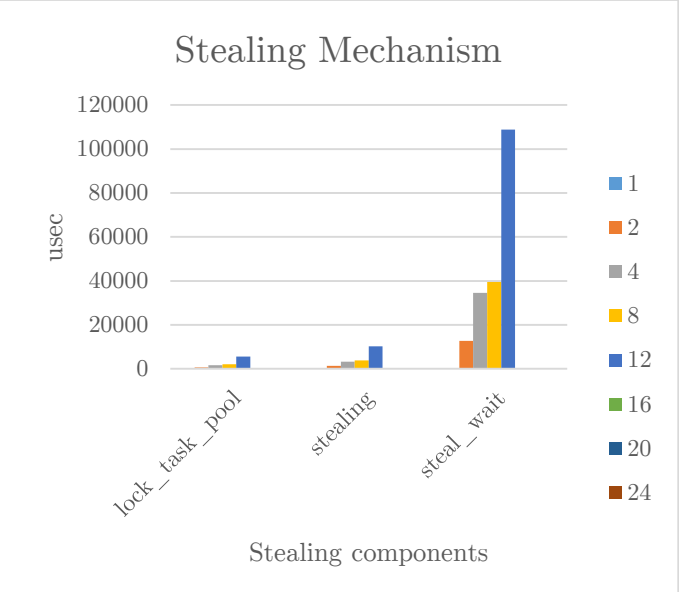

Figure 34. Fluidanimate scalability of stealing components on NUMA for each thread count

2.4.4.3 *Strassen*

The same graphs are presented for the *strassen* application. This application scales poorly on both the SMP and the NUMA platform. Work stealing also dominates in this case the library time and the stealing function does not scale well, as we see increasing execution times as the thread count increases.

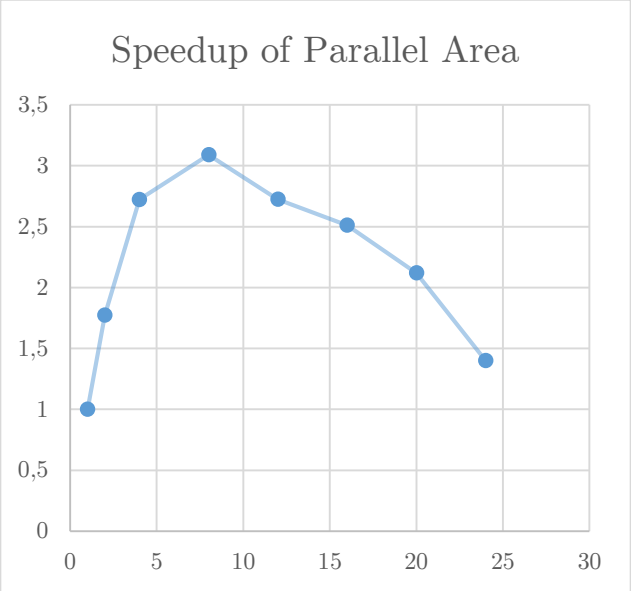

Figure 35. Strassen speedup on SMP

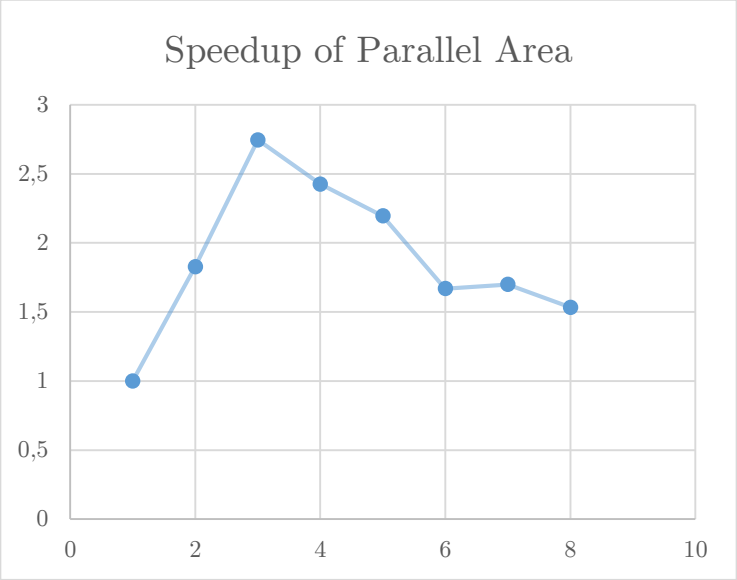

Figure 36. Strassen speedup on NUMA

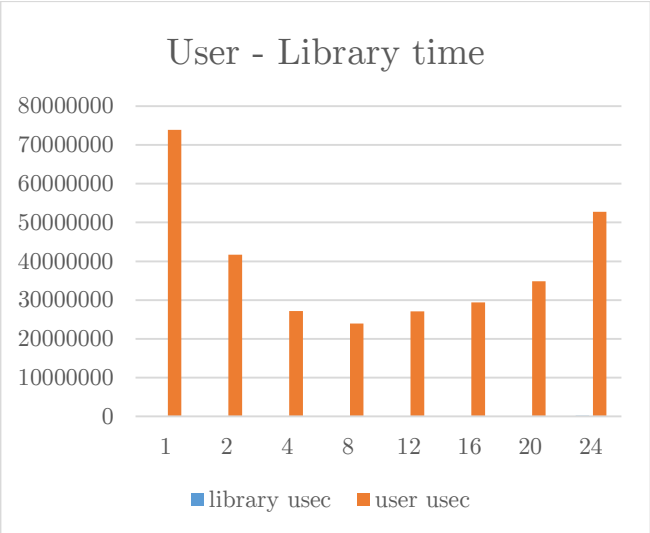

Figure 37. StrassenUser-Library time on SMP for each thread count

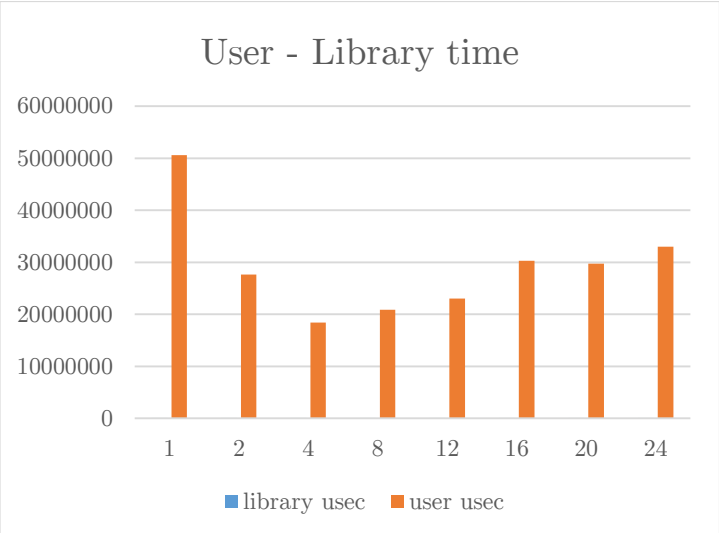

Figure 38. Strassen User-Library time on NUMA for each thread count

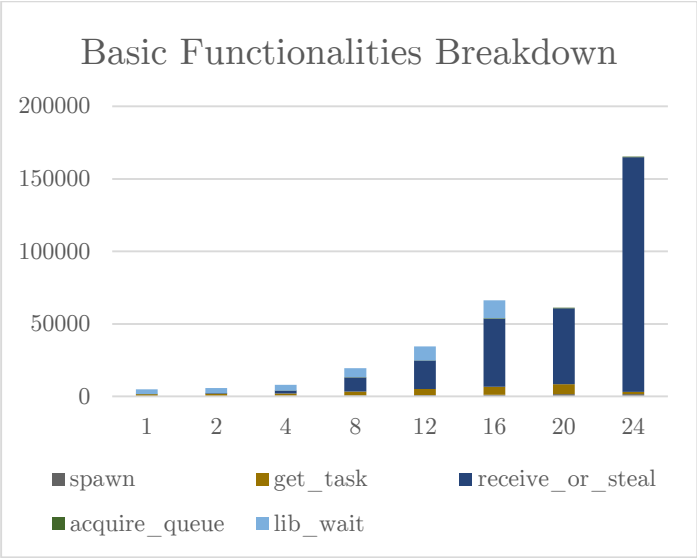

Figure 39. Strassen basic functionalities breakdown on SMP for each thread count

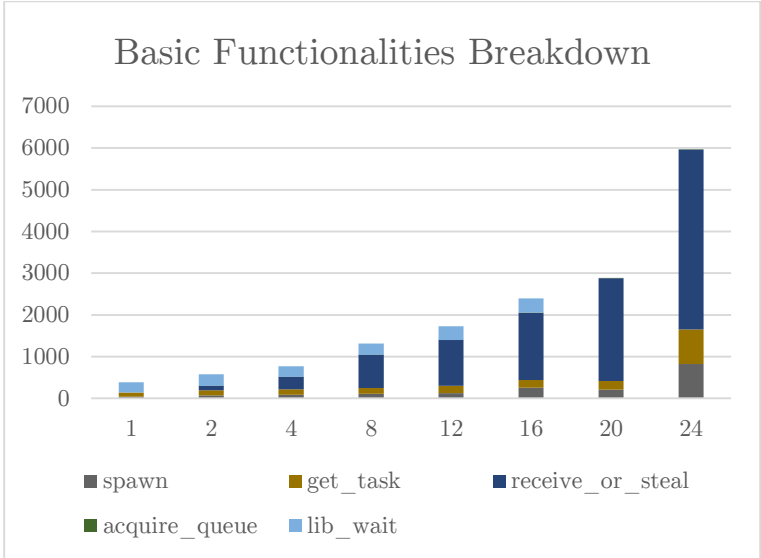

Figure 40. Strassen basic functionalities breakdown on NUMA for each thread count

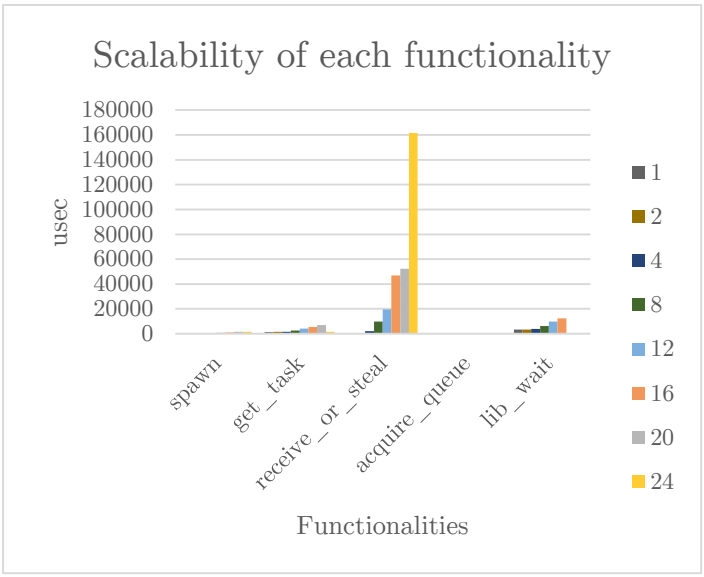

Figure 41. Strassen basic functionalities' scalability on SMP for each thread count

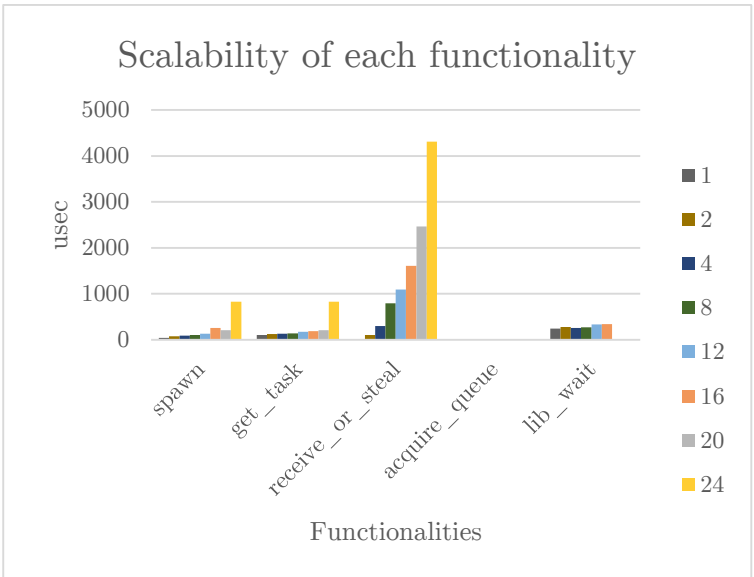

Figure 42. Strassen basic functionalities' scalability on NUMA for each thread count

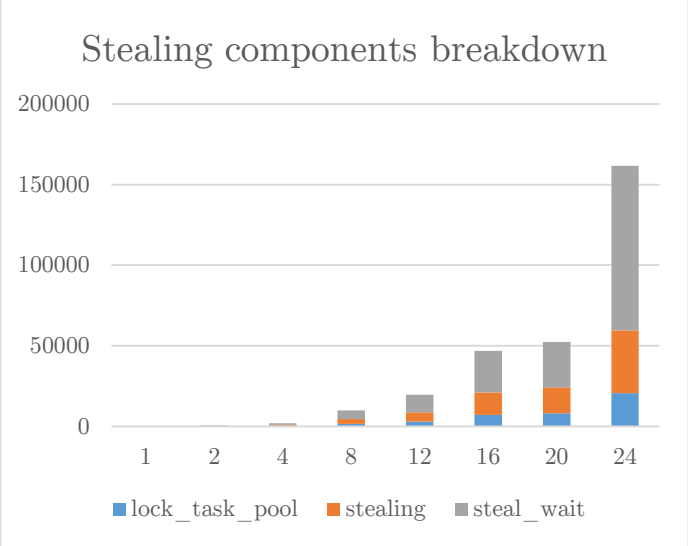

Figure 43. Strassen stealing components breakdown on SMP for each thread count

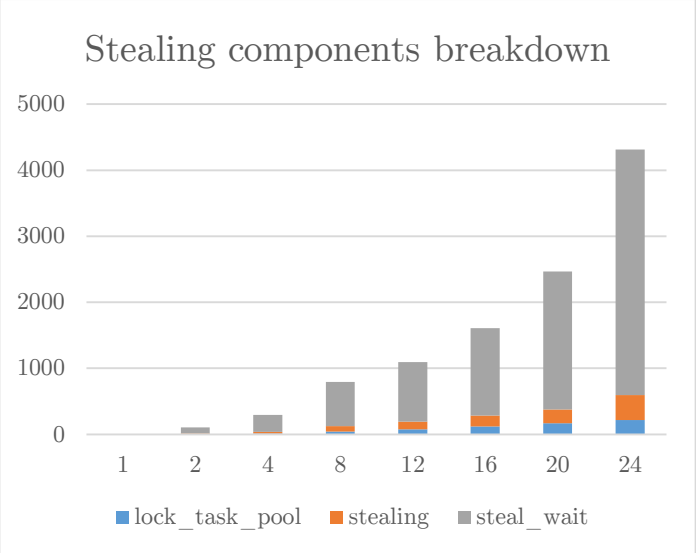

Figure 44. Strassen stealing components breakdown on NUMA for each thread count

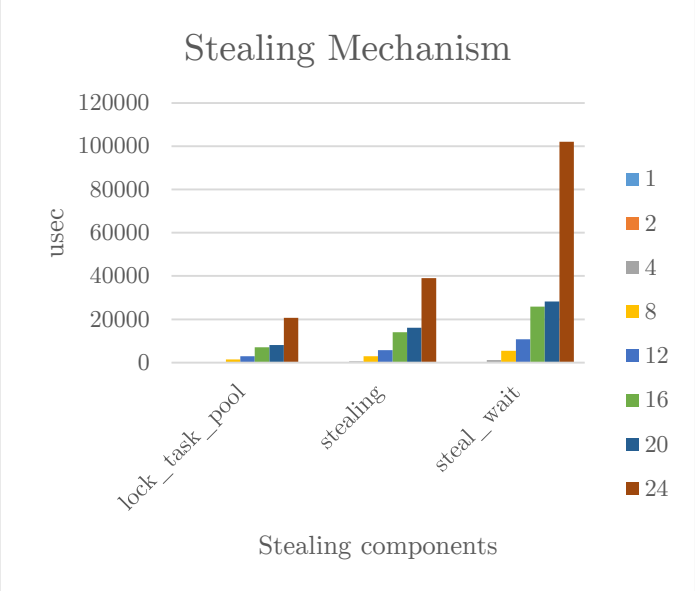

Figure 45. Strassen scalability of stealing components on SMP for each thread count

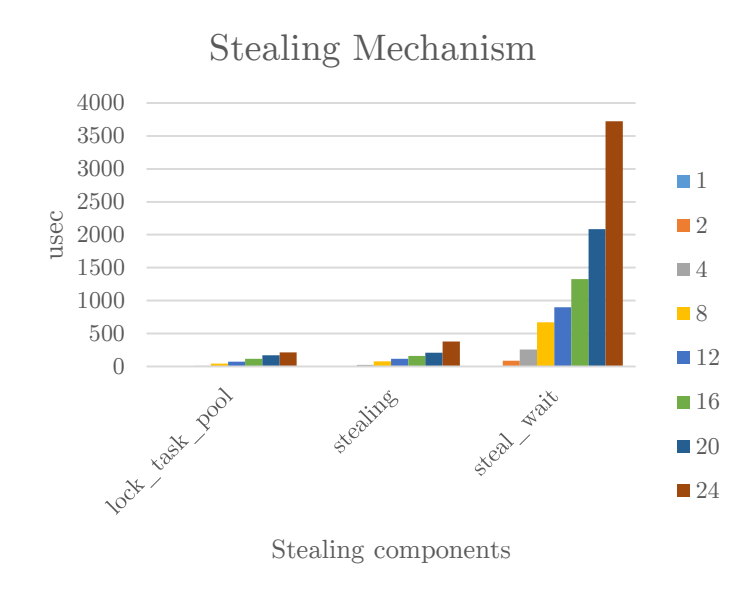

Figure 46. Strassen scalability of stealing components on NUMA for each thread count

2.4.4.4 Streamcluster

The graphs for the *streamcluster* application show another example of poor scalability on the SMP, with the NUMA having a better scalability, though not satisfying. Stealing time also dominates here the library run time.

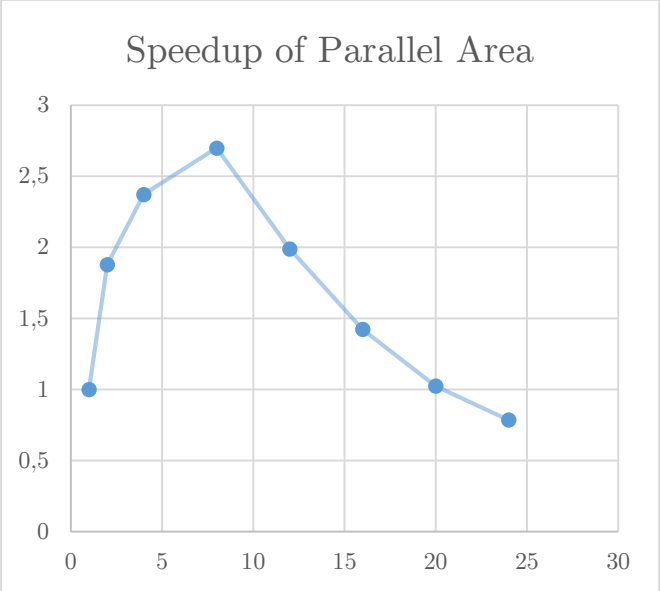

Figure 47. Streamcluster speedup on SMP

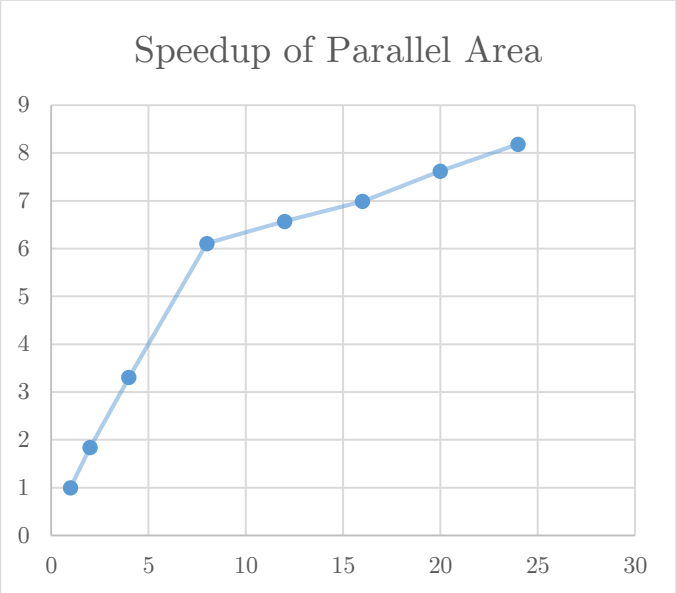

Figure 48. Streamcluster speedup on NUMA

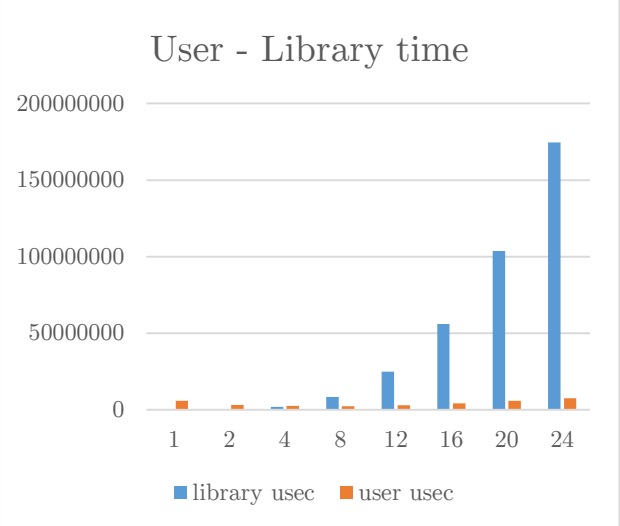

Figure 49. Streamcluster User-Library time on SMP for each thread count

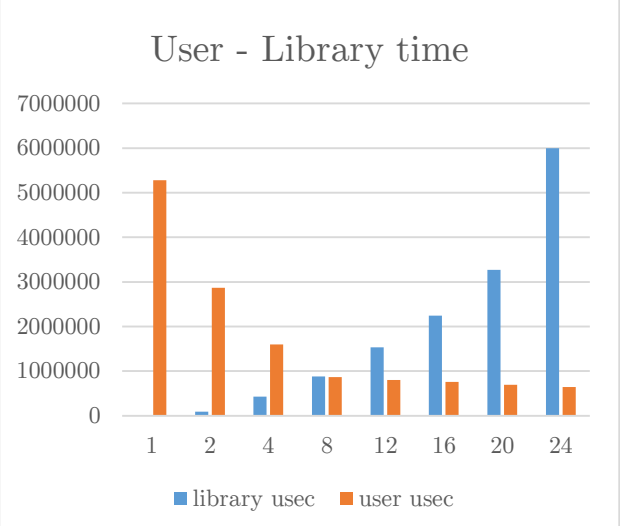

Figure 50. Streamcluster User-Library time on NUMA for each thread count

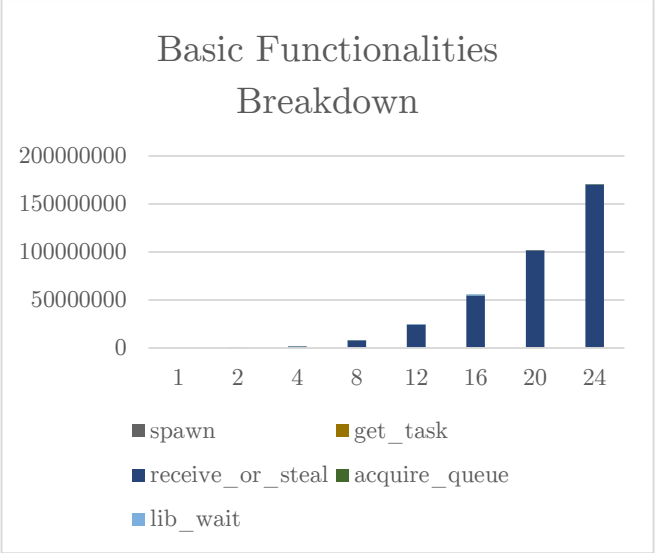

Figure 51. Streamcluster basic functionalities breakdown on SMP for each thread count

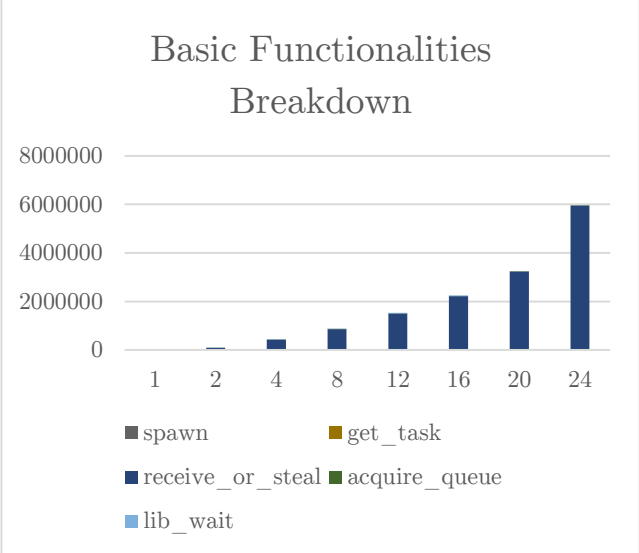

Figure 52. Streamcluster basic functionalities breakdown on NUMA for each thread count

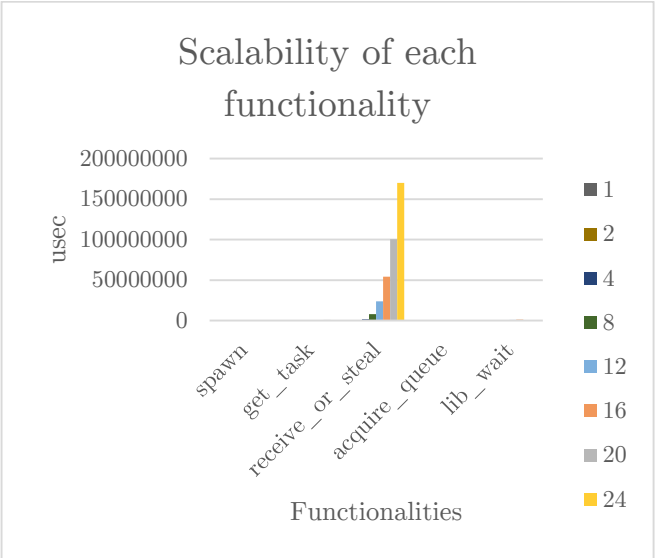

Figure 53. Streamcluster basic functionalities' scalability on SMP for each thread count

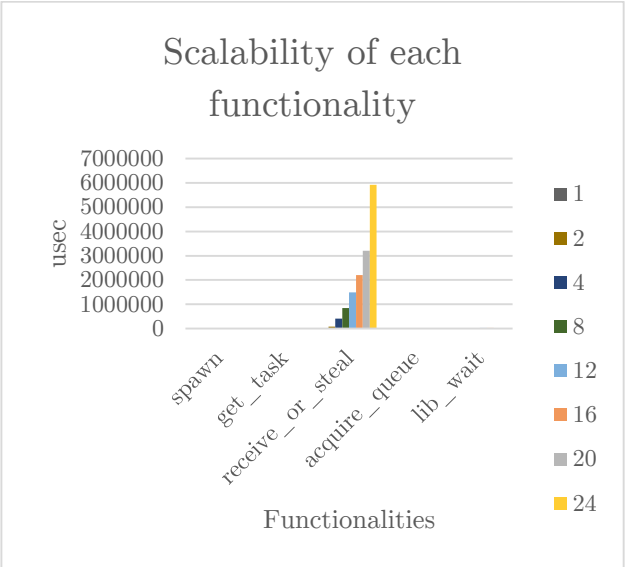

Figure 54. Streamcluster basic functionalities' scalability on NUMA for each thread count

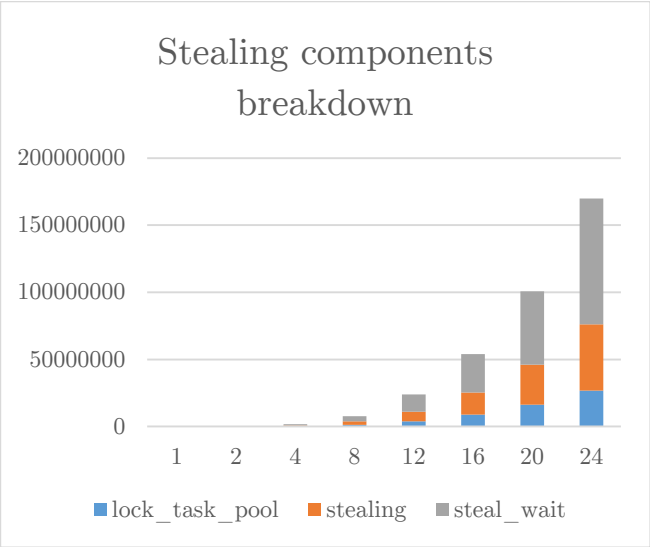

Figure 55. Streamcluster stealing components breakdown on SMP for each thread count

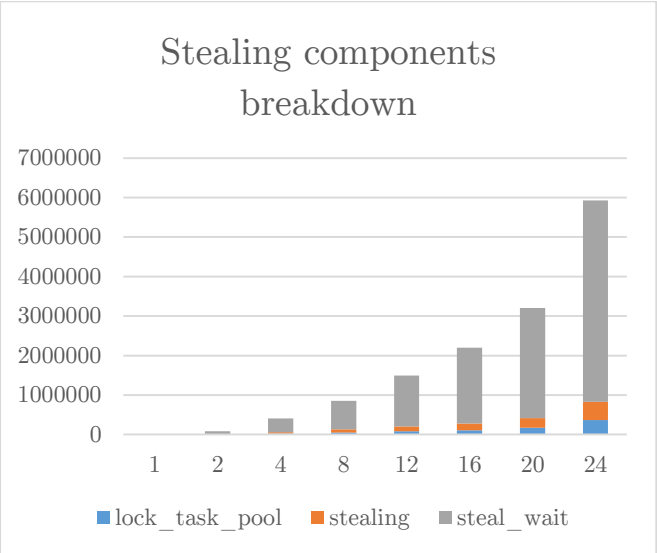

Figure 56. Streamcluster stealing components breakdown on NUMA for each thread count

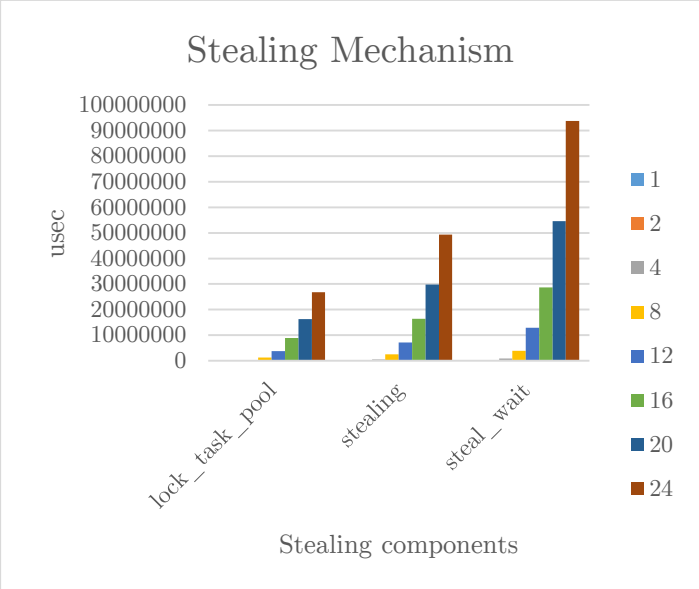

Figure 57. Streamcluster scalability of stealing components on SMP for each thread count

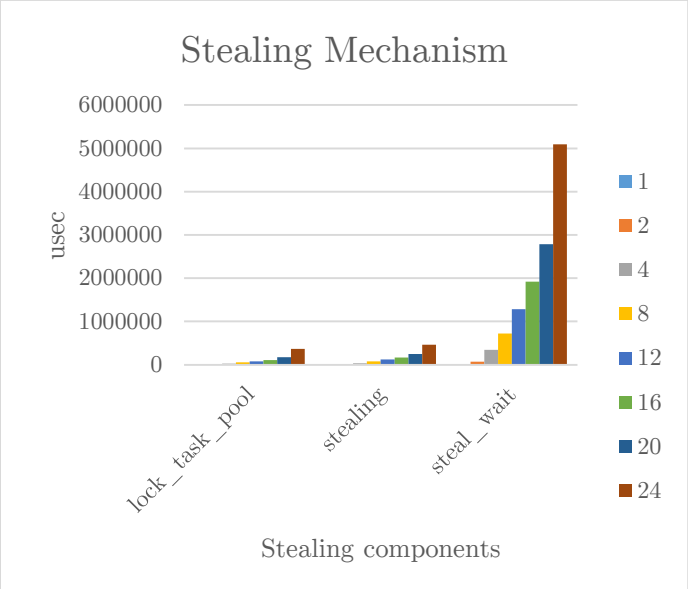

Figure 58. Streamcluster scalability of stealing components on NUMA for each thread count

Observations:

- In many cases, applications scaled better on the NUMA platform. The main reason for that is that the SMP has a single memory bus, making it a bottleneck for any memory transactions, and as the thread count increases more conflicts for the bus occur, serializing transactions that could otherwise be executed in parallel.
- The NUMA platform offers different paths to memory, parallelizing memory accesses of different packages, when of course it is inherently possible by the access pattern of the application.
- The most important observation on all applications is that stealing time dominates the library run time, making it a hot target for optimization.
- In many cases the stealing time is dominated by the *steal_wait* function, which means that the stealing attempts are mostly unsuccessful, a fact that is confirmed by the library statistics and the successful to failed stealing attempts ratio, although it is not presented in this thesis explicitly. According to the graphs, this fact is even worse on the NUMA platform.

Chapter 3
Techniques Used

3.1 Optimization targets

In this section we present the techniques that we applied as an attempt to better exploit the cache hierarchy of the physical system as well as attempts to improve load balancing. Our target for optimization is the stealing mechanism of TBBs. We applied several techniques and variations of them, exposing their strengths and weaknesses on each machine.

3.2 Stealing from the nearest neighbor

The first attempt for optimization of the stealing mechanism targets the cache hierarchy of the underlying physical machine. As explained earlier, work stealing occurs in an architecture-agnostic manner, by choosing random cores as potential victims. This can deteriorate the performance, especially on NUMA platforms that inter-socket communication incurs great overhead.

3.2.1 Technique description

Our approach was choosing the nearest possible core, in terms of cache distance, that has work to offer as victim. As it can be seen in the relevant figures describing the architectural organization of the platforms, each core shares several cache memory levels with different sets of cores.

The key idea is to try stealing from the cores with which we share the L1 cache level. Normally, that is zero or one cores, so in case we fail to steal from him or he doesn't have any tasks enqueued in his task queue we need to have alternatives. In that case we try to steal from the (other) cores that we share the L2 cache level with. And if that fails too, we try the L3 level. In case there is no work in our package, we could end up stealing work from the cores in the other packages, as a last resort. In that way we prioritize our targets in terms of cache level distance.

Another critical detail is the persistency to our first choices of victims. The nearest neighbors, for example the ones we share the L1 cache with, are few in our physical systems. They can either be one or two. In case they are not available for stealing when we probe them, we needed to persist and try again some times before we end up choosing the next nearest neighbor. The most neighbors are located in the other packages, some share the L3 cache level with our cpu, few the L2 and even fewer the L1 cache level.

This approach aims to stealing tasks that have a better chance to find the data they need in a cache level, avoiding communicating with other packages on even worse the main memory, which could happen if we steal from a core that lies on another package.

Moreover, it avoids cache pollution that comes to play when we bring data from other packages to our package. The size of each cache level is limited, so bringing new data could evict other data that are needed by the other workers in the package, leading to cache capacity and conflict misses.

As proposed in[6], it would be better to restrict stealing from other packages, in order to minimize inter-socket communication cost. In order to achieve that, we permit only to one worker from each package to be able to steal from the other packages, while the rest can only steal local (in-package). We will be referring to these threads as the master worker of the package and the slave workers respectively.

3.2.2 Implementation details

In order to be able to choose victims according to the underlying architecture, it was necessary to pin each worker to a specific core. So, we created an extension to the arena class that contains the information needed, that is, the platform representation as well as a table that contains the physical cpu ids that library workers should be pinned on.

For pinning the OS threads to specific physical cores in order to be able to find the nearest core, a basic decision should be made. That is, which class of the library should carry the information about the physical core. The chosen class will enforce the entity that represents to work on a specific core. It would sound reasonable to add this information to the class *private_worker* that it represents a virtual worker and is bound with an OS thread lazily, when the library decides to launch it. However, this information was integrated to the *arena_slot* class. A number of such objects represent the available slots on each arena that require library workers to populate them and execute some of the arena's work. On the instantiation of the master thread's arena, each arena slot is assigned a number that represents the physical core that it should work on. Any worker that occupies an arena slot has to pin himself and work on the core it indicates. This decision was made mainly because the workers of the library, who correspond to OS threads, are entering and leaving arenas in a dynamic and unpredictable manner and it is not easy to keep track of which worker is active and when. That happens for several reasons, including:

- Lazy worker instantiation: the OS threads are created dynamically, according to arenas' needs and there is no guarantee that a worker is bound to an OS thread at a specific point in time.
- In some cases, more workers than actual cpus are instantiated, thus making impossible to bind a *private_worker* object to a specific cpu.
- Load imbalance: if a worker has no work to execute due to load imbalance, he goes to sleep, releasing the CPU and saving cpu time. Thus a worker that occupies a cpu core is not guaranteed to continue to do so in the future, because he may be migrated by the OS to a different core when he runs again.
- Workers are assigned dynamically to arenas that need them, so there is no guarantee that a pinned worker will work on the same arena with another.

The *arena_slot* class instead has a fixed number of instantiations for each arena, making it easier to keep track and identify which one corresponds to the nearest cpu. Moreover, the statistics mechanism of the library keeps track of several events and sums them up for each arena slot. Thus, it was obvious that the library design considers the arena slots as workers and not the dynamic instances of the class *private_worker*.

In order to run applications using different number of threads, we needed a map that maps workers to cpu ids. So, if we were to run an application with eight threads for example, they should be distributed evenly to different packages, two workers to each package in the case of Dunnington that has four packages. The map is essentially a table that contains the cpu ids of the underlying machine, ordered in a way so that for each thread count N, the first N numbers of the table are the cpu ids for an even distribution of the N OS threads to the packages. For the example of Dunnington, we present in the following picture the cpu id distribution that the operating system creates:

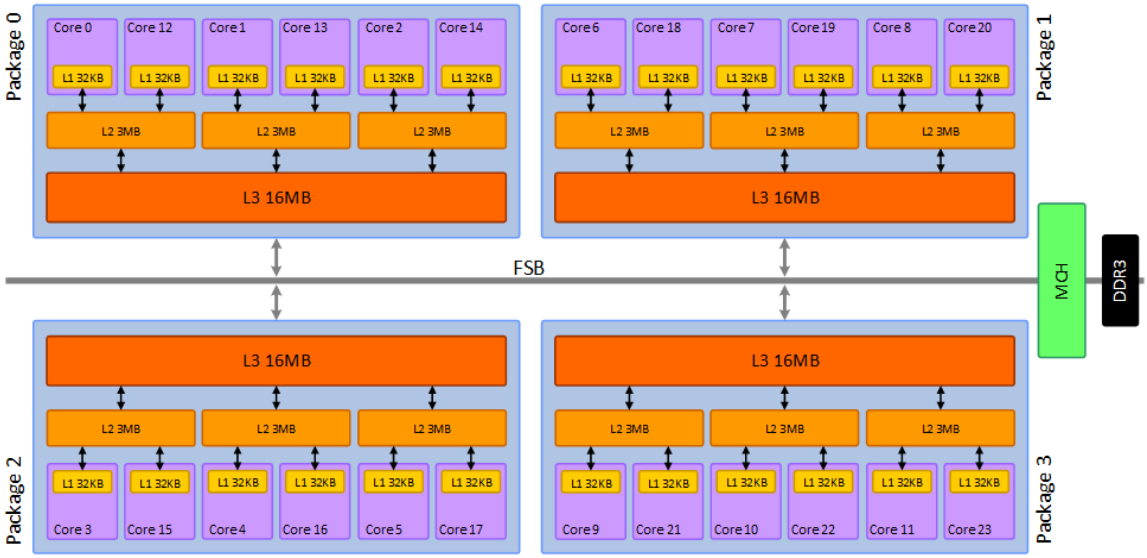

Figure 59. The numbers represent the cpu ids the OS assigns to each core on Dunnington

The corresponding map should be:

[0, 3, 6, 9, 1, 7, 4, 10, 12, 15, 18, 21, 13, 16, 19, 22, 2, 5, 8, 11, 14, 17, 20, 23]

On the arena instantiation, each of the arena slots that are created is assigned with a cpuid from this table, starting from the beginning, as shown in the following picture:

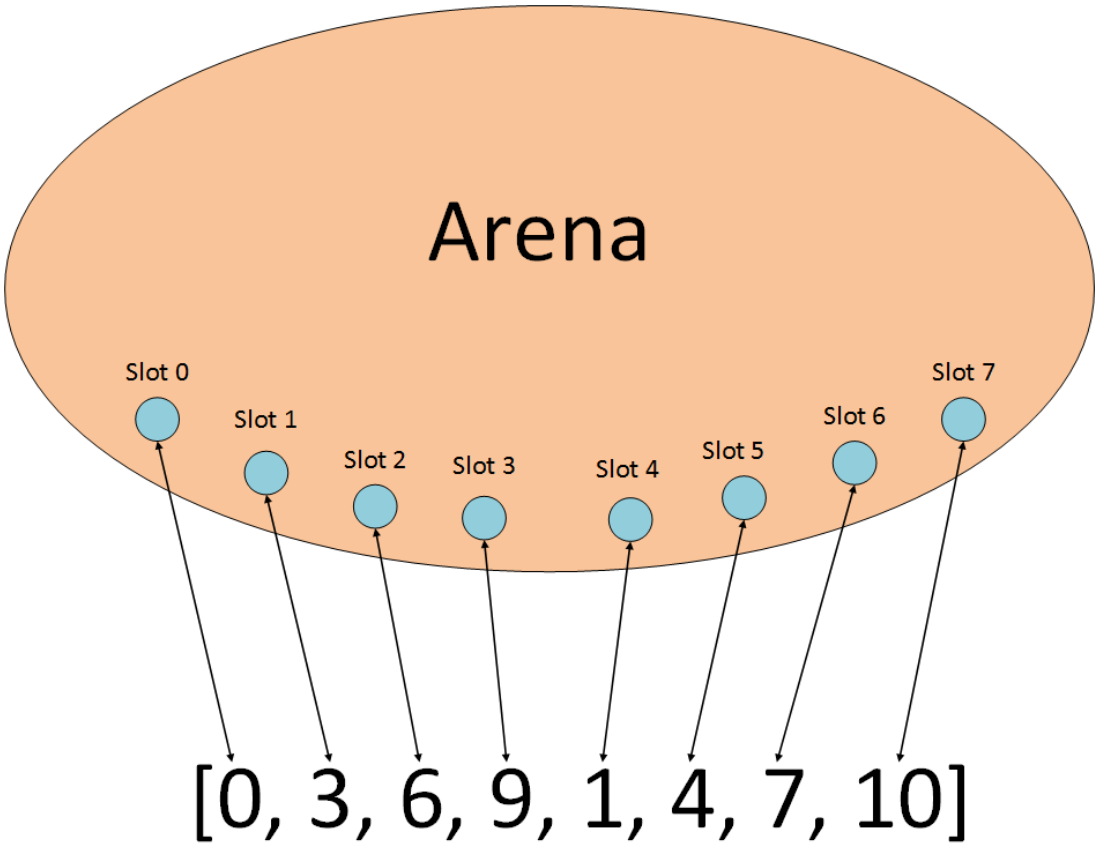

Figure 60. Cpu-id distribution to arena's slots

For choosing the closest neighbor, another data structure was necessary, namely, an adjacency list for each core that contains the core ids of the machine ordered from the closest to the furthest to that core. On the arena instantiation, we compute a 2-dimensional matrix that contains cpu ids and whose i-th row represents the adjacency list for cpu with id i. So, first come the cpus that share the L1 cache level with cpu i, then follow the cpus that share the L2 cache level, then the L3, and then come the cpus from the other packages. To avoid resigning too early from the nearest neighbors in case of failure and add the aforementioned persistency to choices, the first level of nearest neighbors (L2sharers in Dunnington and L1 and L2 sharers in Termi) are unfolded 50 times, because they are the fewest but nearest, the next level of sharers (L3 in both Dunnington and Termi) are unfolded 3 times, and the furthest (in the other packages) are unfolded only once. This matrix is used by the workers when they want to find a victim to steal work from.

Finally, a map from core ids to slot indexes is needed, which is essentially the inverse map from the distribution table that we mentioned first.

When a worker is left with no work and decides to steal some, he executes the stealing loop described by the pseudo code in Listing 4. In more detail, the mechanism works as follows: first, we find the row of the 2d-matrix with the neighbors that corresponds to the cpu id that our slot is working on. Second, we enter a loop that scans our neighbors

from the nearest to the furthest until we find an available one. We choose him as victim and from the inverse map we find the slot that he works on and try to steal from it. In case it fails, we continue with the next neighbor, until we reach the end of the array, in case we are the master worker of the package, or the end of the L3 neighbors, in case we are a slave worker. In this way, we achieve to steal from the nearest neighbors first.

```
fail_count=0;
while (fail_count < fail_threshhold)  {
        int idx;
        neighbors_list = find_my_neighbor_adjacency_list( my_cpu );
        do {
                int victim_core = get_next_neighbor(neighbors_list);
                idx = get_slot_index_from_cpu_id( cpu_to_index_map, victim_core );
                if ( I am master worker of mackage ) {
                        if ( reached the end of array neighbors )
                                continue_from_the_beginning;
                else
                        if ( reached the end of L3 neighbors )
                                continue_from_the_beginning;
                }
        } while ( idx slot is NOT populated by a worker );
        arena_slot* victim = &my_arena->my_slots[idx];
        t = steal_task( *victim );
        if (!t) {
                fail_count++;
                continue;
        }
}
```

Listing 4. Worker stealing loop

The results of the evaluation of this method are presented in the next Error! Reference source not found..

3.3 Stealing from the most loaded processor

The second attempt for optimization tries to tackle load imbalance problems. Load imbalance occurs naturally in some applications, especially when more synchronization points exist, for example in algorithms like parallel_reduce.

3.3.1 Technique description

Previous attempts [7] indicate that an occupancy-based approach to task stealing can bring performance improvements under some circumstances and in some scenarios. Our approach was finding the most loaded worker and stealing from him. It is obvious that such an approach can help distributing work more evenly than with the random stealing

approach. Load balancing may be a desired property, but we should also point out that stealing work from the heaviest could also result in cache pollution, if that worker is located to a different package.

The first approach is rather straightforward. If a worker needs to steal, he just scans the arena to find the most loaded worker and steals from him. It is easy to implement but doesn't give us any flexibility in case of failures.

This approach can be very costly to use each time a worker needs to steal work. For this reason, we tried to make a compromise by employing the stealing from the heaviest technique once in five stealing attempts. For the four remaining stealing attempts the classical random victim approach was followed.

In order to have more flexibility in case of failure, a second approach would be that each worker keeps a sorted list of the task loads of all the other workers and uses it to search for alternatives in case something goes wrong with the heaviest worker. This approach helps to distribute steals so that not all attempts fall on the heaviest worker.

There are two main variations of this technique, keeping global and local task load lists. The first variation tries to balance globally in order to alleviate inter-socket load imbalances, but it can lead to cache pollution and heavy inter-socket communication. The second variation tries to be more optimistic and cache friendly, by scanning locally in each package, permitting only the master worker of each package to scan globally, like we discussed earlier for the cache-aware technique. When work is more evenly distributed to packages it is makes sense to search locally for the heaviest, in order to balance the load even further without incurring too much overhead. On the other hand, if the load imbalance does exist between packages, overheads can be reduced by letting only the master workers of each package to contribute to balancing it, thus minimizing inter-socket communication. When scanning of the entire arena occurs, there are ping-pong effects between packages, because all the workers need to read the task load of all other workers.

3.3.2 Implementation details

To implement a mechanism to steal from the heavier cpu, in terms of task load, we needed to add a *current_load* field to the *arena_slot* class. When it is occupied by a worker, it is initialized to 0. The following three events induce changes to this variable:

- spawn by the owner worker: In this case the owner's current load is incremented by 1.
- get_task by the owner worker: In this case the owner's current load is incremented by 1.
- successful steal by any worker: In this case the victim's current load is decremented by 1.

The *current_load* variable counts in essence the enqueued tasks and works as an estimation of each active worker's load. It is not precise, because various tasks can differ in

size, so the number of enqueued tasks can be misleading. Previous work [7]indicates though that it can be a reasonably adequate estimation.

3.3.2.1 Finding Max

The implementation is rather straightforward. We scan every arena slot to find the max the worker that has the maximum *current_load*. Because this proved very costly we implemented a variation that employs this technique once in five steals, and the other four we follow the original random stealing algorithm.

3.3.2.2 Sorted List approach

Each worker keeps a list with the other workers and their load, sorted from the heaviest to the lightest. When stealing needs to happen, the worker picks his victims from this list, trying to steal from the most loaded worker.

The list needs to be refreshed with new estimations of each worker's load. To do this, we need to scan every slot and collect each worker's load. After that, they need to be sorted in reverse order. This procedure is costly and can cause performance degradation. Even if we simply scan the slots only to find the most loaded victim, it causes excessive performance degradation, if it is done on each stealing attempt. For this reason, we decided to refresh the list with new estimations once every five and once every ten stealing attempts. Every time the list is refreshed, we begin searching victims from the beginning.

In case of stealing failure, we move to the next most loaded worker. If we reach the end of the list we jump to the beginning. In case of successful stealing, we followed two different policies. The first policy (*Policy 1*) dictates that next time we need to steal, we will try to steal from the same victim with the previous successful stealing and we return to the beginning of the list only if we reach the end or we refresh the list. The second policy imposes that every time we need to steal again, even after a successful steal, we begin from the beginning of the list, that is, the most loaded worker. The following listing describes the technique in pseudo code:

```
fail_count=0;
while (fail_count < fail_threshhold)  {
        arena_slot *victim;
        if ( it's time to refresh the list ) {
                        read_task_loads
                        sort_loads_descending
                        go_to_the_beginning_of_the_list
        }
        if ( end_of_list )
                        go_to_the_beginning_of_the_list;

        victim = get_victim_from_list();

        t = steal_task( victim );
        if( !t ) {
                        move_to_next_victim_on_list;
                        fail_count++;
                        continue;
        }
        /*Policy2*/victim_iterator = victim_loads.begin();
}
```

Listing 5. Sorted-list technique algorithm

3.3.2.2.1 Global occupancy scan

The simple idea was to scan all the slots of the arena in order to refresh the occupancy list. That contributes to better load balancing, as it tries to alleviate the load differences of all the workers.

The tradeoff is that except for balancing, it can also cause workers to steal from others that are very far away, in terms of cache hierarchy, thus polluting their cache levels with potentially irrelevant data, since neighboring cpus share some cache levels and may work on data that are unrelated to the newcomers, leading to more capacity and conflict misses. Except for that, false sharing between packages can cause severe performance degradation.

3.3.2.2.2 Local occupancy scan

The other approach would be to scan only the local workers within the package and keep an occupancy list that contains only estimations about local workers. So each worker can steal only from workers in his package, preferring the most loaded each time. This approach also incorporates benefits from the cache-aware stealing mechanism than was analyzed in the section 3.2, like maintaining cache locality of the data, minimizing cache pollution and inter-package stealing.

The implementation is exactly the same with Listing 5, except for the "read_task_loads", which in this case uses the adjacency list we implemented for the cache-aware technique to find the in-package workers and their loads.

Chapter 4
Evaluation

4.1 Physical Systems

The physical systems we used were the same as in the profiling section (Dunnington and Termi) as well as "Sandman" NUMA Platform, a 32-core NUMA machine with the following characteristics:

- 4 Packages (Intel(R) Xeon(R) X5650 @ 2.67GHz)
- 8cores per Package
- Hyperthreading (64 threads in total)
- 32KB L1 cache per core (2 threads)
- 256KB L2 cache per core (2 threads)
- 16MB L3 per package (8 cores, 16 threads)
- 257.931 MB RAM

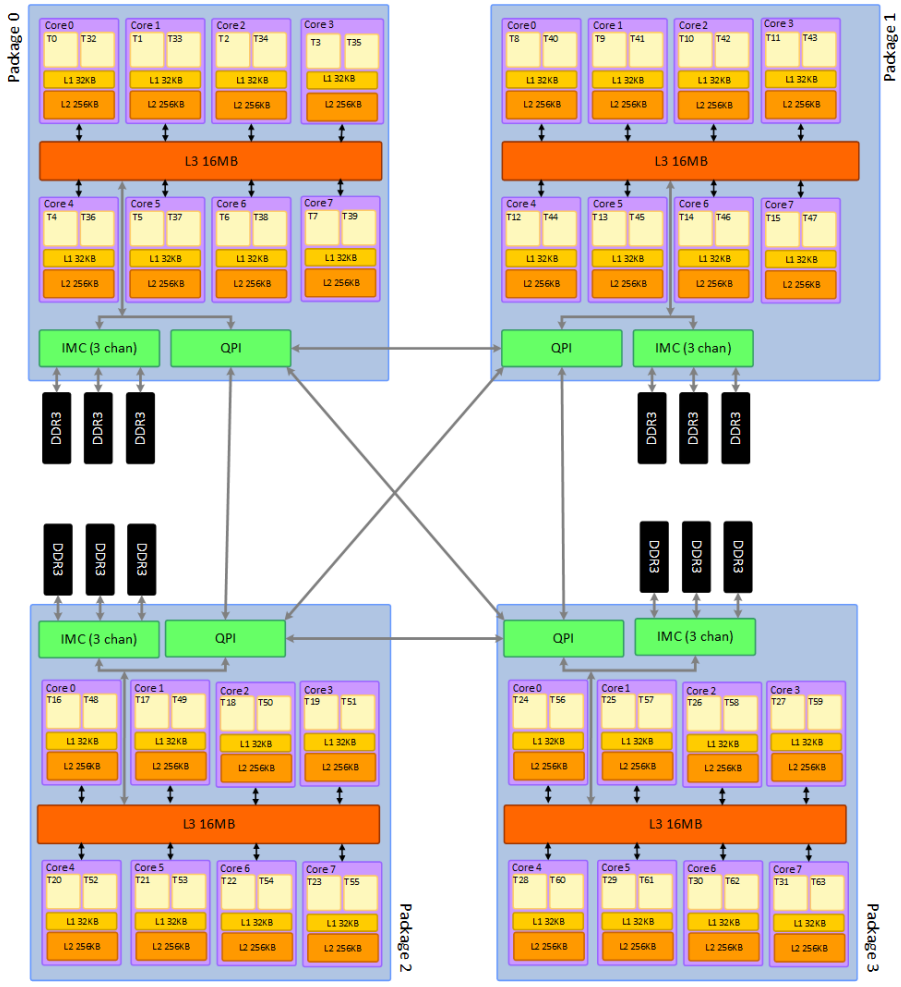

Figure 61. 32-core "Sandman" NUMA Platform

4.2 Stealing from the nearest neighbor

4.2.1 Benchmarks Used

Based on the previous work, we used a series of applications that are known to be sharing-intensive as well some sharing-mild algorithms. The sharing-intensive algorithms are Gauss Elimination, Heat and Floyd-Warshall. The memory-mild are quicksort and matrix multiplication. We included an implementation of Word-Count as a sharing-intensive representative example of the map-reduce algorithm category, using the parallel-reduce template algorithm of TBBs.

Algorithm	Description	Input size
Gauss Elimination	Linear systems solution	1024x1024
5-point Heat Algorithm	2D Heat Equation	2048x2048
Floyd-Warshall	All-pairs shortest paths	4096x4096
Word Count	Counting number occurrences in matrix	12000x12000

4.2.2 Results

In order to test our implementation on equal terms with the random mechanism, we pinned the OS threads of the original library to the same cores for each thread count as the custom library, while still using the random stealing policy. This is especially important in the case of small numbers of threads. If the original library lets the operating system distribute the threads to packages and cores at will, thread migrations between packages as well as uneven distribution have been observed, in opposition to the custom library that keeps the OS threads to specific cores throughout the execution.

In the following sections we present selected results of the aforementioned applications.

4.2.2.1 Heat

As the following figures suggest, the 5-point heat algorithm benefited greatly from the cache-aware approach on the NUMA platforms. On SMP platforms there was performance degradation, as it can be seen on the relevant figures of Appendix B. On Termi there is a performance benefit of up to 3,6% for large thread counts.

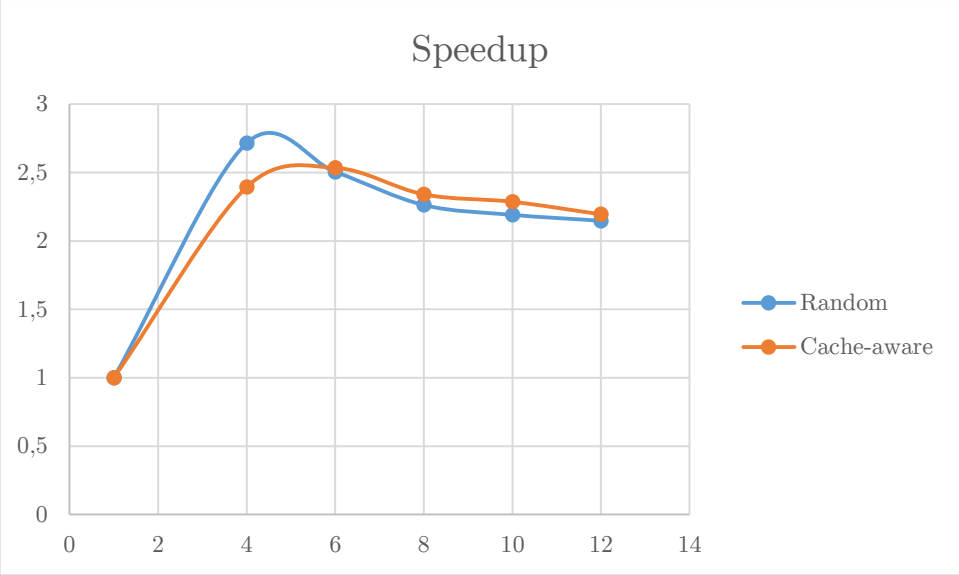

Figure 62. Speedup of 5-point Heat on Termi (Cache-aware)

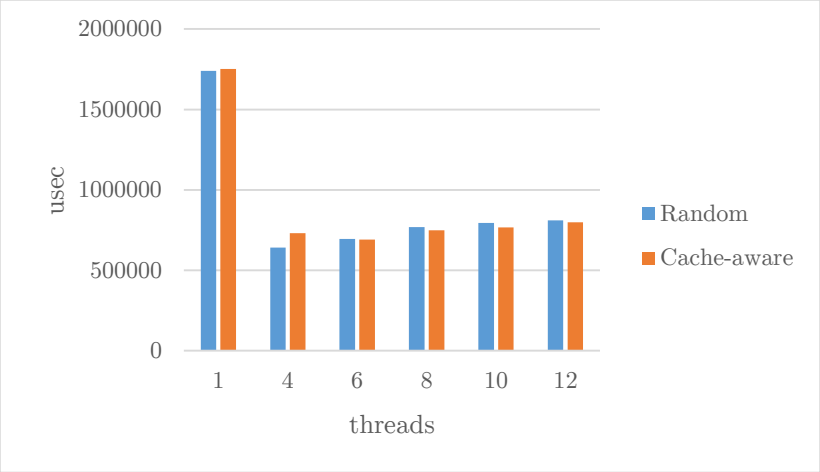

Figure 63. Execution times of 5-point Heat on Termi (Cache-aware)

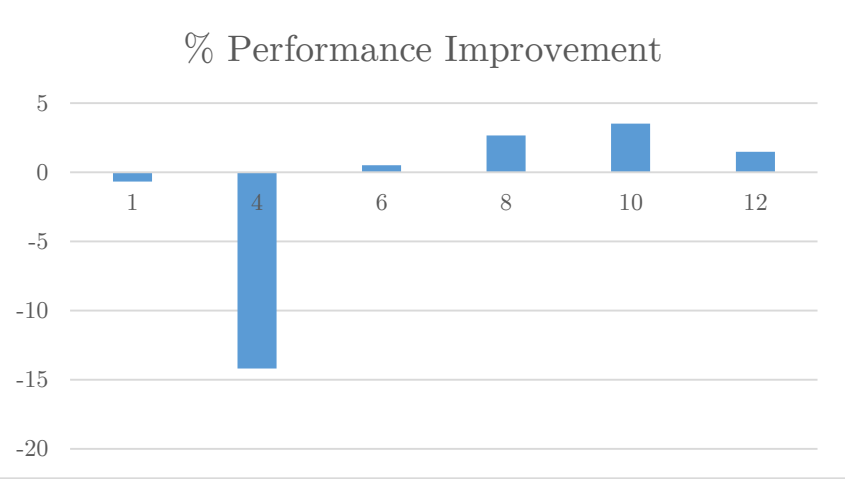

Figure 64. Performance gains of 5-point Heat on Termi (Cache-aware)

On Sandman there was even greater performance boost, reaching up to 40% in large thread counts, indicating that the more NUMA packages, the larger the potential of the cache-aware technique to exploit localized work-stealing.

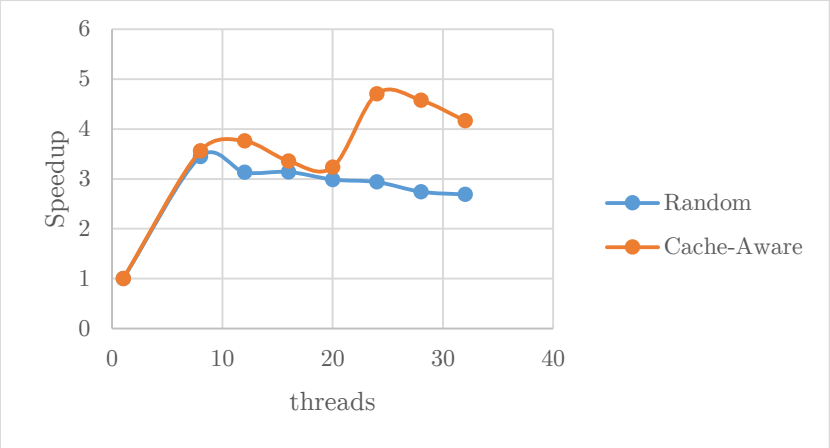

Figure 65. Speedup of 5-point Heat on Sandman (Cache-aware)

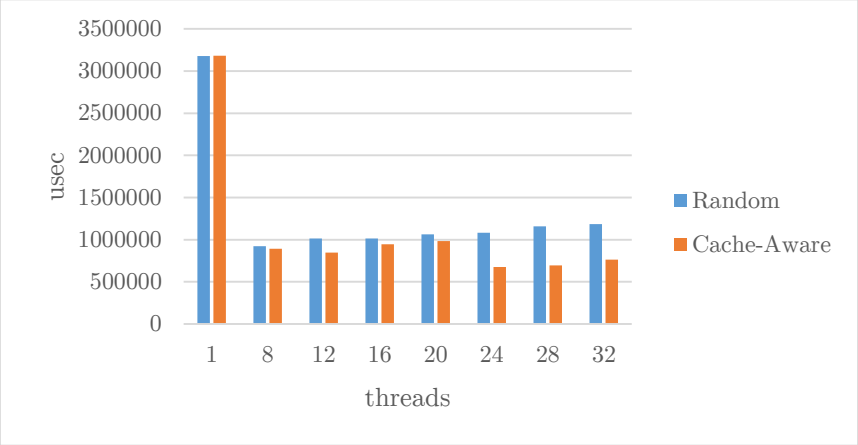

Figure 66. Execution times of 5-point Heat on Sandman (Cache-aware)

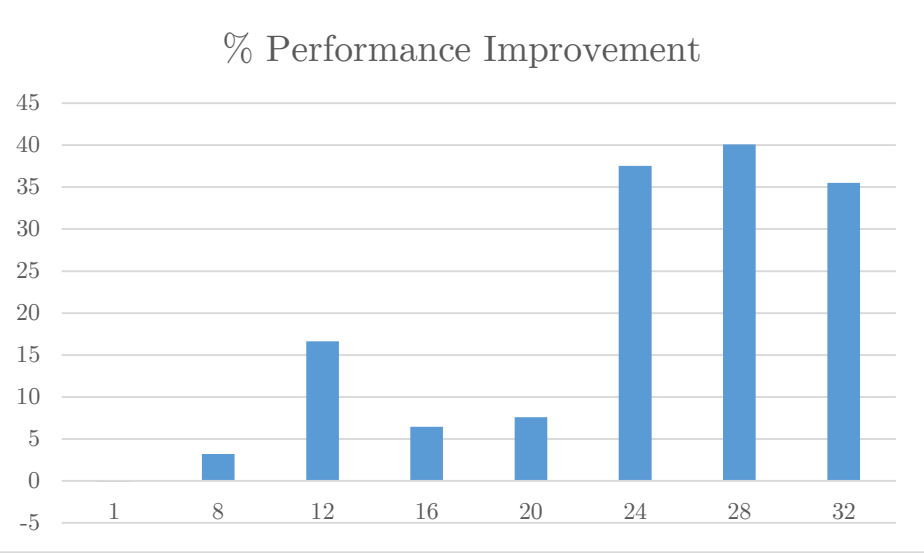

Figure 67. Performance gains of 5-point Heat on Sandman (Cache-aware)

4.2.2.2 Word count

Word count showed great performance improvement on Sandman, while on Dunnington showed almost the same performance and on Termi suffered from excessive performance degradation.

In this algorithm, every task keeps a private map that counts the words for its subproblem, and when a join occurs the task merges its private map with the private map of another task. That means accessing data that were written by another core recently. This results in flushing the changes to the main memory. Stealing tasks from the same package effectively reduces this overhead as most of this information is likely to be found in some cache level, like the L3 level.

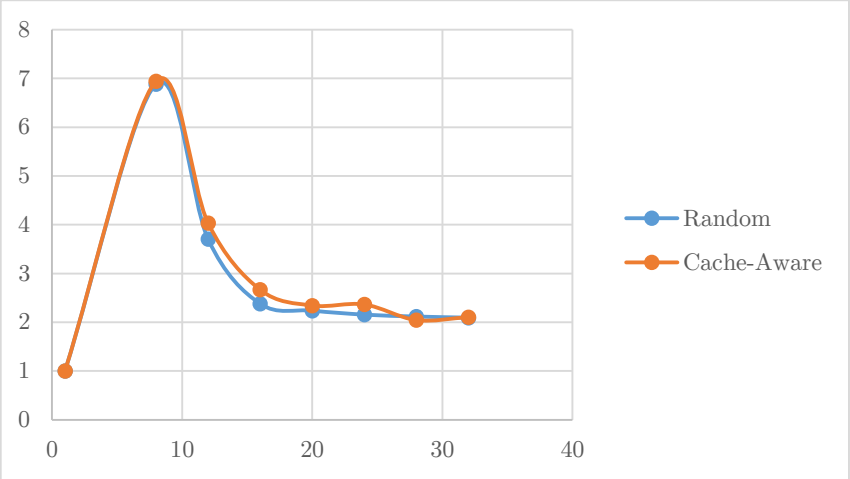

Figure 68. Word Count speedup on Sandman (Cache-aware)

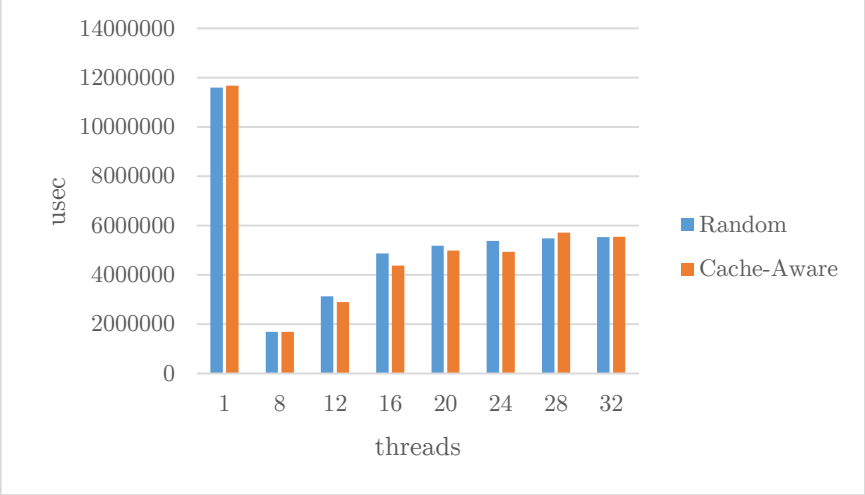

Figure 69. Word Count execution times on Sandman (Cache-aware)

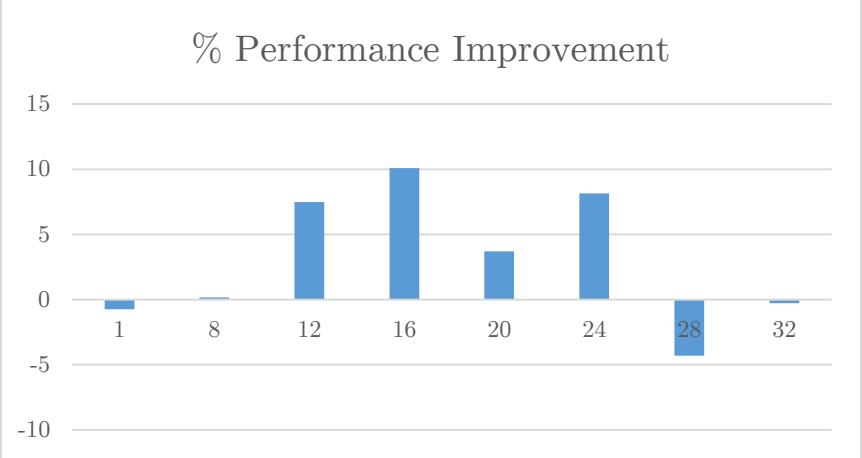

Figure 70. Word Count performance gains on Sandman (Cache-aware)

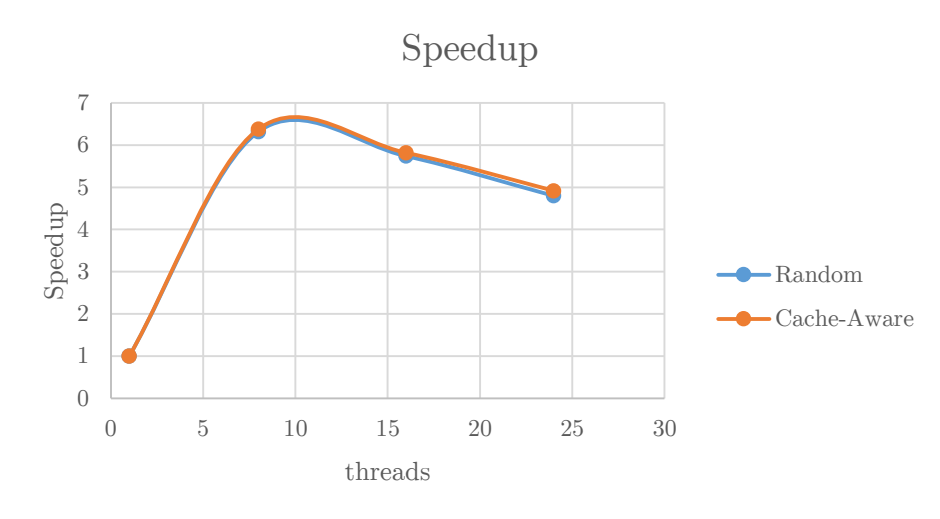

Figure 71. Word Count speedup on Dunnington (Cache-aware)

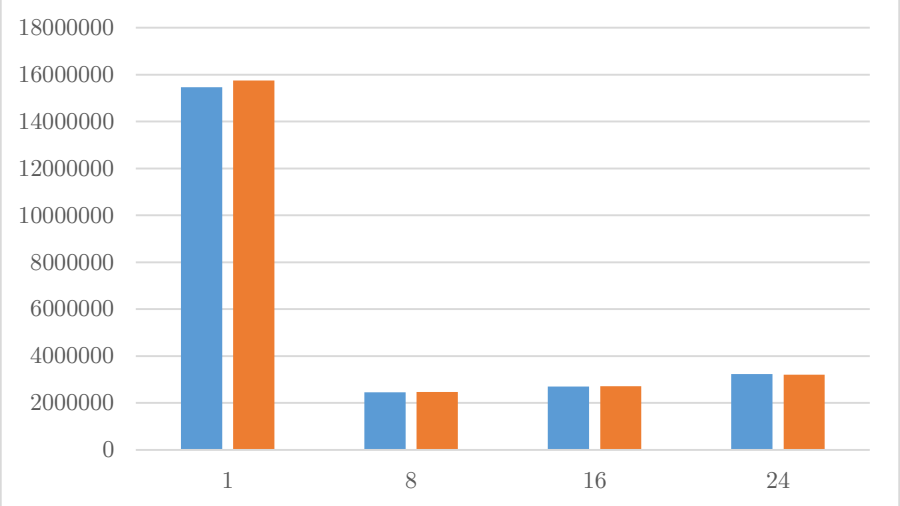

Figure 72. Word Countexecution times on Dunnington (Cache-aware)

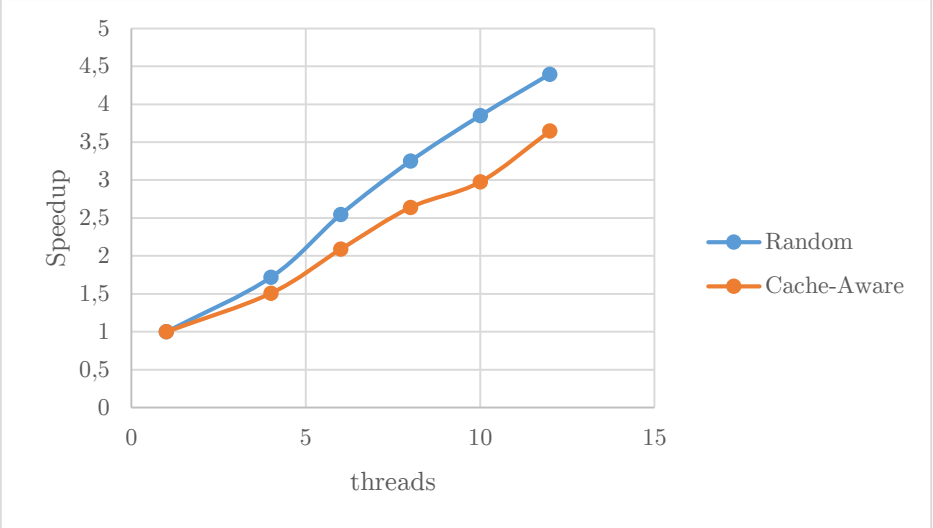

Figure 73. Word Count speedup on Termi (Cache-aware)

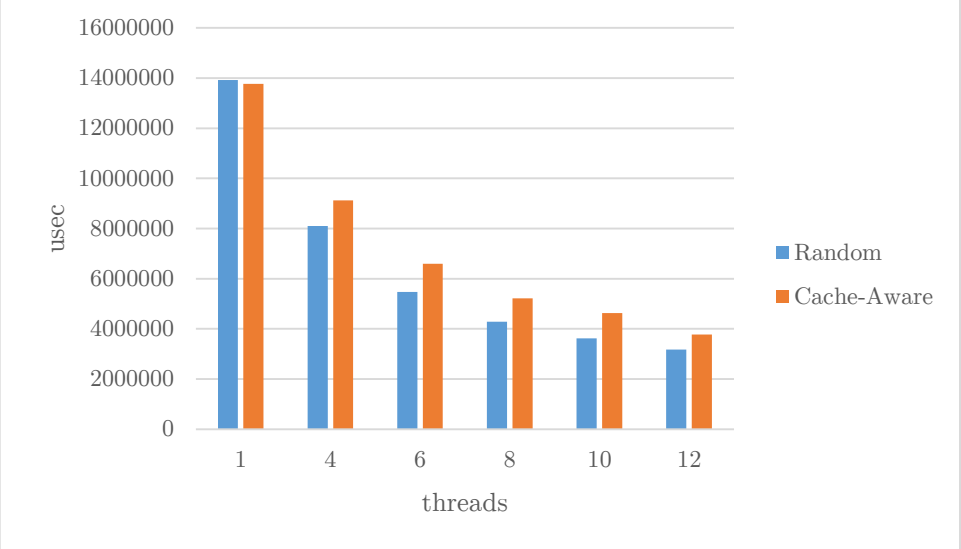

Figure 74. Word Count execution times on Termi (Cache-aware)

4.2.3 Remarks

The above examples offer the proof of concept that preferring stealing from neighbors that lie near our core, in terms of cache hierarchy, can indeed bring great performance improvements.

It is notable to mention that the mechanism we implemented does not incur large overheads and applications that do not benefit from this technique do not suffer from performance deterioration either. A number of applications appear to have the same performance as the random stealing, as their access pattern is not affected by locality issues because there is not substantial read-write sharing between cores, thus degenerating the choice to equal to random. In particular, these applications are *Quicksort* and *Matrix multiplication.*

4.3 Load Balancing

4.3.1 Benchmarks Used

We used a series of applications that previous work has shown to suffer from load imbalance [7], namely *streamcluster*, *swaptions*, *blackscholes*, with emphasis on the first. For some implementations we also used *strassen*, *quicksort* and *matrix multiplication*.

4.3.2 Results – Finding Max

4.3.2.1 Searching for the heaviest every time

This approach was proved very inefficient, as it can be seen on the following figures, which present the behavior of the *streamcluster* application. The main reason is the overhead to scan all arena slots on each stealing attempt to find the heaviest. All the results can be found in Appendix B.

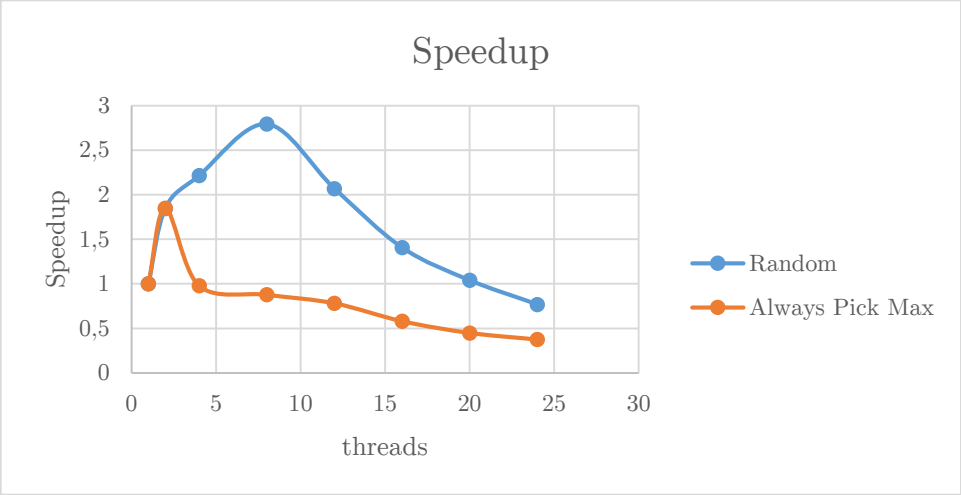

Figure 75. Streamcluster speedup on Dunnington (Just pick max)

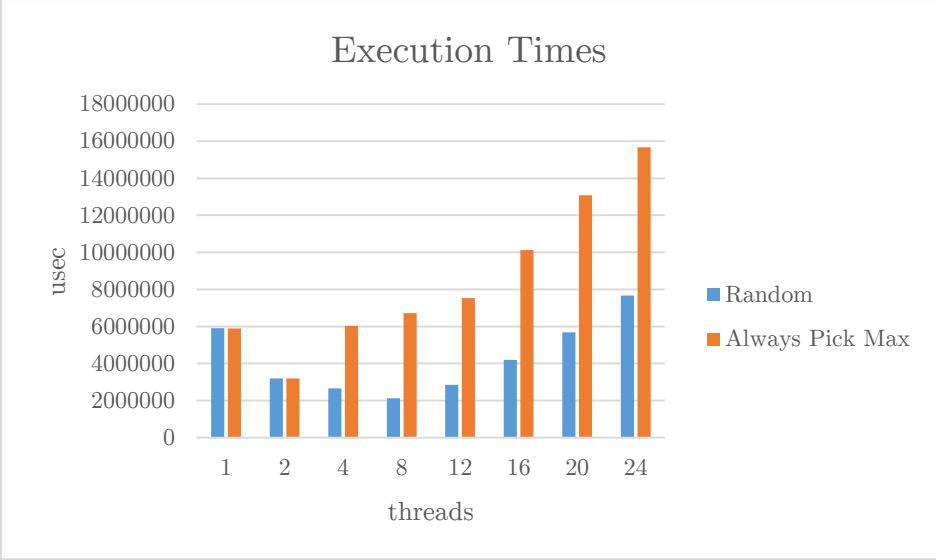

Figure 76. Streamcluster execution times on Dunnington (Just pick max)

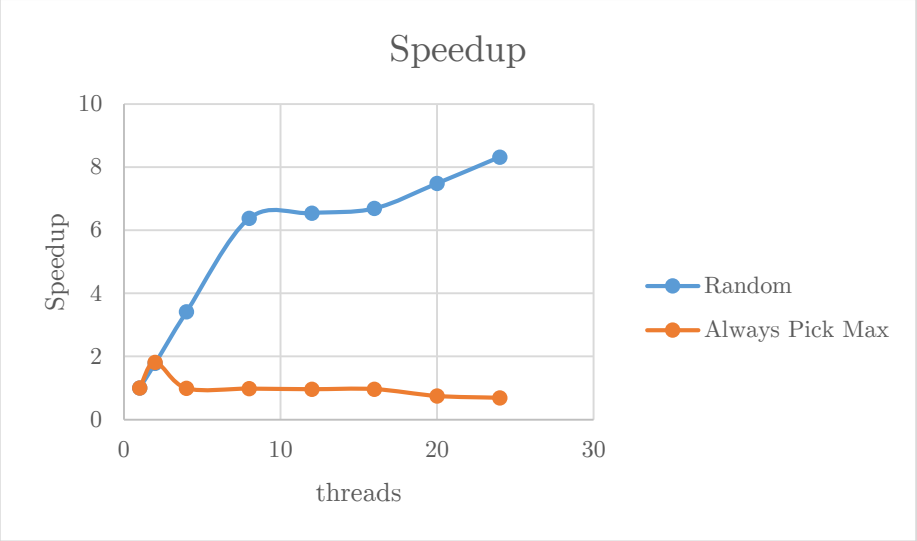

Figure 77. Streamcluster speedup on Termi (Just pick max)

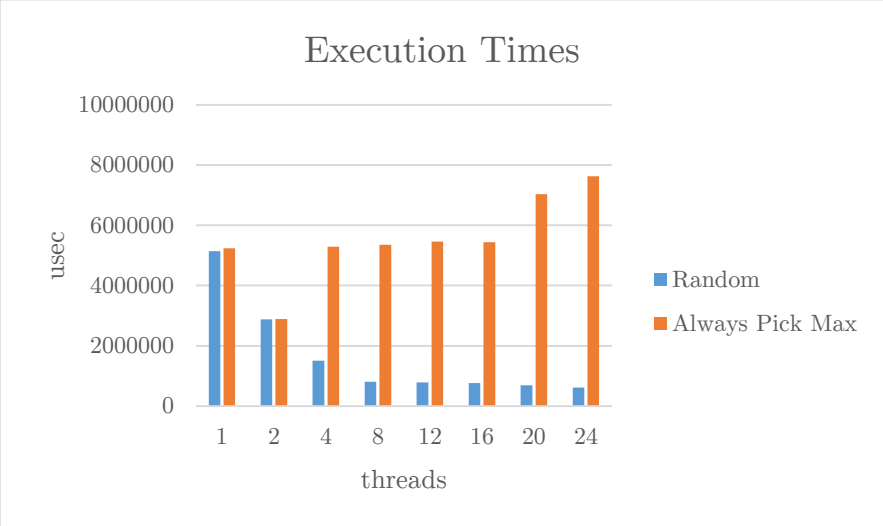

Figure 78. Streamcluster execution times on Termi (Just pick max)

4.3.2.2 *Searching for the heaviest once in five steals*

The excessive performance degradation caused by the overhead of scanning the whole arena on every stealing attempt can be effectively avoided through a compromise between it and the original random stealing technique. The result is effective without incurring much overhead.

The following figures show the performance gains of *streamcluster* on Dunnington and Sandman.

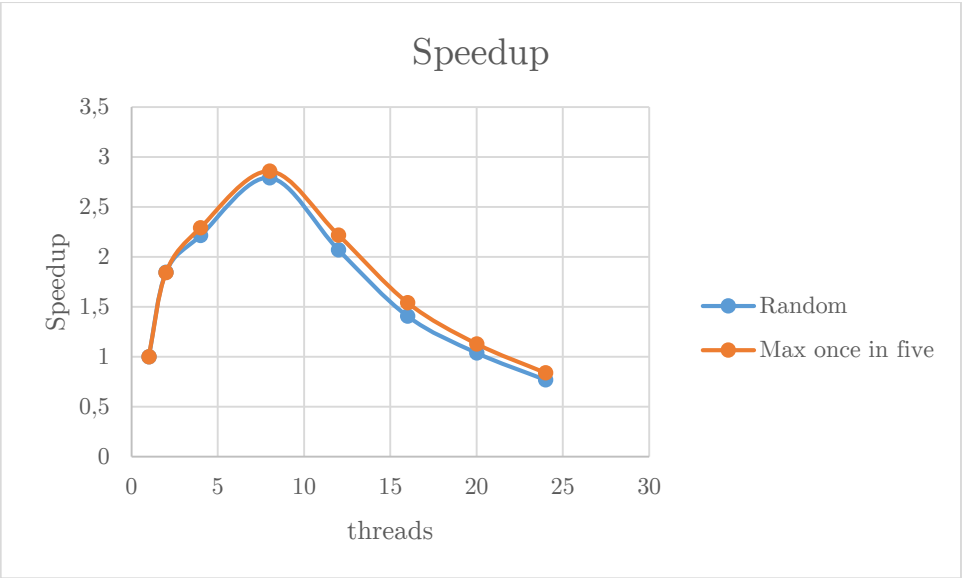

Figure 79. Streamcluster once in five steals speedup on Dunnington (Once in five)

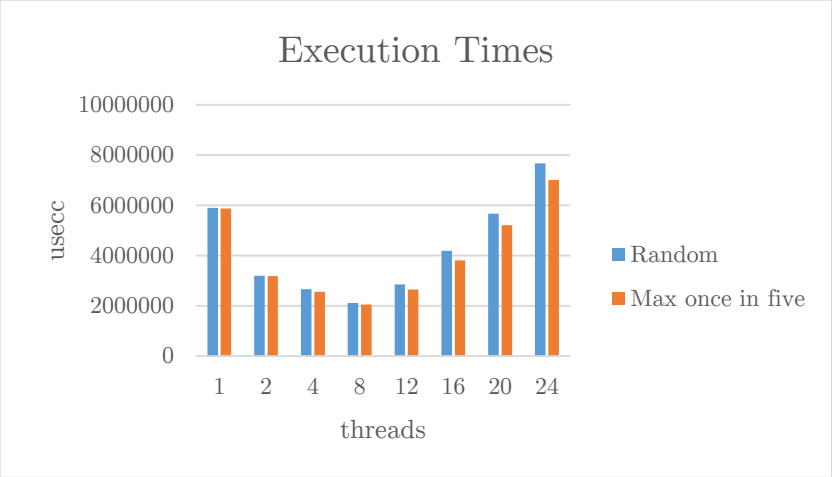

Figure 80. Streamcluster once in five execution times on Dunnington (Once in five)

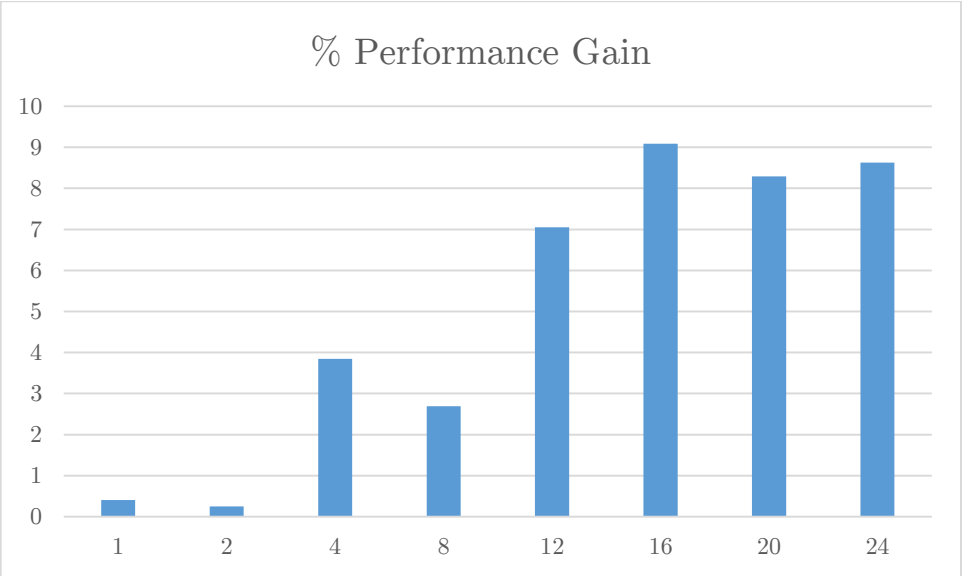

Figure 81. Streamcluster once in five Dunnington % Performance Improvement (Once in five)

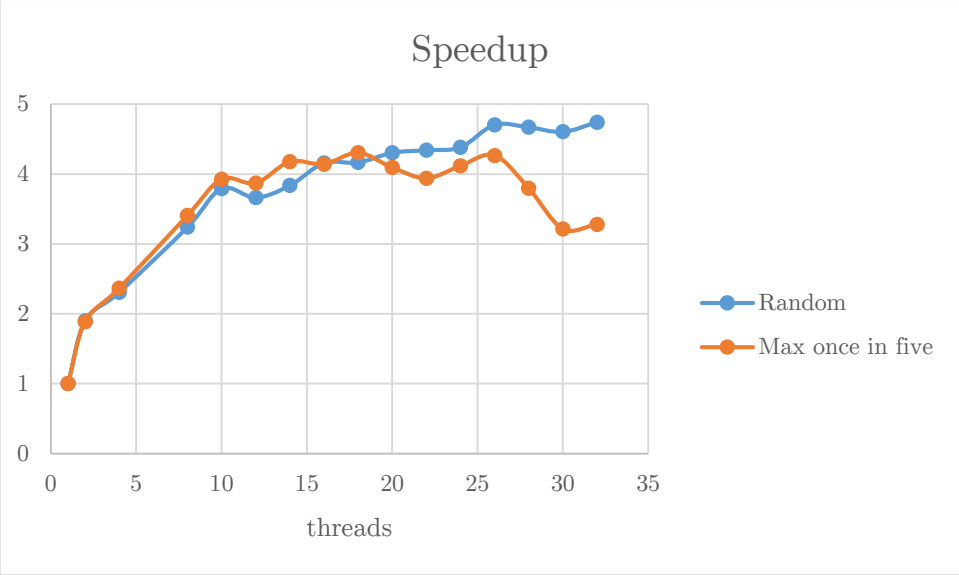

Figure 82. Streamcluster once in five Sandman Speedup (Once in five)

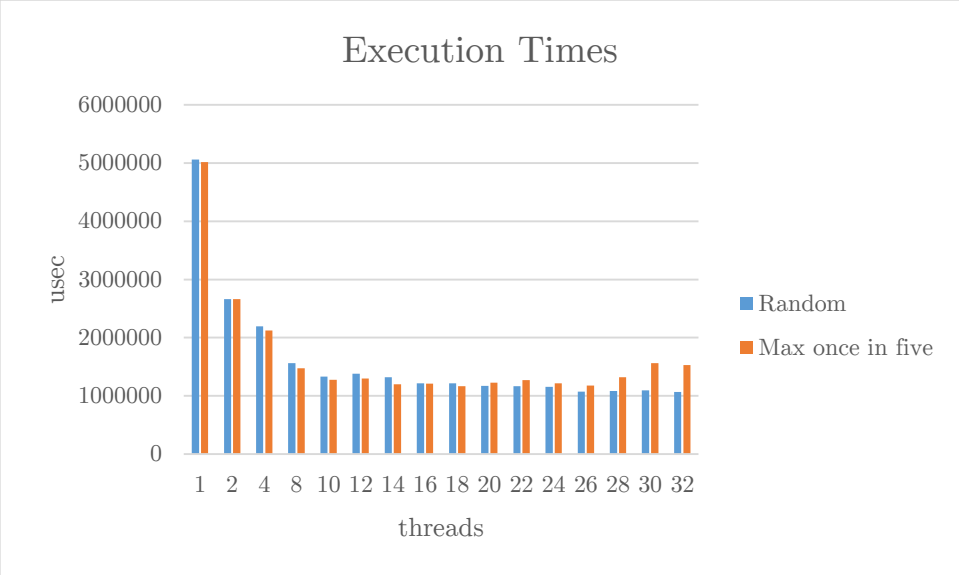

Figure 83. Streamcluster once in five Sandman Execution Times (Once in five)

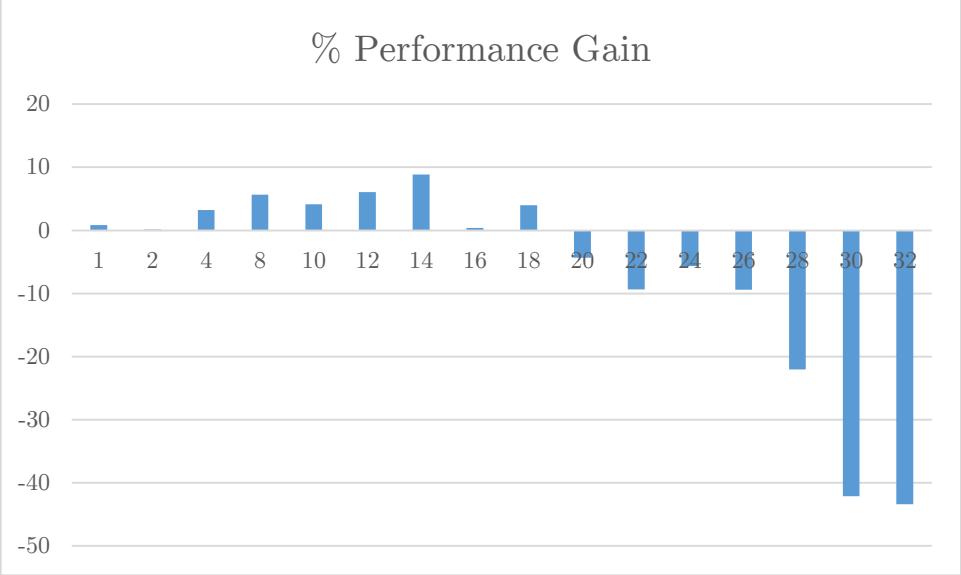

Figure 84. Streamcluster once in five Sandman % Performance Improvement (Once in five)

This technique has positive effects up to a certain number of threads. For larger thread counts it suffers from performance degradation. The reason for that is that every scan needs to check the load of all the other cores, which comes with inter-socket communication. As the thread count rises, the overhead of these communications dominates the benefits of the technique, resulting in performance deterioration. In order for applications to benefit from this technique, there should be load imbalance between packages and run them with a limited number of threads.

4.3.3 Results–Global vs Local Sorted List

The following sections present the results of the *sorted list* technique, demonstrating the impact of the various alternatives of the two implementations on each application. Most of the results are in Appendix B, due to their large number. Here we present the most notable results, namely the applications and cases that benefited most from this technique.

4.3.3.1 Streamcluster

Streamcluster benefited by the local search technique on the SMP for large thread counts, achieving performance boost up to 26%, although in smaller thread counts there was significant performance degradation. This occurs in small thread counts especially in the non-grouped versions because in every package only a few workers exist and it is more often to pick a victim from a different package, resulting in more ping-pong effects, given the initial load imbalance that comes with *streamcluster*.

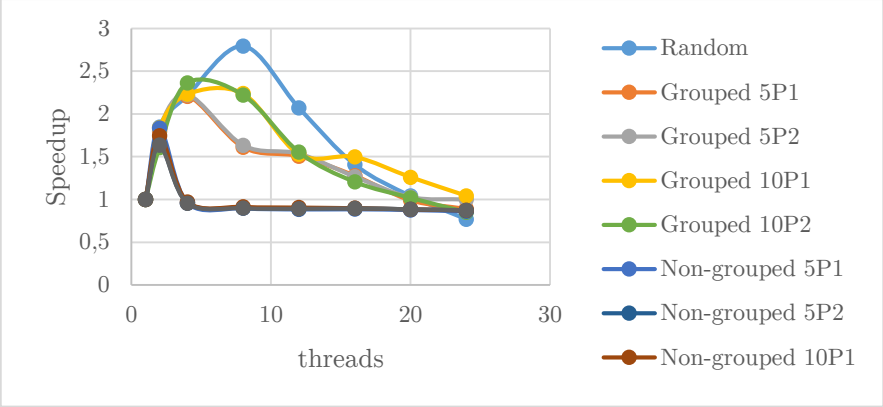

Figure 85. Streamcluster speedup on Dunnington (sorted-list)

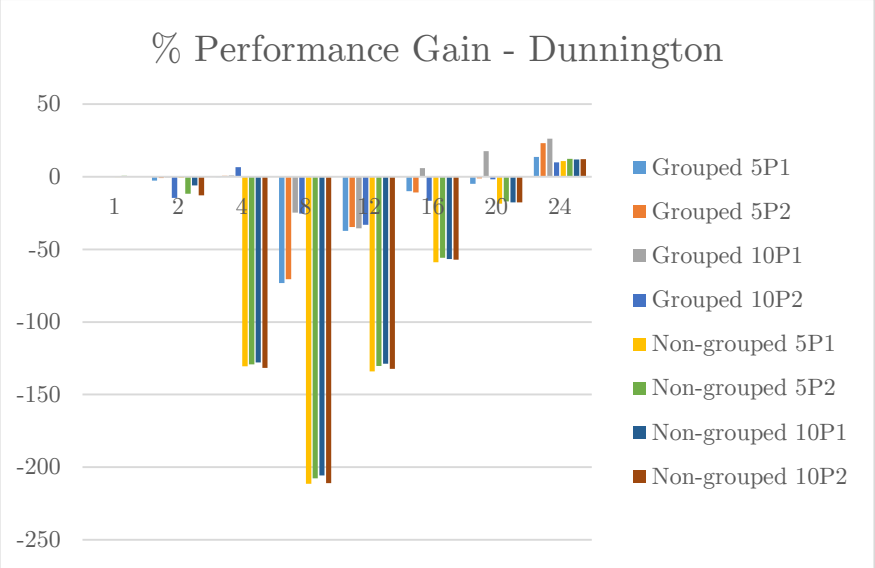

Figure 86. Streamcluster performance gains on Dunnington (sorted list)

4.3.3.2 Quicksort

Quicksort benefited by the local search version on large thread counts on the SMP, achieving a performance boost up to 17% without incurring too much overhead on smaller thread counts.

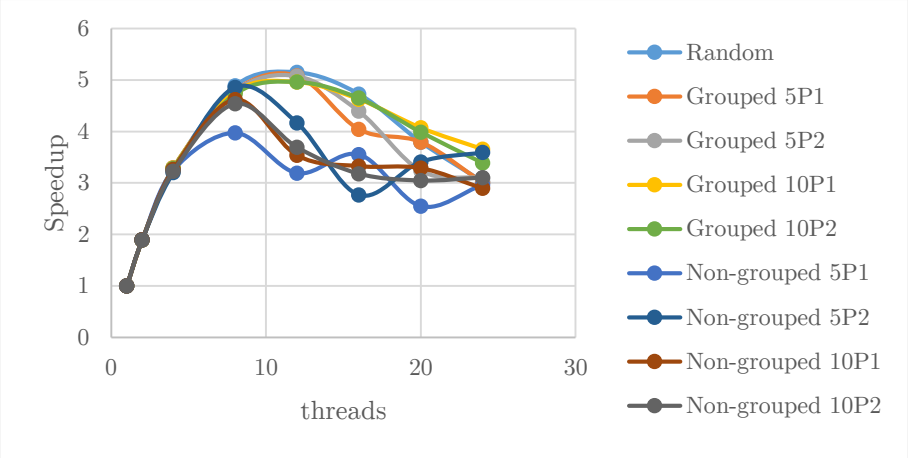

Figure 87. Quicksort speedup on Dunnington (sorted list)

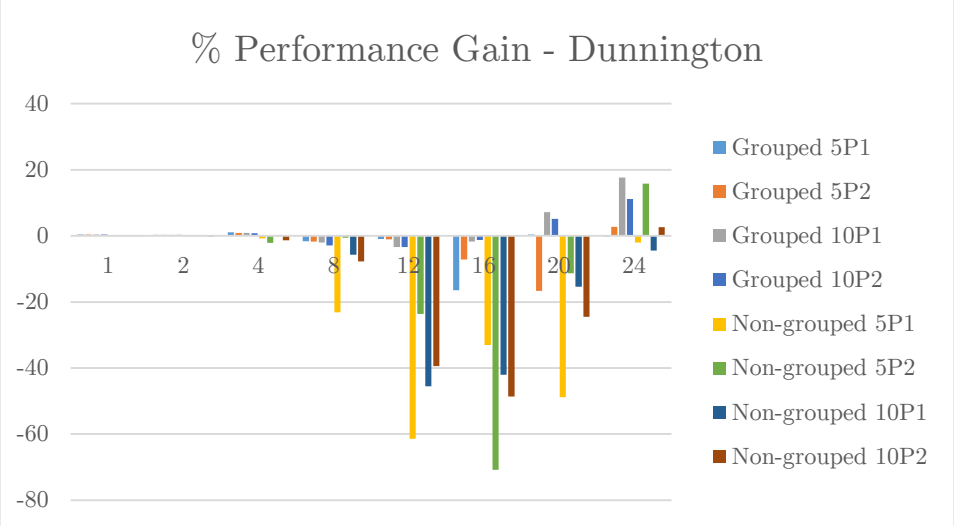

Figure 88. Quicksort performance gains on Dunnington (sorted list)

4.3.3.3 Matrix Multiplication

Matrix multiplication benefited on Sandman from both techniques on various large thread counts, achieving up to 6.2% improvement. Nevertheless, the global search technique incurred more overhead on smaller thread counts.

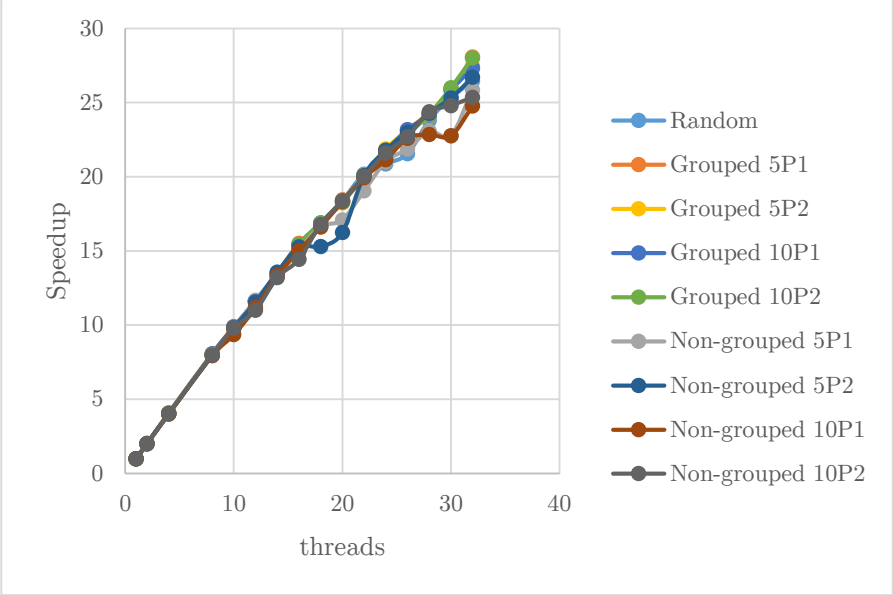

Figure 89. Matrix multiplication speedup on Sandman (sorted list)

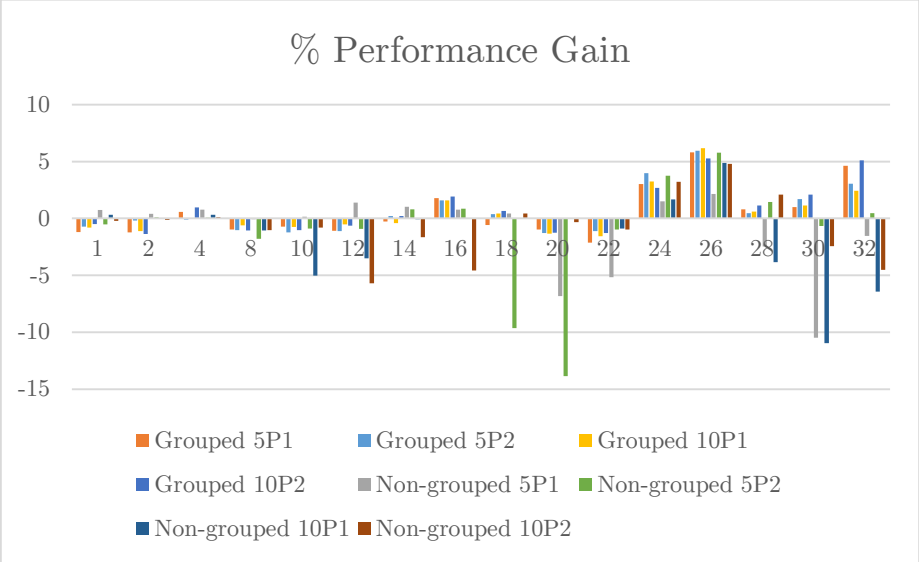

Figure 90. Matrix multiplication performance gains on Sandman (sorted list)

4.3.3.4 Remarks

From the graphs presented, we can make the following remarks:

- The applications can be divided into two main categories, the ones that benefit from the global search technique on all machines (*swaptions*) and the ones that benefit from the local search technique on all machines (*strassen* and *streamcluster*). That means that the applications of the first category have load imbalance that is located mainly on a small number of workers and it can be alleviated by the global search technique, and the applications of the other category have a more even distribution of work, so that local search benefits local balancing without incurring as much overhead as the global search.

- In most cases the applications suffer from severe performance deterioration, due to the overhead of refreshing the sorted list.
- The two techniques have almost the same performance on Dunnington and Sandman for the *Matrix multiplication*.
- There are some applications for which some techniques work better on some machines, while the other techniques have advantage on other machines. To be more specific:
  - Global/Non-Grouped Search
    - *Quicksort*, *Blackscholes* and *Matrix multiplication* on Termi
  - Local/Grouped Search
    - *Quicksort* and *Blackscholes* on Dunnington and Sandman

  In general we could argue that local search brings performance benefits on SMP platforms for large thread counts, because it avoids cache pollution and minimizes the ping-pong effect between packages and transactions with the memory that are serialized on the memory bus, which is shared by all cores. That means that we could benefit from the local search technique only in large thread counts on such platforms.

  Moreover, we could further argue that the overhead of each technique always exists, suggesting that in the cases that a technique appears to have insignificant overhead, the reason is that the benefits of the technique compensate for its overheads. That means that although we do not have positive performance gains in most cases, each technique does have a positive impact on stealing, despite that its overheads dominate in some cases. Testing the techniques on a simulator rather than on real machines, where we could minimize the overheads ideally, could help making the benefits of each technique more apparent.

Chapter 5
Epilogue – Conclusions & Future Work

5.1 Conclusions

From all the above, it should be clear that there is not a silver bullet for reducing overheads on parallel programs. Many of the desired properties may be even contradictory, as with load balancing and better cache use. It was proven by testing on physical machines that applications can benefit from a cache-aware approach to stealing on large NUMA machines for large thread counts, depending on the memory access pattern of the application and the read/write sharing dependencies between workers. Balancing techniques, like stealing from the most loaded worker, can bring performance boost when used in cooperation with the random stealing technique. More complex mechanisms that keep track of each worker's task load in a more detailed manner, could perform effective load balancing on some cases, they can suffer though from severe overheads on physical machines.

5.2 Related Work

In [6], a similar approach to our cache-aware stealing mechanism was tested on Cilk, which has a similar task stealing mechanism as TBBs. During the first run of a parallel algorithm the runtime classifies the tasks as inter-socket and intra-socket, mainly by their sizes and their depth on the recursive tree splitting. On later runs of the same algorithm, the runtime permits inter-socket stealing of the inter-socket tasks (which are the largest and do not fit in the cache) and intra-socket stealing of the intra-socket tasks (that fit in the cache).

In [7] a thorough characterization on the basic TBB functions is presented as well as an occupancy based approach to stealing as well as a criticality-based approach, which takes into consideration the relative complexities and lengths of tasks. The tests were carried out on simulator, which gives insight about the effectiveness of the idea, but not its overhead on physical machines.

In [8] various optimization approaches are presented, including lazy split and join for the parallel-reduce template algorithm, automatic grain size determination as well as various changes to the loop templates and the task scheduler to impose better task affinity between and optimize some stealing scenarios.

5.3 Future Work

There is an extremely large number of combinations of access patterns and machine architectural configurations, thus making the exploration and categorization of each situation a difficult task. Nevertheless, based on our work, there are some steps that should be tested in the future in order to shed some more light on the directions we already have started to explore:

- Grouping applications and patterns, in order to be able to make predictions about the best technique for the best case.
- Run configurations on larger NUMA machines, so as to expose the potential of the cache-aware technique under such architectural specificities.
- Implement the actual inter- and intra-socket task queues on TBBs, as specified in [6].
- Creation of a mechanism that breaks the initial work into large pieces and distributes them locally to each package at the beginning. This technique, when combined with the cache-aware stealing policy, would enable a more efficient utilization of the memory hierarchy (more effective cache space available to workers, more memory accesses satisfied locally, etc.), while maintaining the properties of load balancing.
- Investigate hybrid schemes that compromise the aforementioned techniques with the classic random stealing (it was proved successful with some load balancing techniques).
- Implement an automatic mechanism to pick the largest number of workers that we could benefit from.

Chapter 6
Bibliography - References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine,* 1965.

[2] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal,* vol. 30, no. 3, pp. 202-210, 2005.

[3] M. J. Flynn, "Some computer organizations and their effectiveness.," *IEEE Transactions on Computers,* Vols. C-21, no. 9, 1972.

[4] G. Amdahl, "The validity of the single processor approach to achieving large scale computing capabilities.," in *In Proceedings of AFIPS Spring Joint Computer*, N. J., 1967.

[5] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors.," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009.

[6] Q. Chen, M. Guo and Z. Huang, "CATS: cache aware task-stealing based on online profiling in multi-socket multi-core architectures," in *ICS '12 Proceedings of the 26th ACM international conference on Supercomputing*, New York, 2012.

[7] A. Bhattacharjee, G. Contreras and M. Martonosi, "Parallelization libraries: Characterizing and reducing overheads," *ACM Transactions on Architecture and Code Optimization (TACO),* vol. 8, no. 1, 2011.

[8] A. Robinson, M. Voss and A. Kukanov, "Optimization via Reflection on Work Stealing in TBB," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on , vol., no., pp.1,8, 14-18 April 2008*, 2008.

[9] J. Reinders, Intel threading building blocks: outfitting C++ for multi-core processor parallelism, O'Reilly, 2010.

[10] U. Drepper, "What every programmer should know about memory.," Red Hat Inc., 2007.

Chapter 7
Appendix A – Profiling Results

7.1 Quicksort

Quicksort showed similar behavior to *Blackscholes*, that is, better scaling on the NUMA platform.

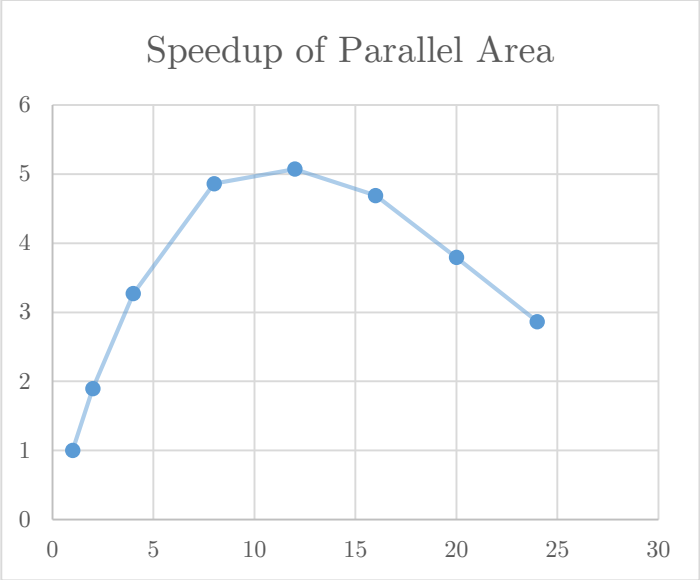

Figure 91. Quicksort speedup on SMP

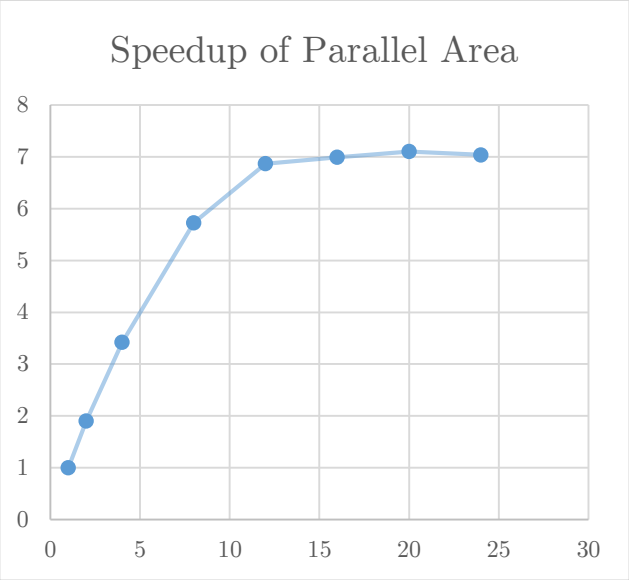

Figure 92. Quicksort speedup on NUMA

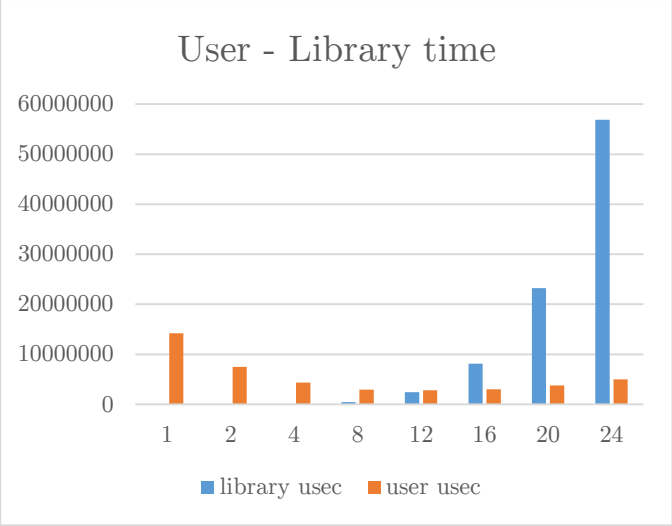

Figure 93. Quicksort User-Library time on SMP for each thread count

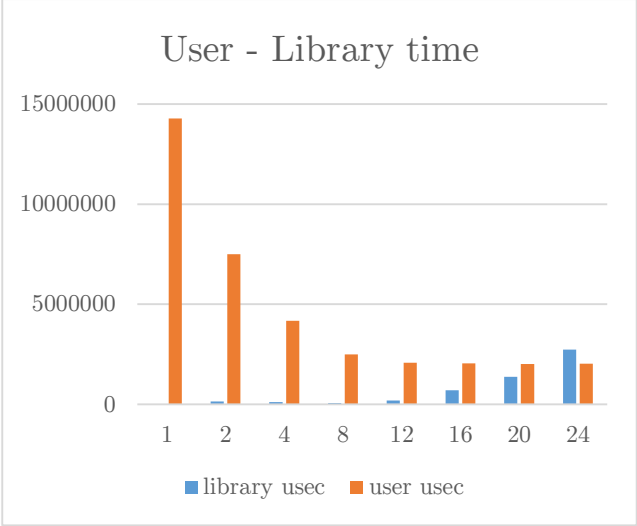

Figure 94. Quicksort User-Library time on NUMA for each thread count

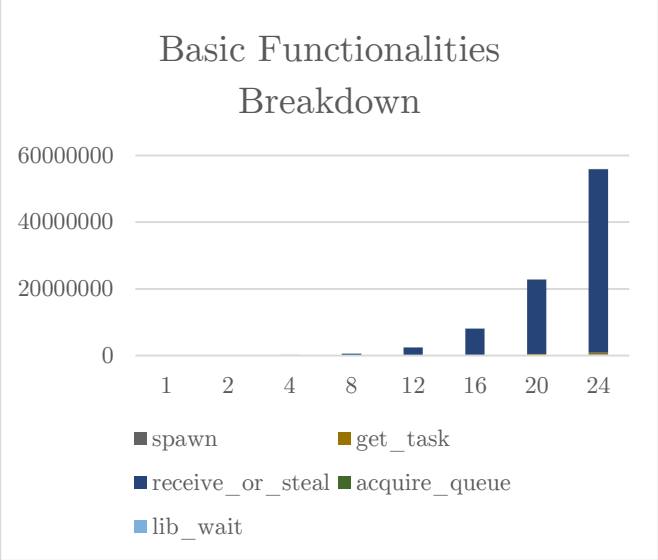

Figure 95. Quicksort basic functionalities breakdown on SMP for each thread count

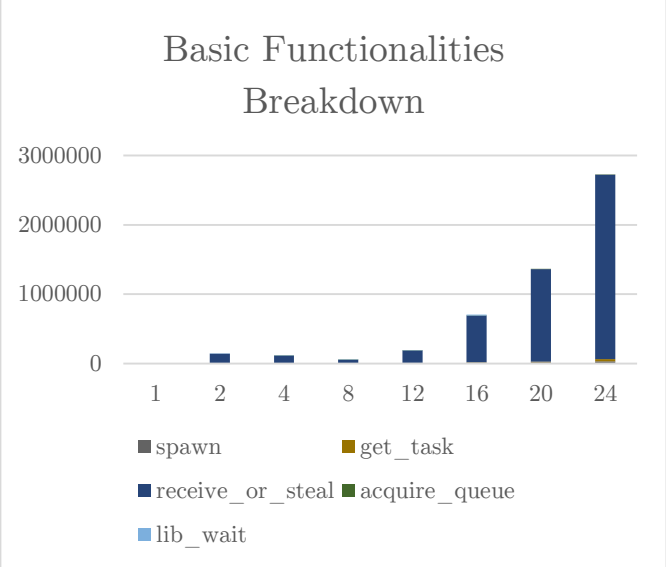

Figure 96. Quicksort basic functionalities breakdown on NUMA for each thread count

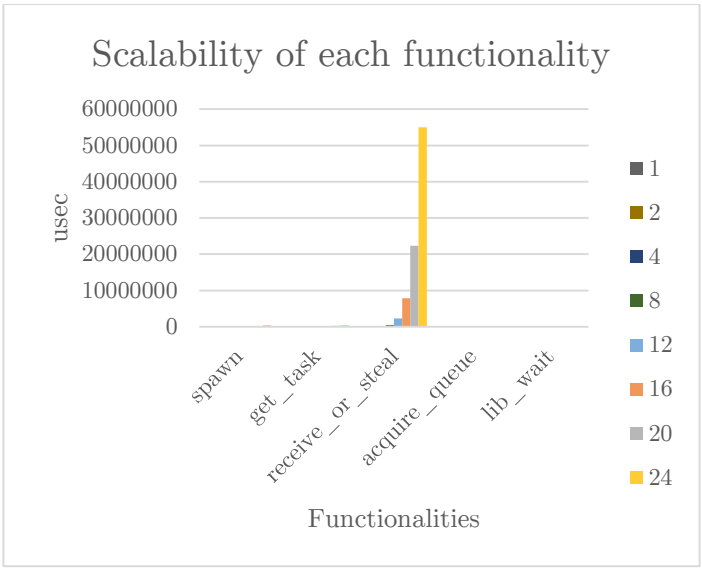

Figure 97. Quicksort basic functionalities' scalability on SMP for each thread count

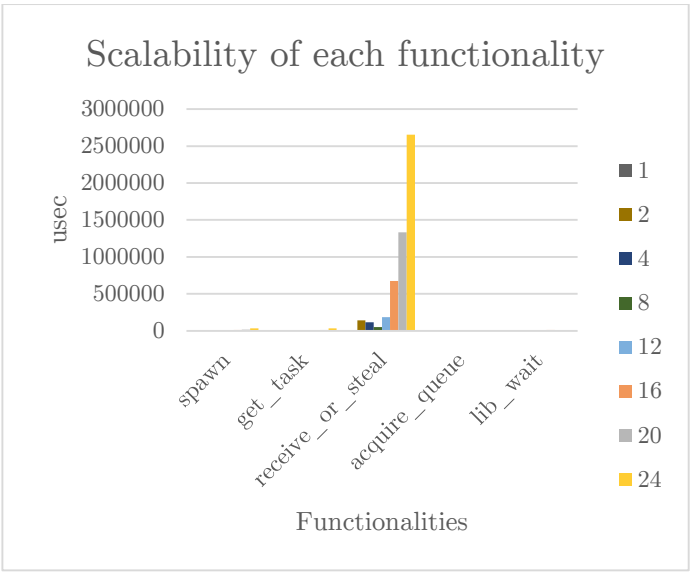

Figure 98. Quicksort basic functionalities' scalability on NUMA for each thread count

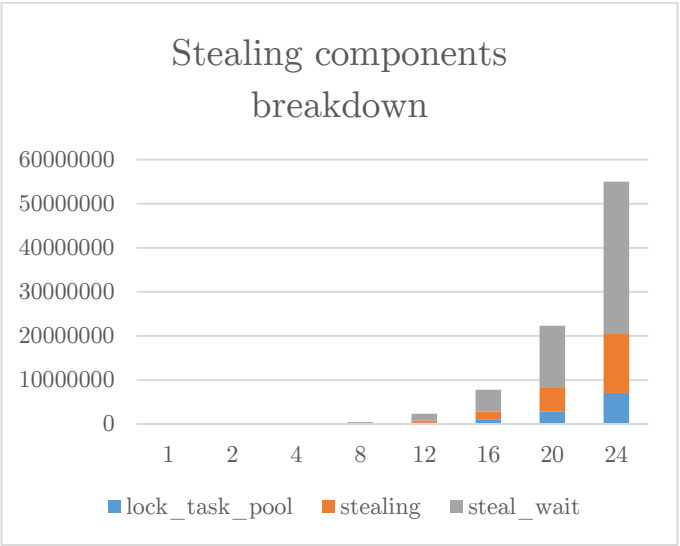

Figure 99. Quicksort stealing components breakdown on SMP for each thread count

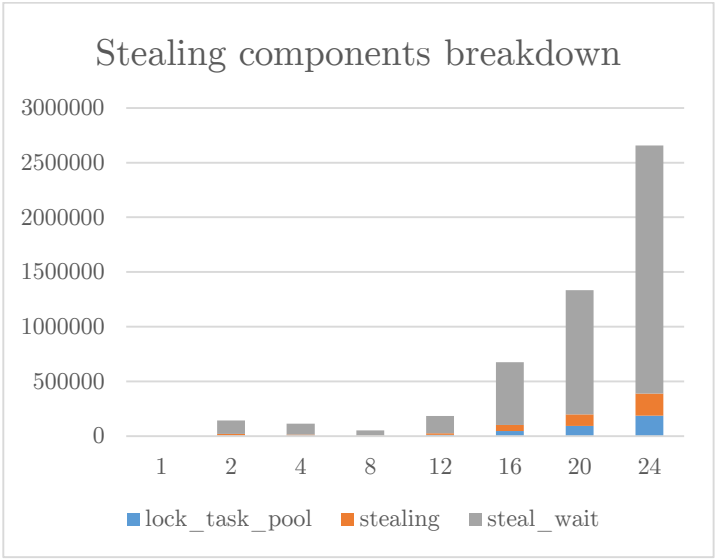

Figure 100. Quicksort stealing components breakdown on NUMA for each thread count

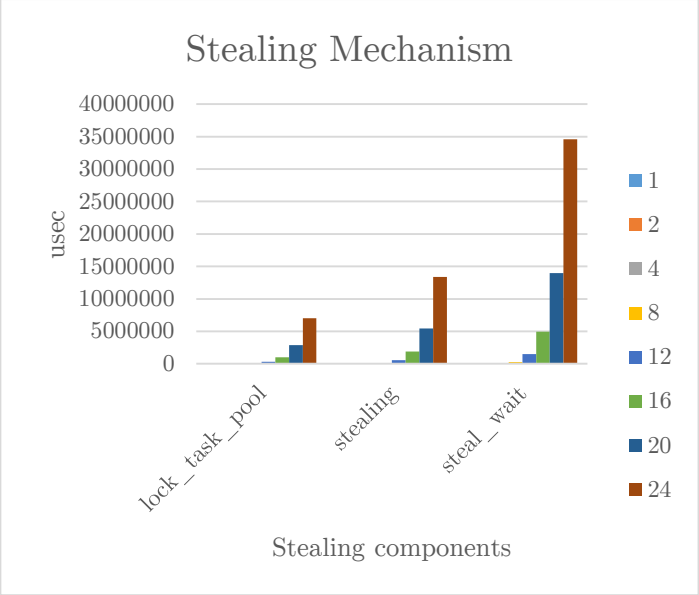

Figure 101. Quicksort scalability of stealing components on SMP for each thread count

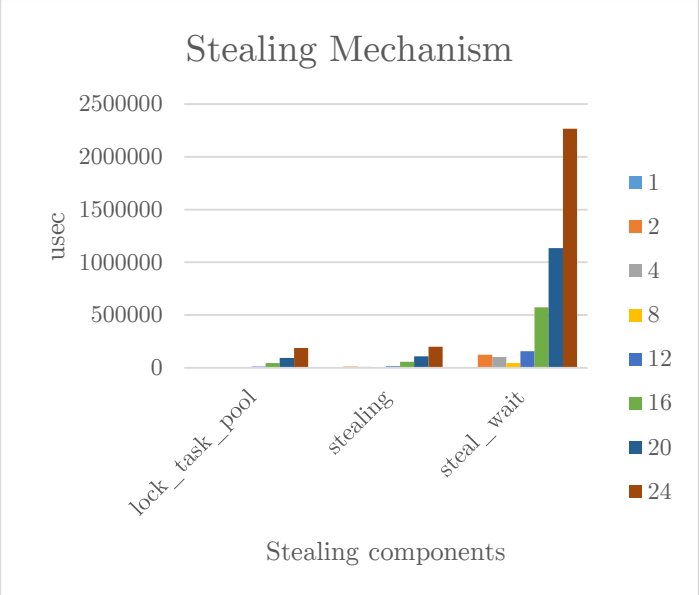

Figure 102. Quicksort scalability of stealing components on NUMA for each thread count

7.2 Swaptions

The same diagrams are given for the Swaptions application.

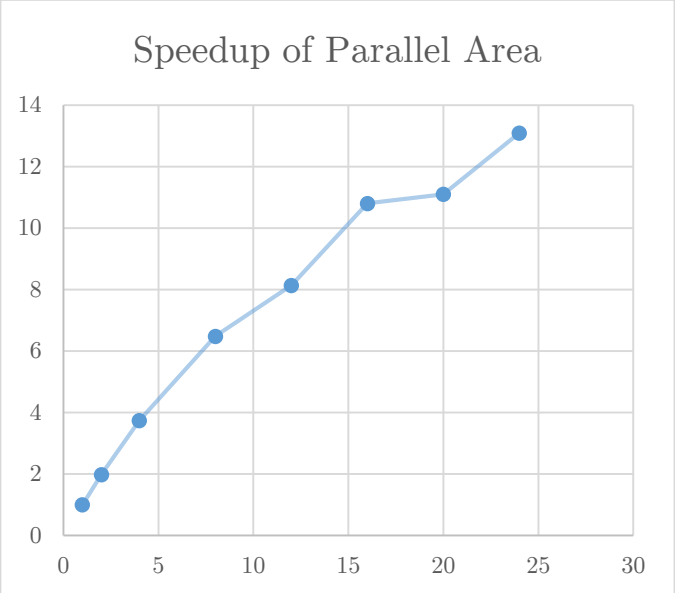

Figure 103. Swaptions speedup on SMP

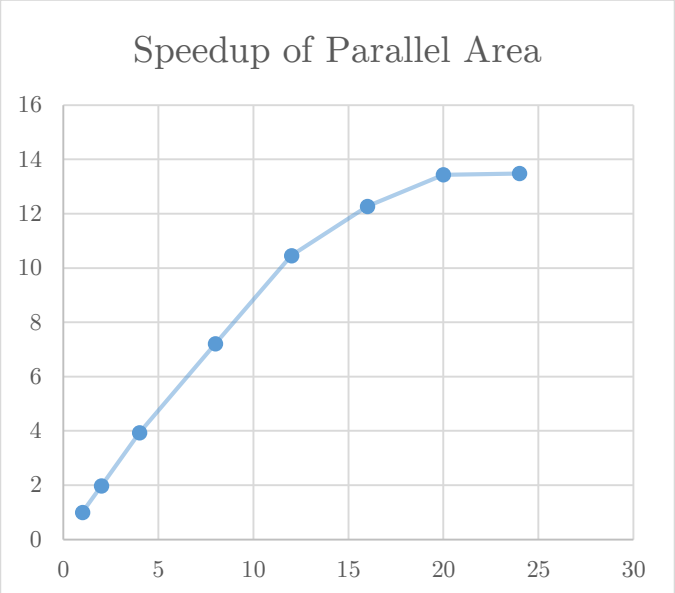

Figure 104. Swaptions speedup on NUMA

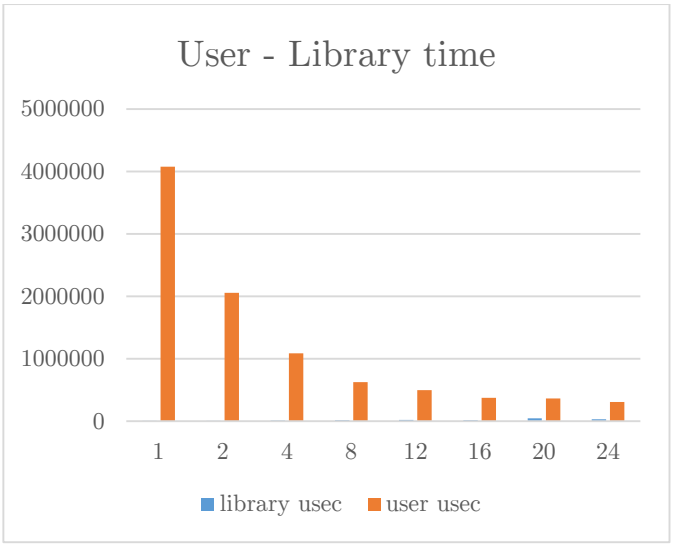

Figure 105. Swaptions User-Library time on SMP for each thread count

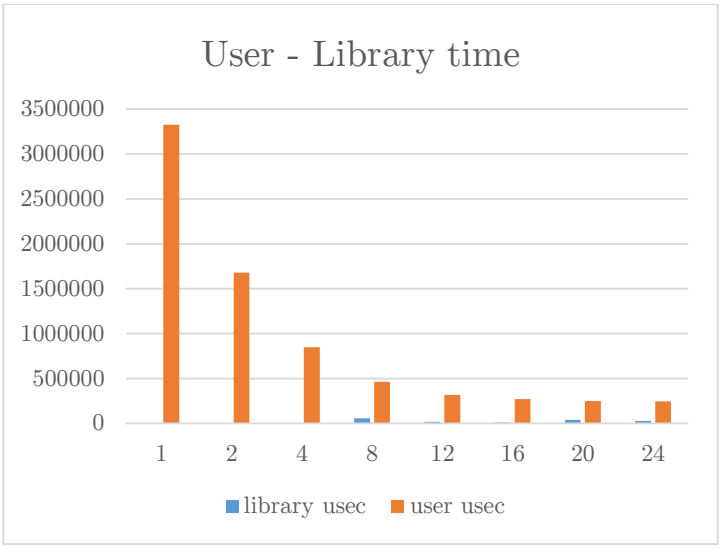

Figure 106. Swaptions User-Library time on NUMA for each thread count

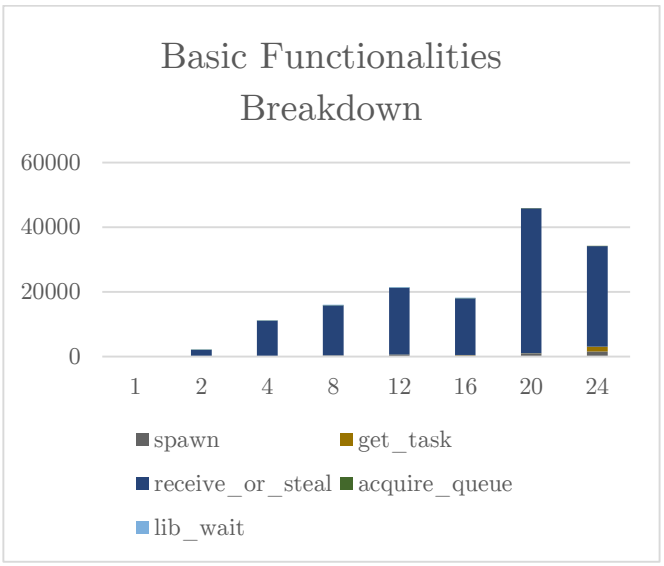

Figure 107. Swaptions basic functionalities breakdown on SMP for each thread count

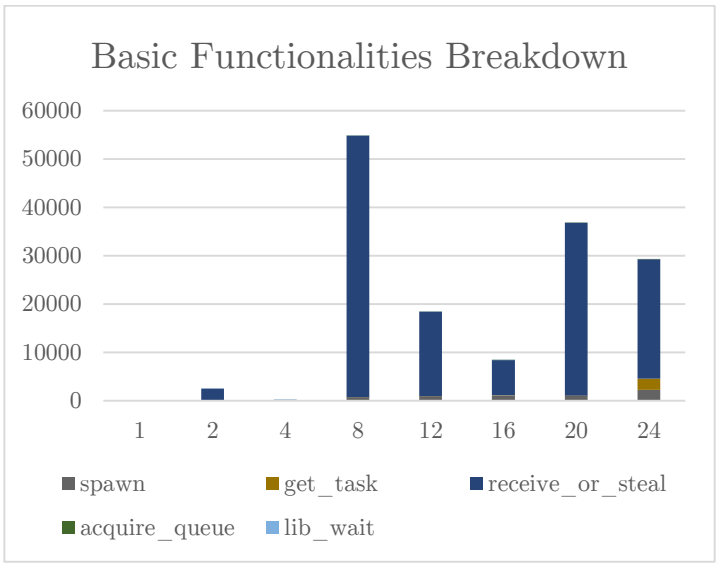

Figure 108. Swaptions basic functionalities breakdown on NUMA for each thread count

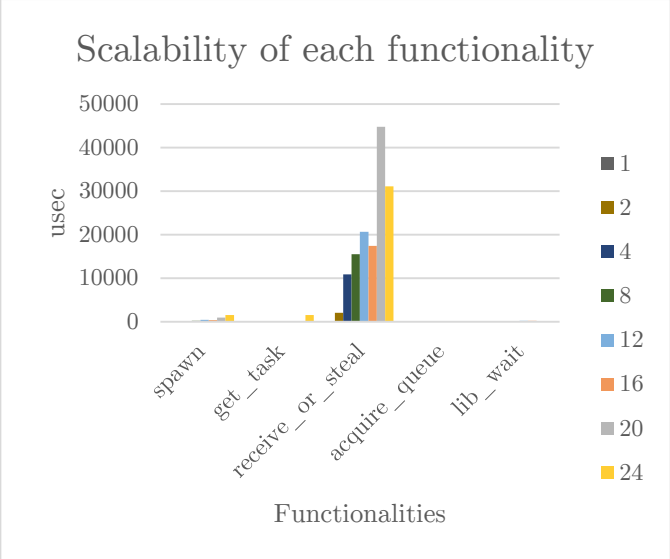

Figure 109. Swaptions basic functionalities' scalability on SMP for each thread count

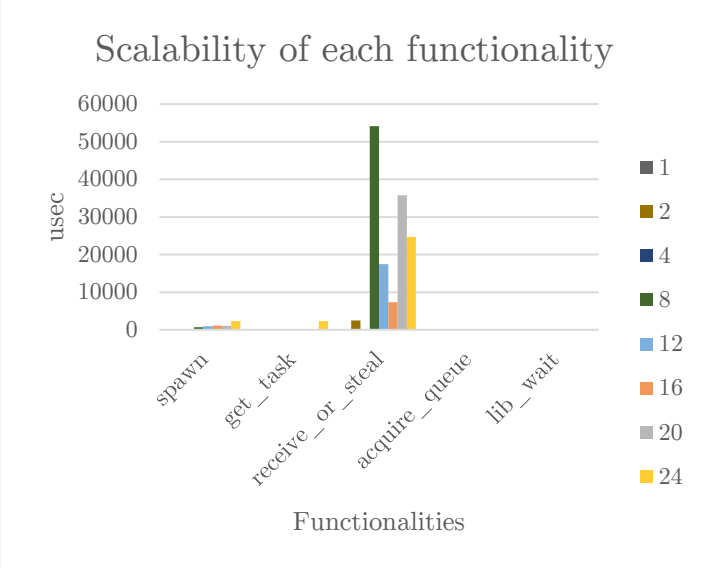

Figure 110. Swaptions basic functionalities' scalability on NUMA for each thread count

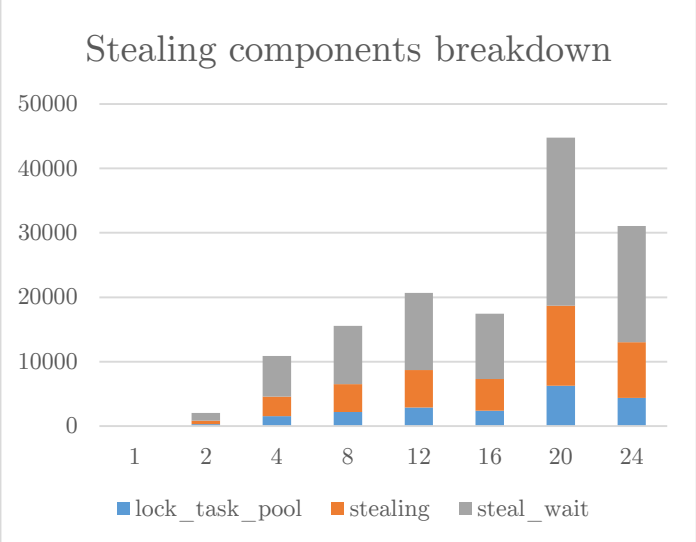

Figure 111. Swaptions stealing components breakdown on SMP for each thread count

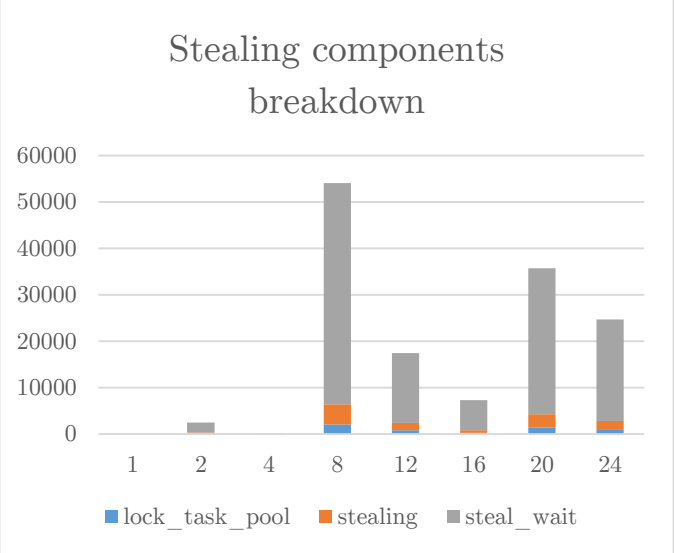

Figure 112. Swaptions stealing components breakdown on NUMA for each thread count

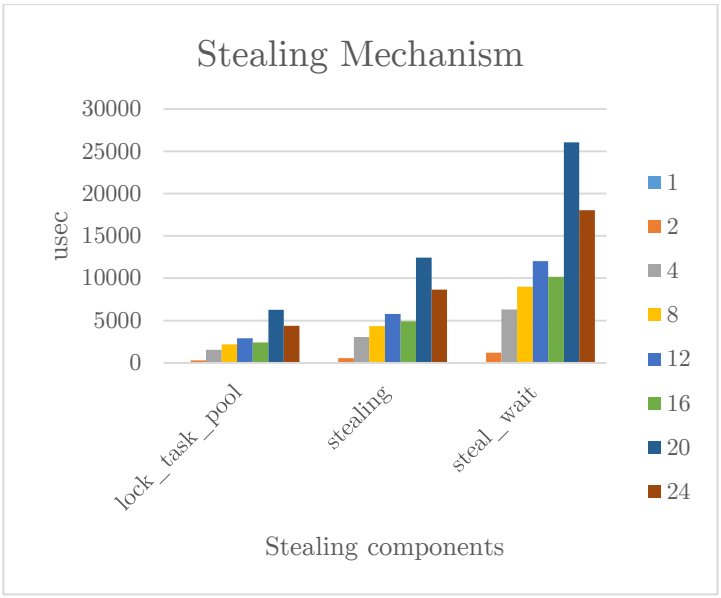

Figure 113. Swaptions scalability of stealing components on SMP for each thread count

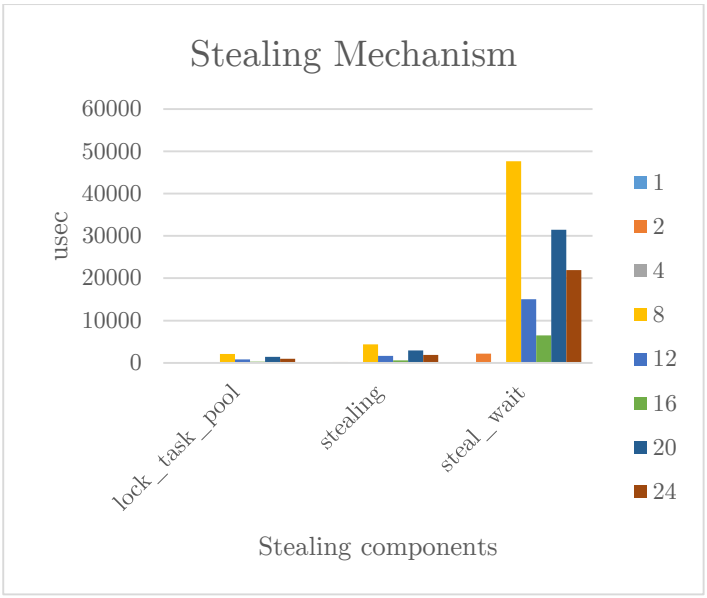

Figure 114. Swaptions scalability of stealing components on NUMA for each thread count

7.3 Matrix Multiplication

The same diagrams are given for the Matrix Multiplication application. We can see here that it scales linearly on the SMP but not as well on the NUMA when multithreading kicks in.

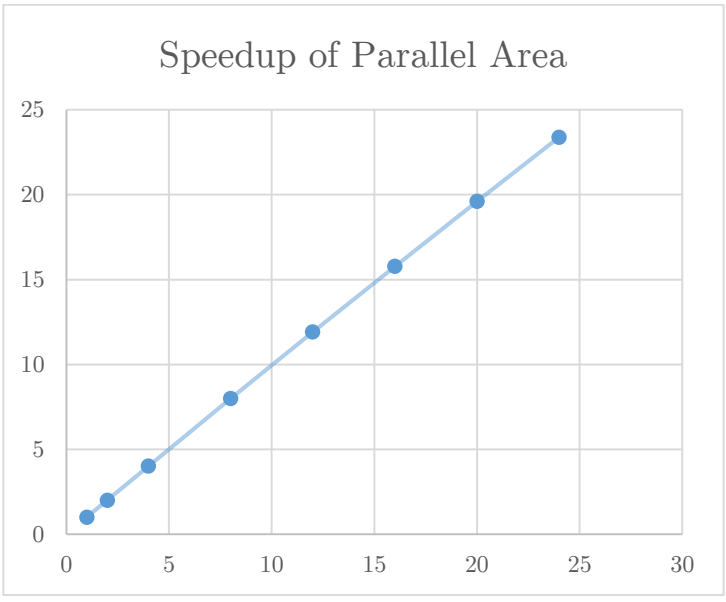

Figure 115. Matrix multiplication speedup on SMP

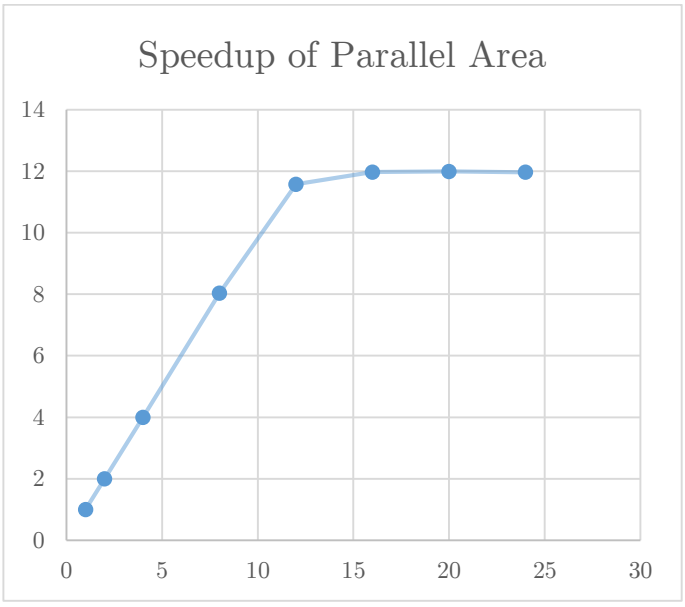

Figure 116. Matrix multiplication speedup on NUMA

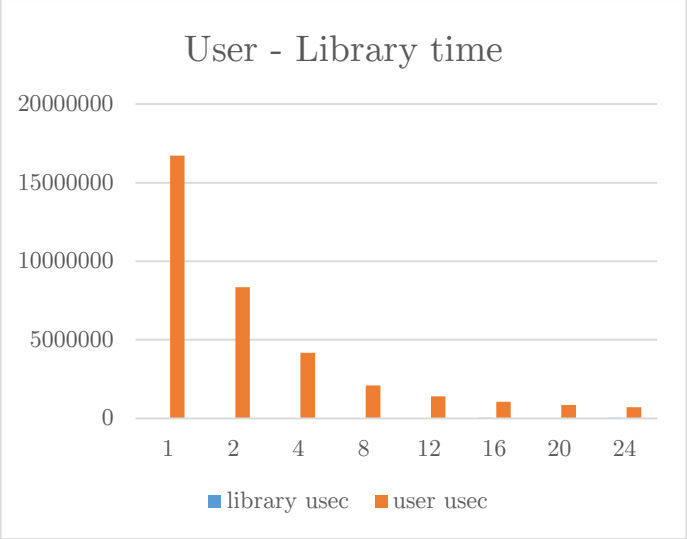

Figure 117. Matrix multiplication User-Library time on SMP for each thread count

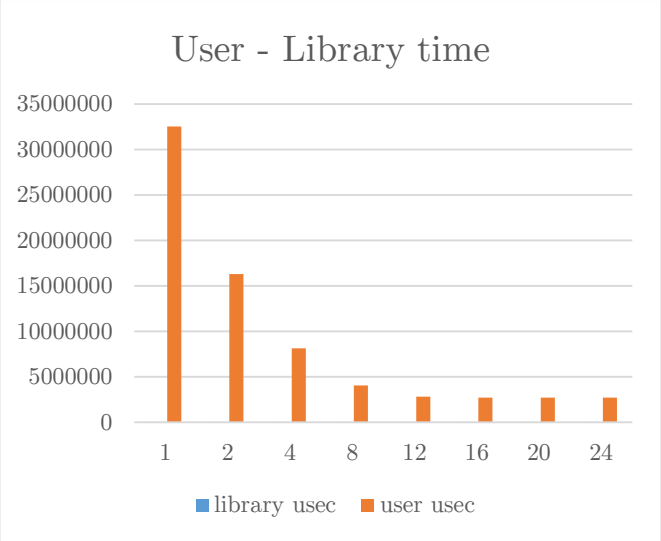

Figure 118. Matrix multiplication User-Library time on NUMA for each thread count

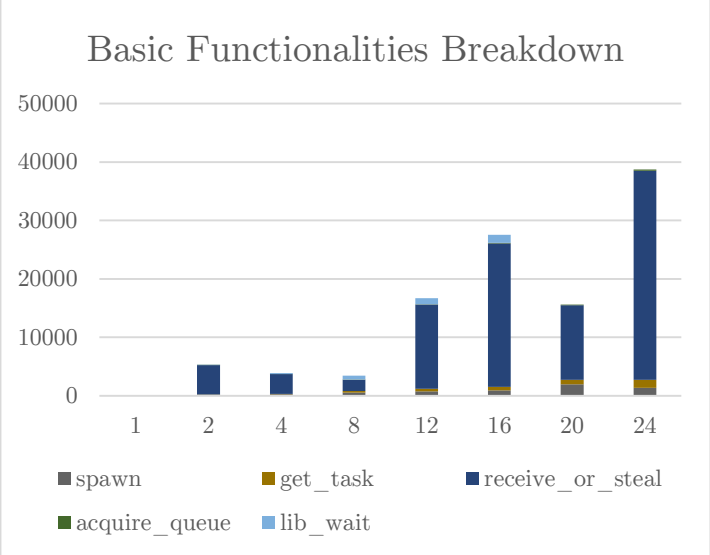

Figure 119. Matrix multiplication basic functionalities breakdown on SMP for each thread count

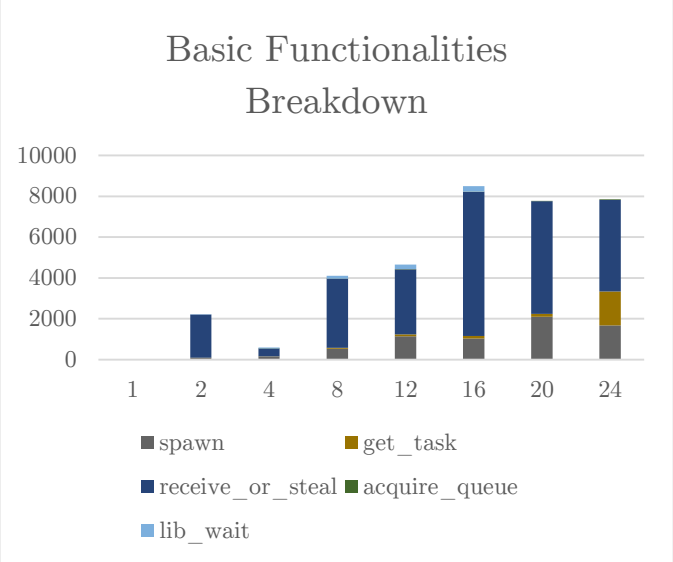

Figure 120. Matrix multiplication basic functionalities breakdown on NUMA for each thread count

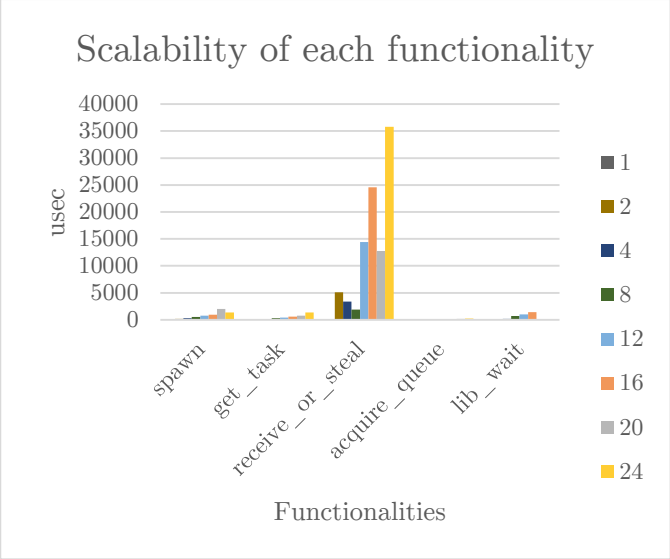

Figure 121. Matrix multiplication basic functionalities' scalability on SMP for each thread count

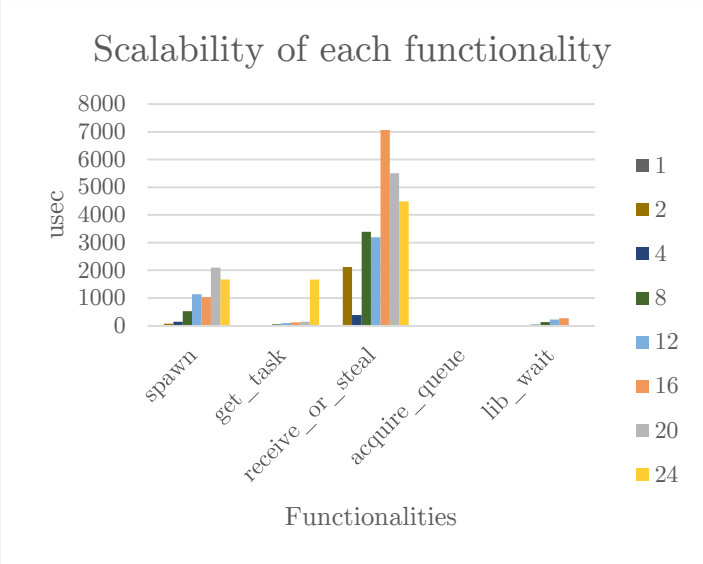

Figure 122. Matrix multiplication basic functionalities' scalability on NUMA for each thread count

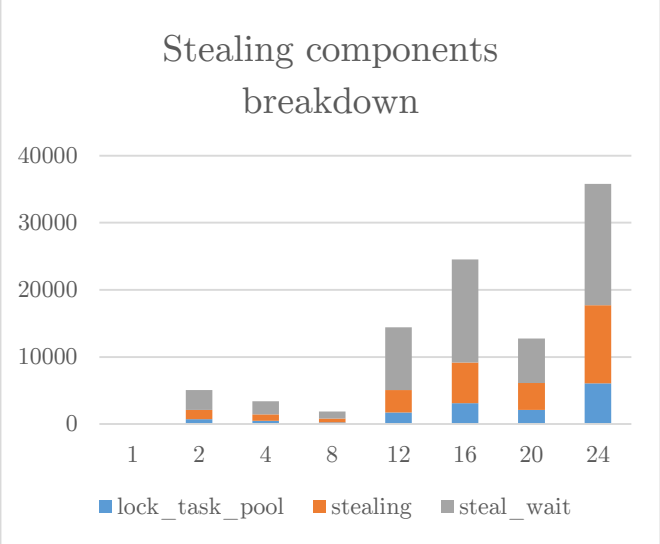

Figure 123. Matrix multiplication stealing components breakdown on SMP for each thread count

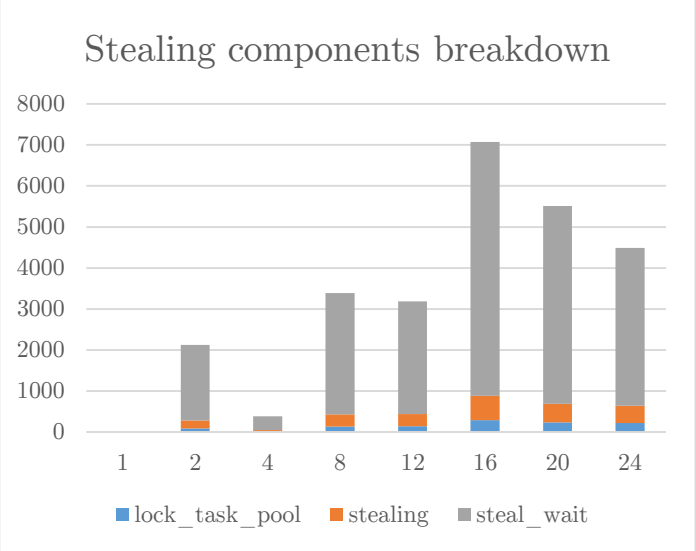

Figure 124. Matrix multiplication stealing components breakdown on NUMA for each thread count

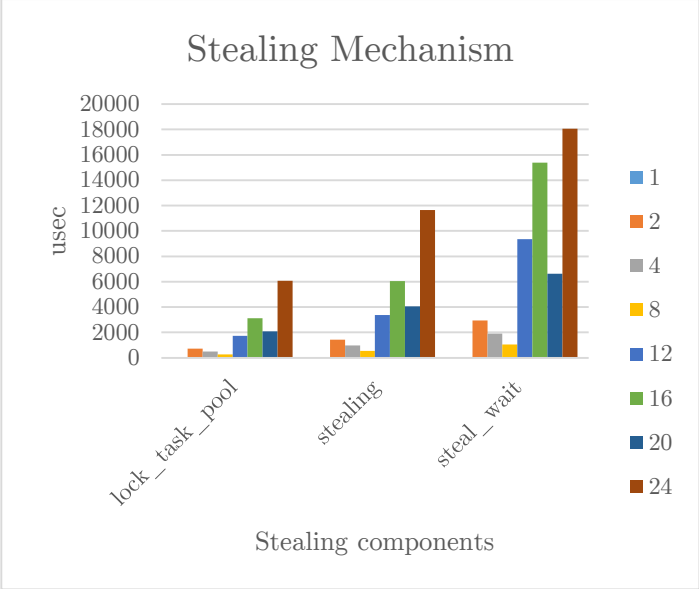

Figure 125. Matrix multiplication scalability of stealing components on SMP for each thread count

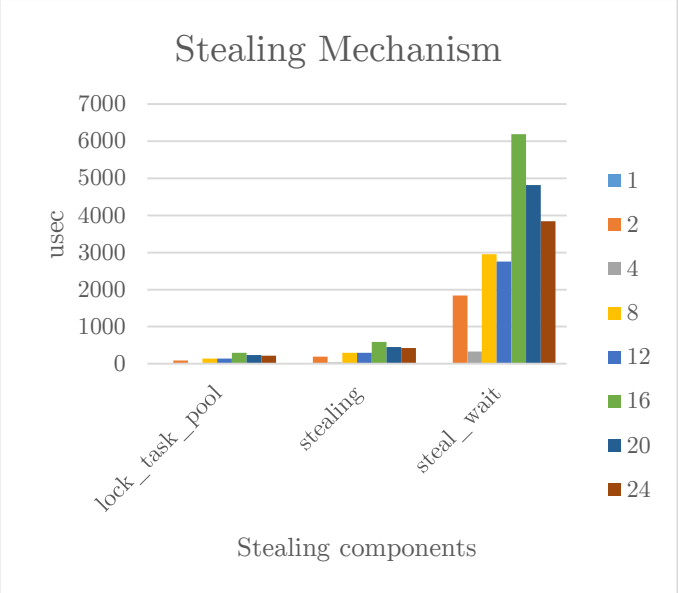

Figure 126. Matrix multiplication scalability of stealing components on NUMA for each thread count

7.4 Convex-hull

The same diagrams are given for the Convex-hull application, which is another application that scales better on the NUMA than the SMP.

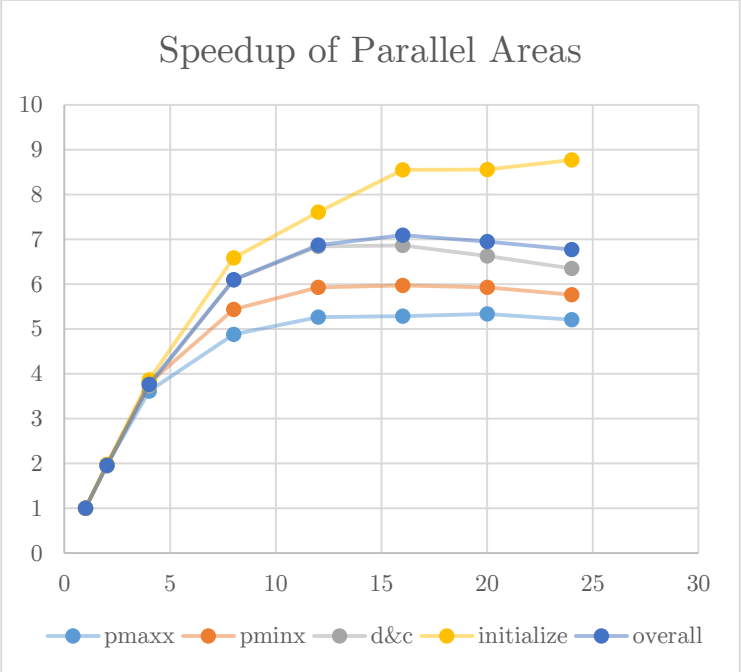

Figure 127. Convex Hull speedup on SMP

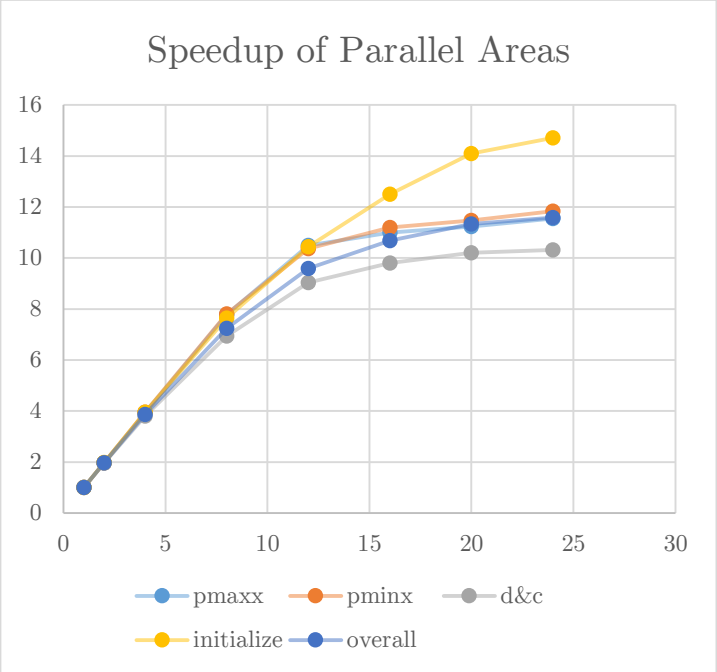

Figure 128. Convex Hull speedup on NUMA

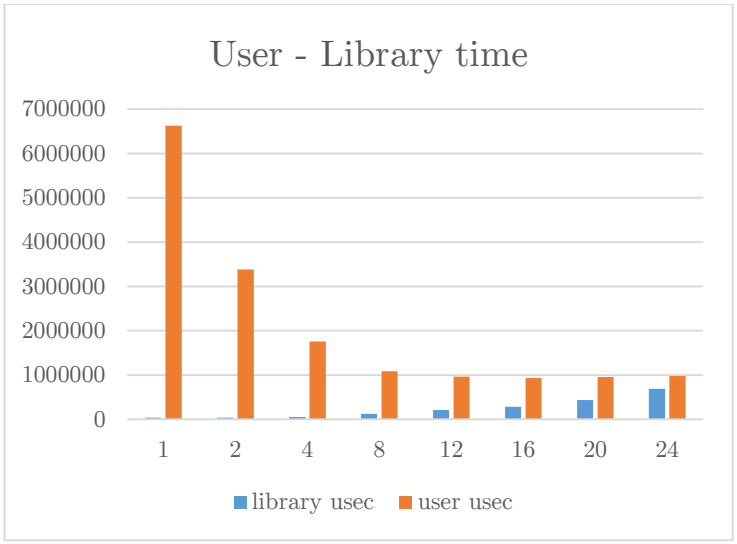

Figure 129. Convex Hull User-Library time on SMP for each thread count

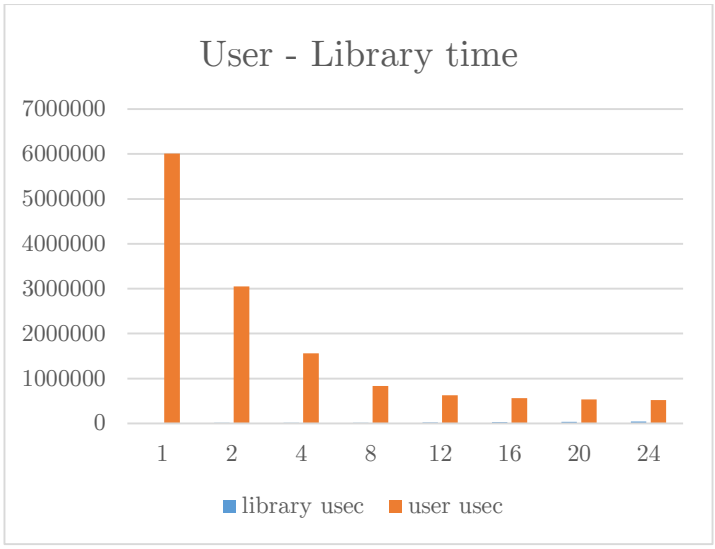

Figure 130. Convex Hull User-Library time on NUMA for each thread count

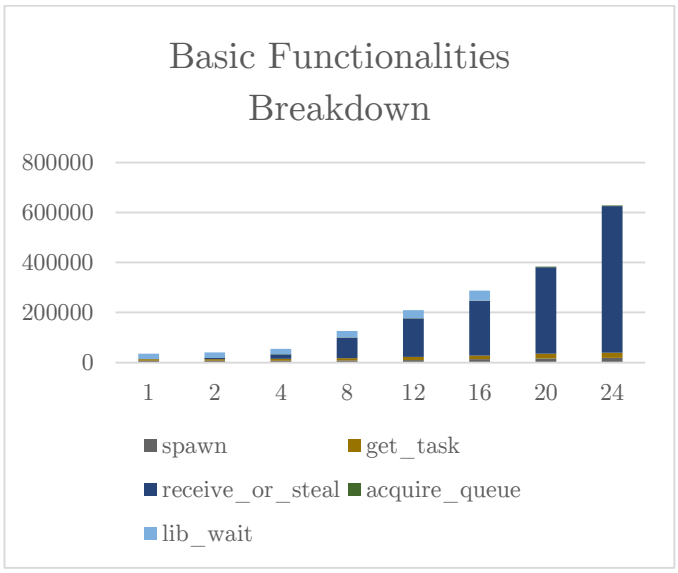

Figure 131. Convex Hull basic functionalities breakdown on SMP for each thread count

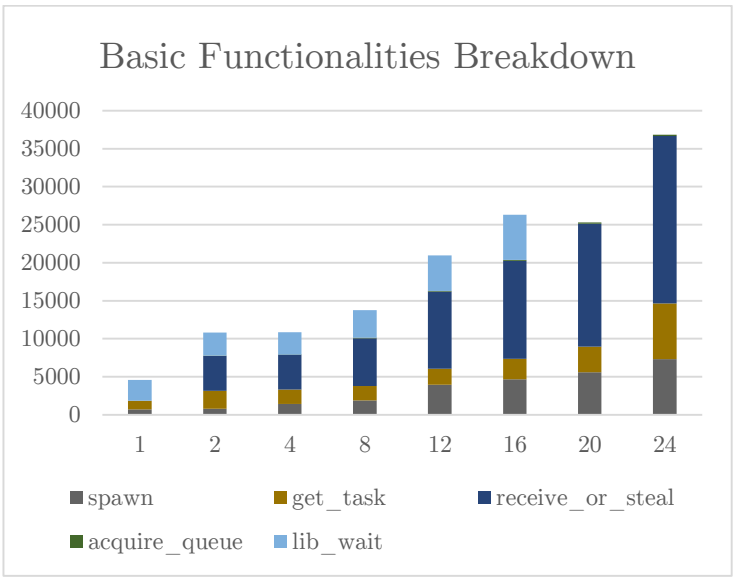

Figure 132. Convex Hull basic functionalities breakdown on NUMA for each thread count

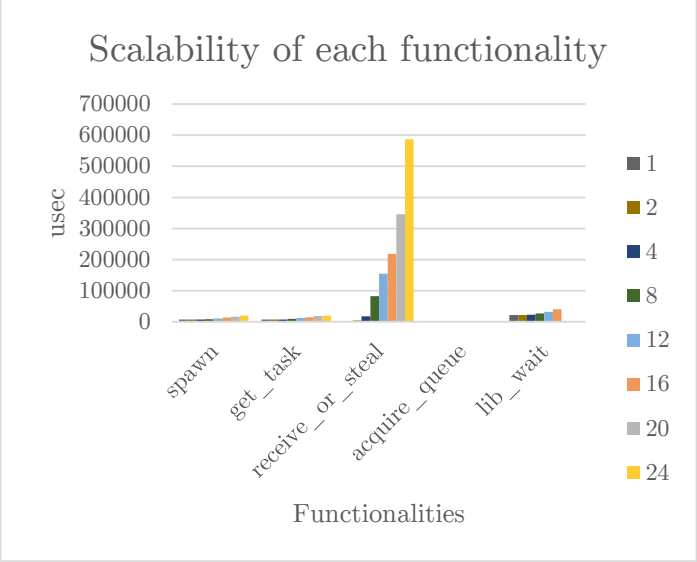

Figure 133. Convex Hull basic functionalities' scalability on SMP for each thread count

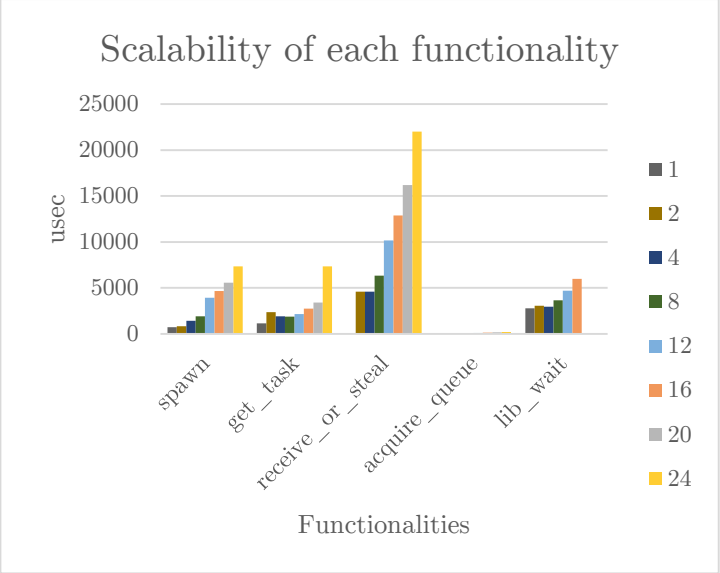

Figure 134. Convex Hull basic functionalities' scalability on NUMA for each thread count

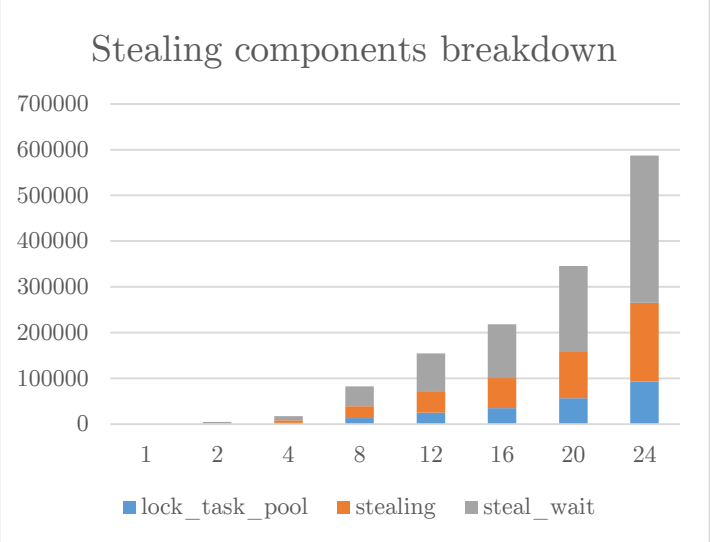

Figure 135. Convex Hull stealing components breakdown on SMP for each thread count

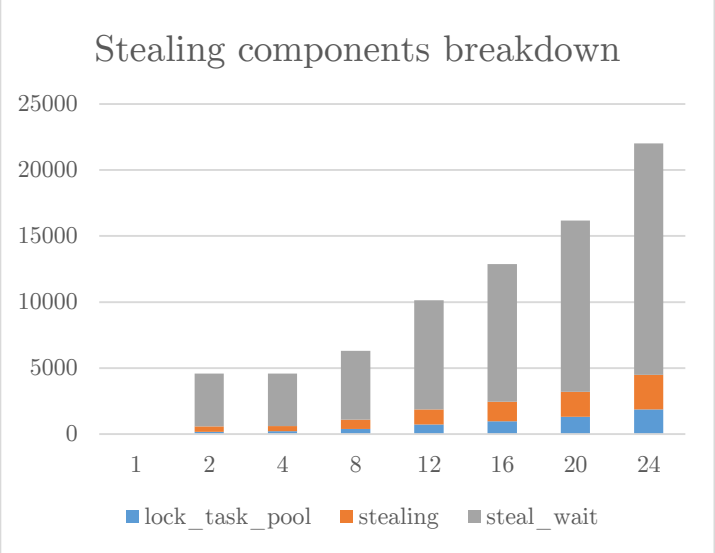

Figure 136. Convex Hull stealing components breakdown on NUMA for each thread count

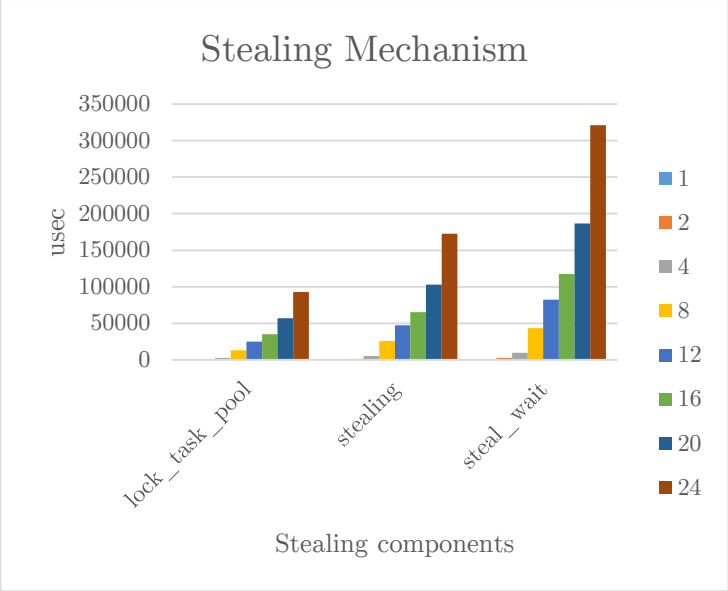

Figure 137. Convex Hull scalability of stealing components on SMP for each thread count

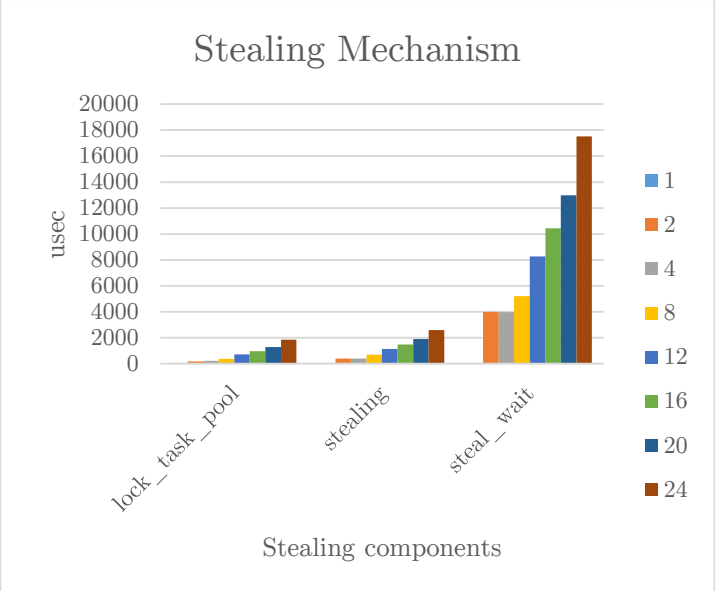

Figure 138. Convex Hull scalability of stealing components on NUMA for each thread count

Chapter 8
Appendix B – Evaluation Results

8.1 Cache-Aware Techniques

8.1.1 Heat

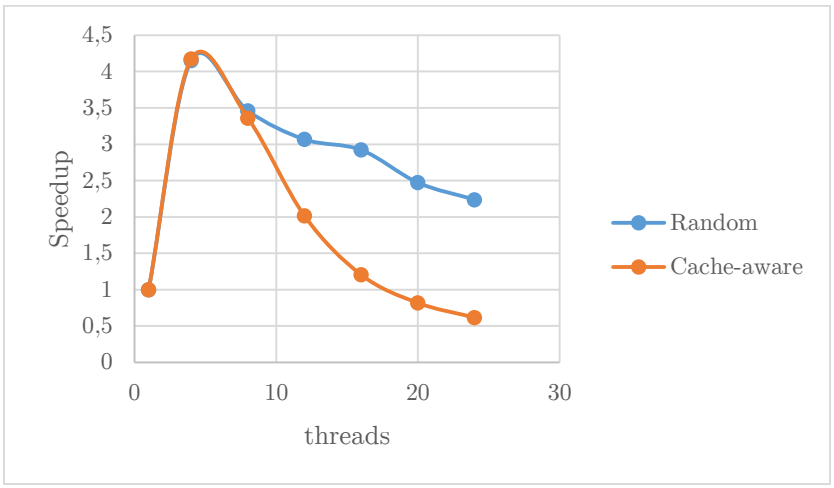

Figure 139. Speedup of 5-point Heat on Dunnington(Cache-aware)

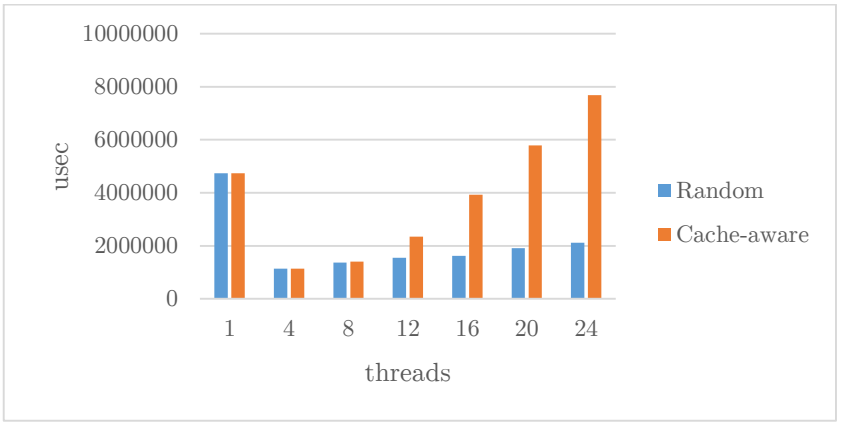

Figure 140. Execution Times of 5-point Heat on Dunnington(Cache-aware)

8.1.2 Gauss elimination

As we can see from the following figures, Gauss Elimination did not benefit from the cache-aware approach on Dunnington and Termi, in terms of execution time, although it achieved greater speedup on Termi. On Sandman there was performance improvement up to 3% on individual thread counts.

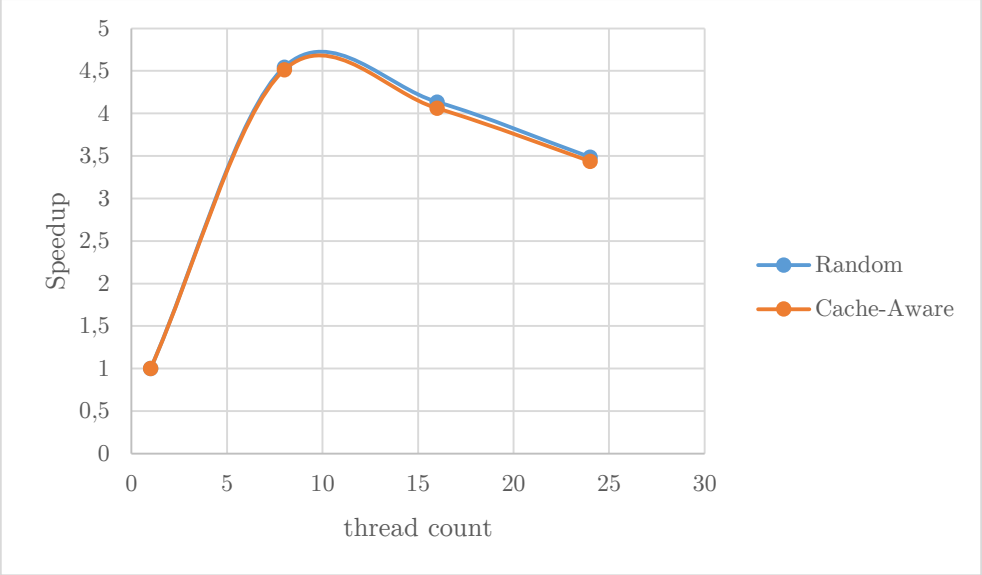

Figure 141. Gauss elimination Speedup on Dunnington(Cache-aware)

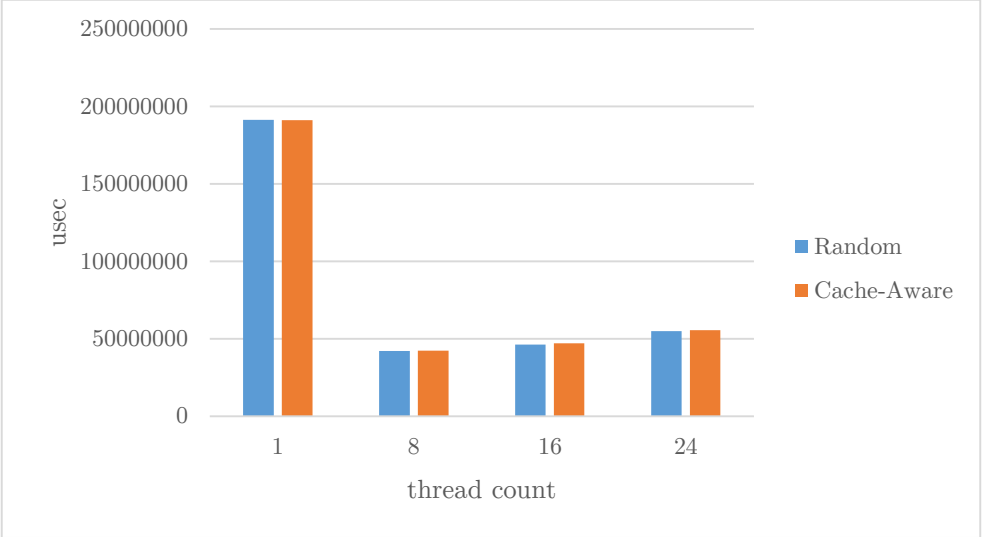

Figure 142. Gauss elimination execution times on Dunnington(Cache-aware)

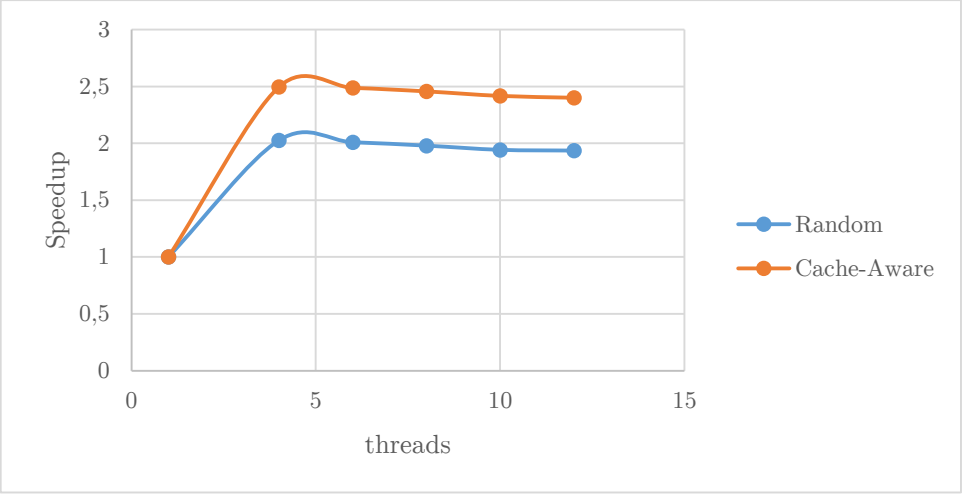

Figure 143. Gauss elimination speedup on Termi(Cache-aware)

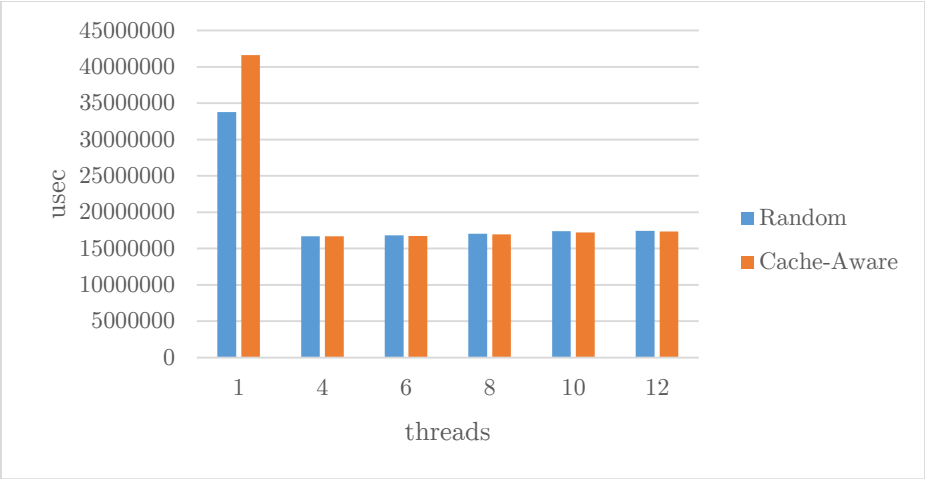

Figure 144. Gauss elimination execution times on Termi(Cache-aware)

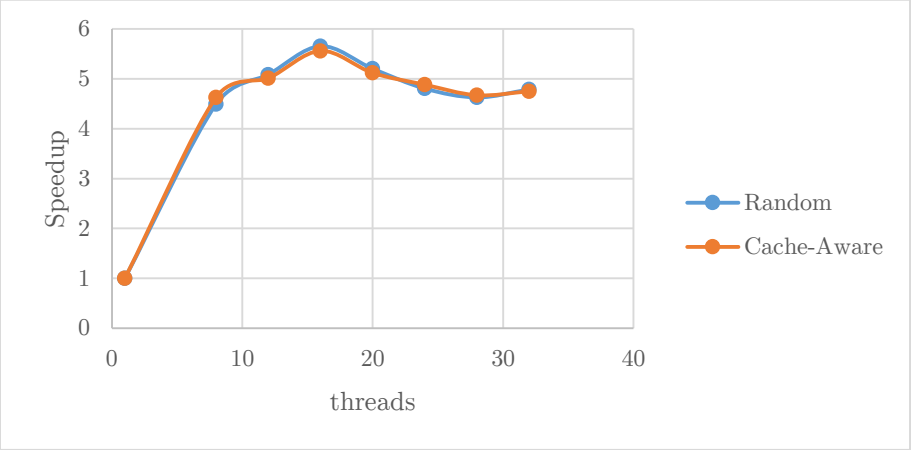

Figure 145. Gauss elimination speedup on Sandman(Cache-aware)

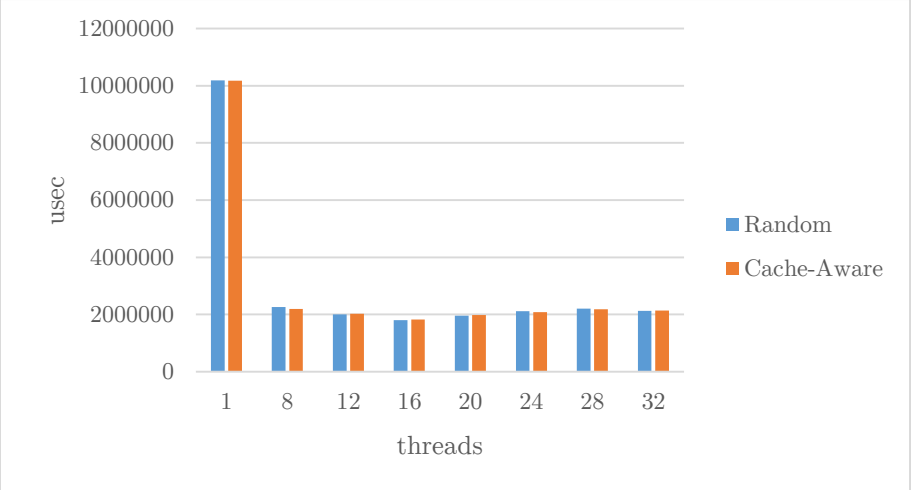

Figure 146. Gauss elimination execution times on Sandman(Cache-aware)

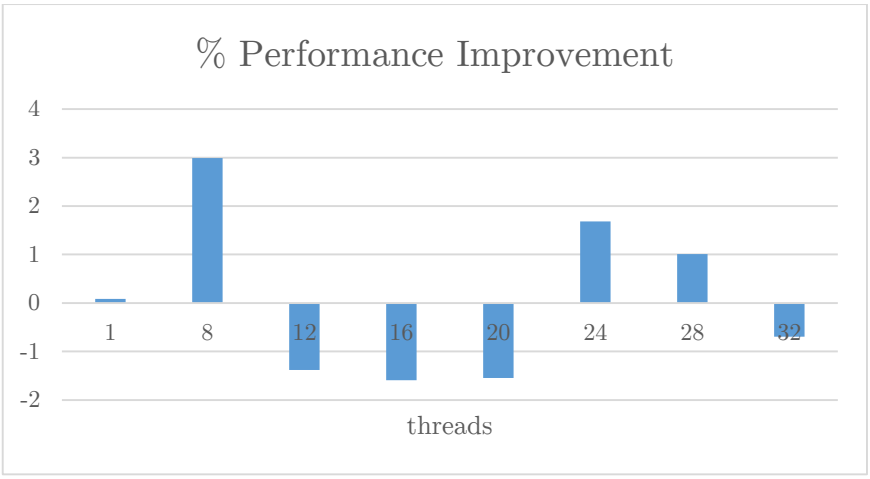

Figure 147. Gauss elimination % performance improvement on Sandman(Cache-aware)

8.1.3 Floyd-Warshall

It is obvious form the following figures that the *Floyd-Warshall* algorithm did not benefit from the cache-aware approach on any machine, neither did it suffer from performance degradation.

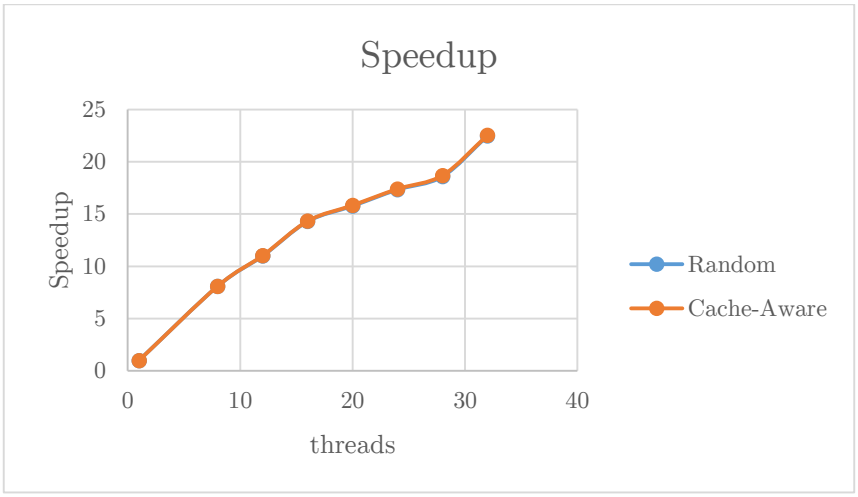

Figure 148. Floyd-Warshall speedup on Sandman(Cache-aware)

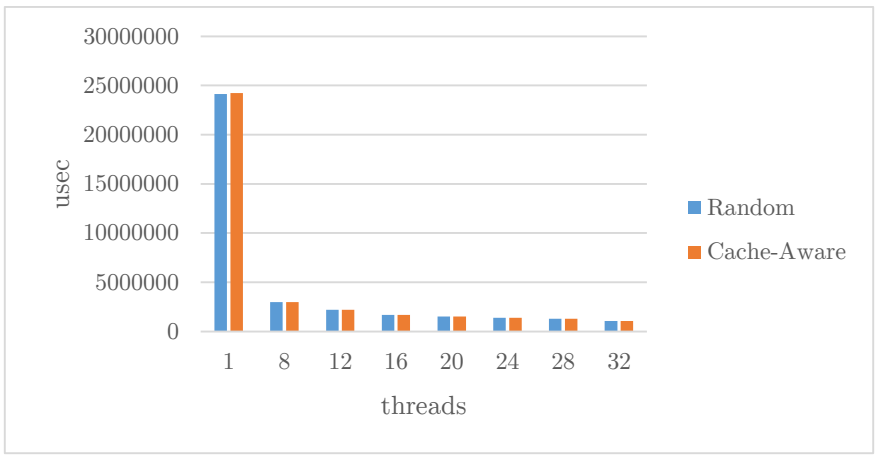

Figure 149. Floyd-Warshall execution times on Sandman(Cache-aware)

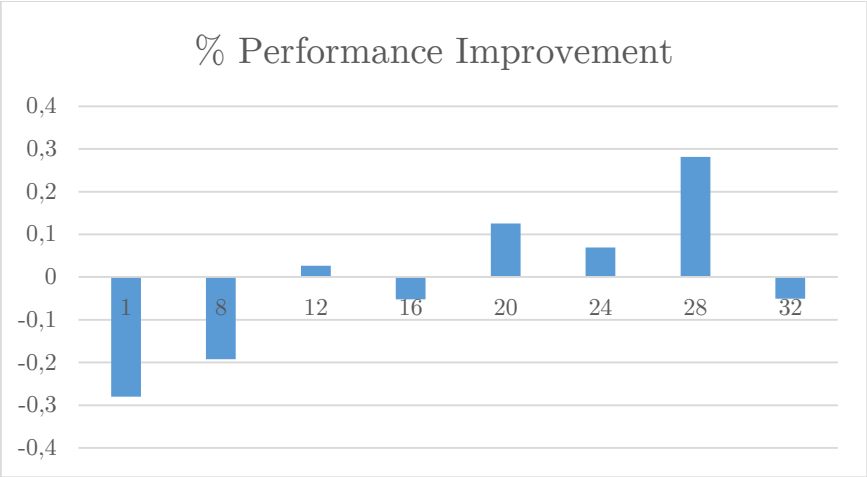

Figure 150. Floyd-Warshall performance gains on Sandman(Cache-aware)

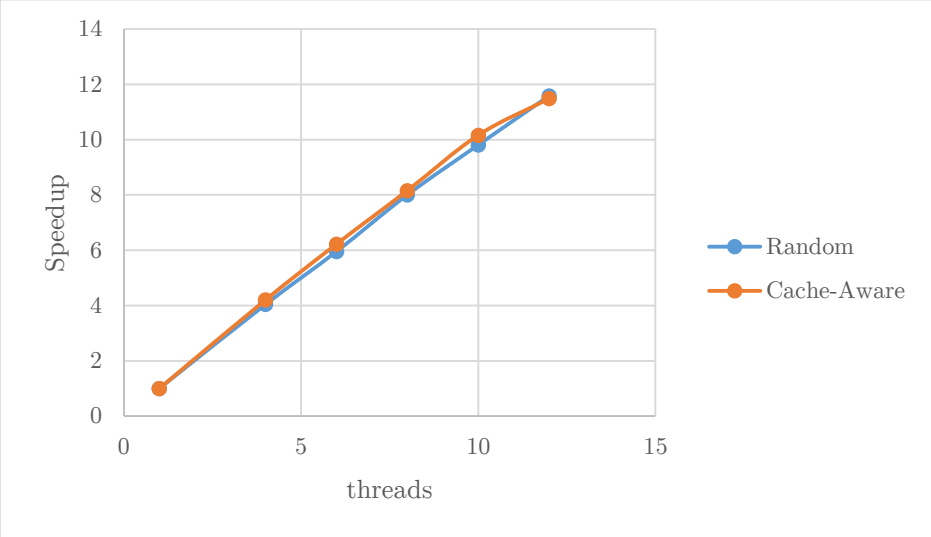

Figure 151. Floyd-Warshall speedup on Termi(Cache-aware)

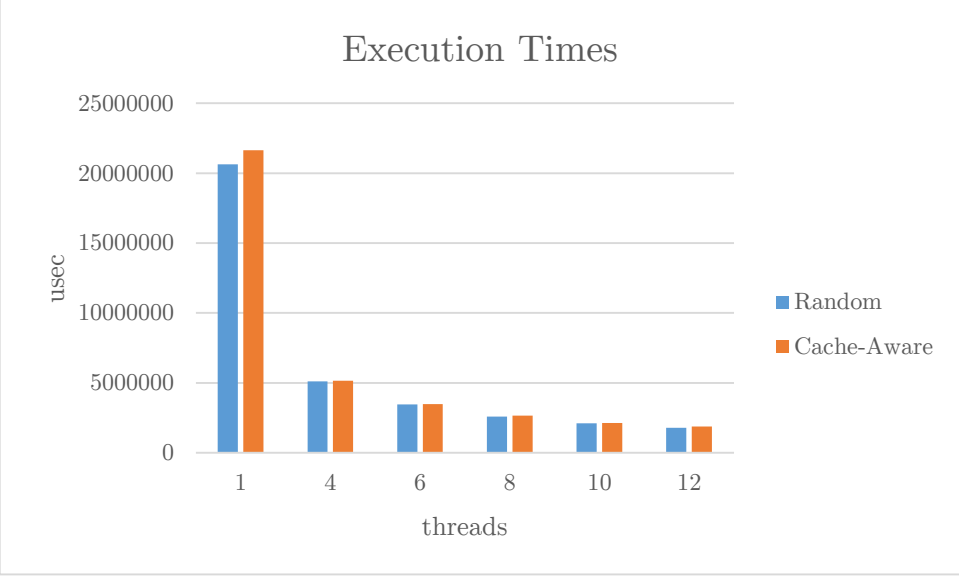

Figure 152. Floyd-Warshall execution times on Termi(Cache-aware)

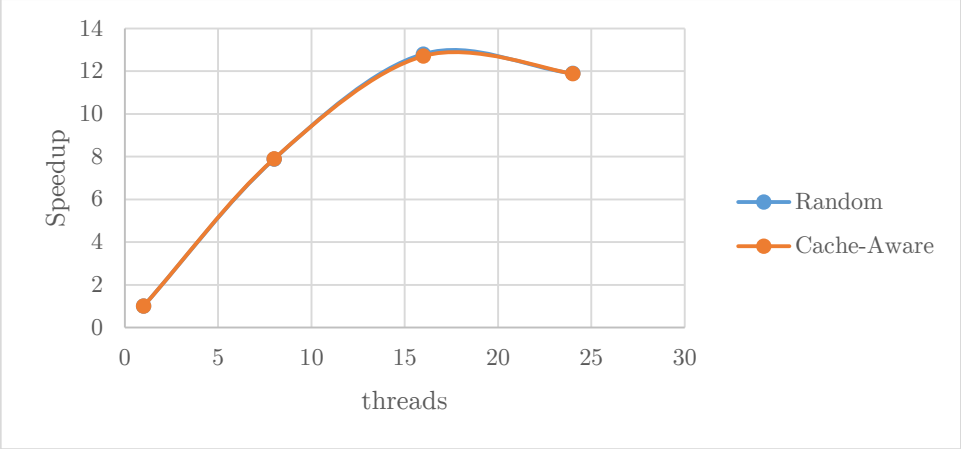

Figure 153. Floyd-Warshall speedup on Dunnington(Cache-aware)

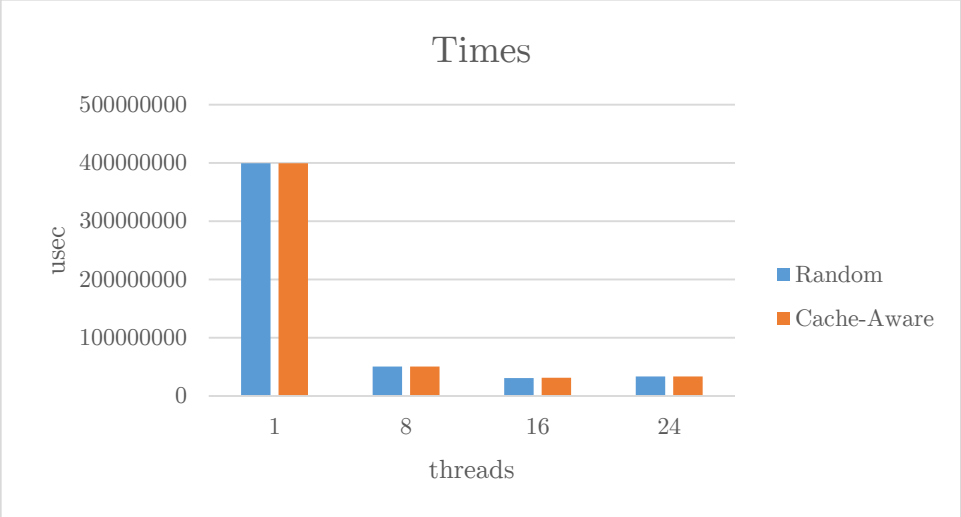

Figure 154. Floyd-Warshall execution times on Dunnington(Cache-aware)

8.1.4 Quicksort

Quicksort did not benefit from the cache-aware approach to work stealing, as shown in the following figures. The main reason for this is that the overhead of the mechanism we implemented dominates the benefits for the access pattern of this application.

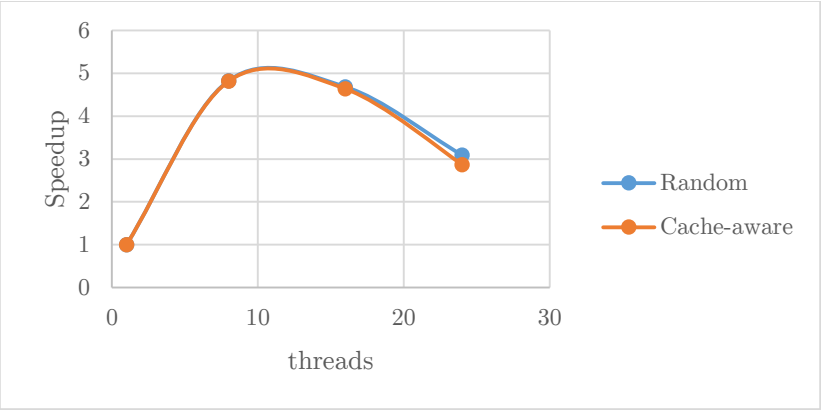

Figure 155. Quicksort speedup on Dunnington(Cache-aware)

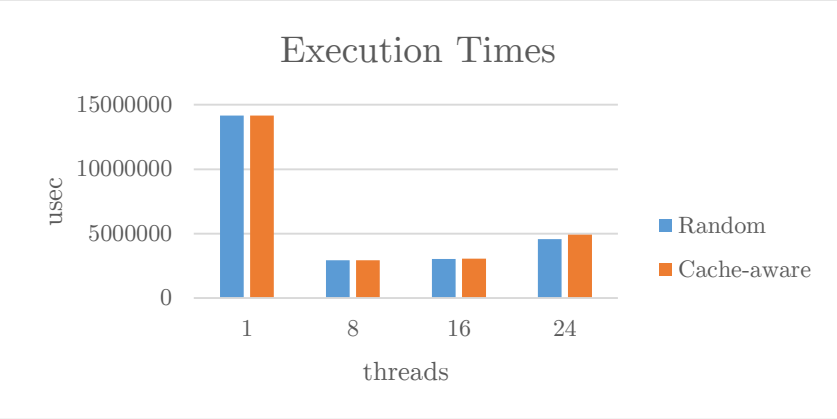

Figure 156. Quicksort execution times on Dunnington(Cache-aware)

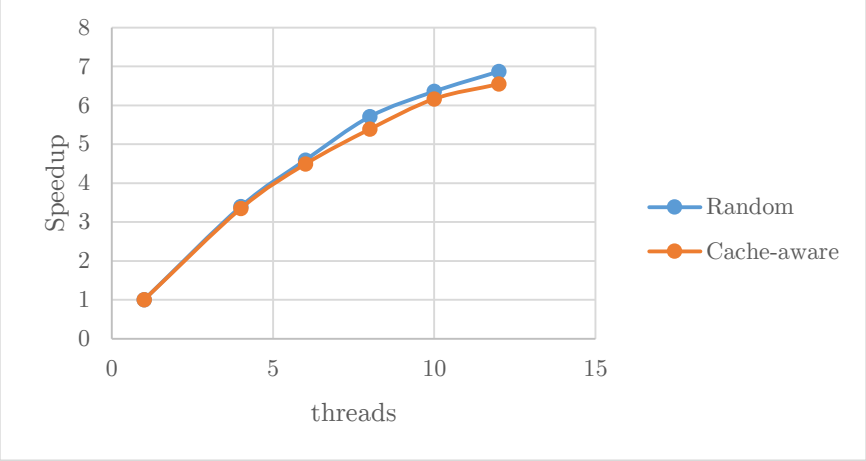

Figure 157. Quicksort speedup on Termi(Cache-aware)

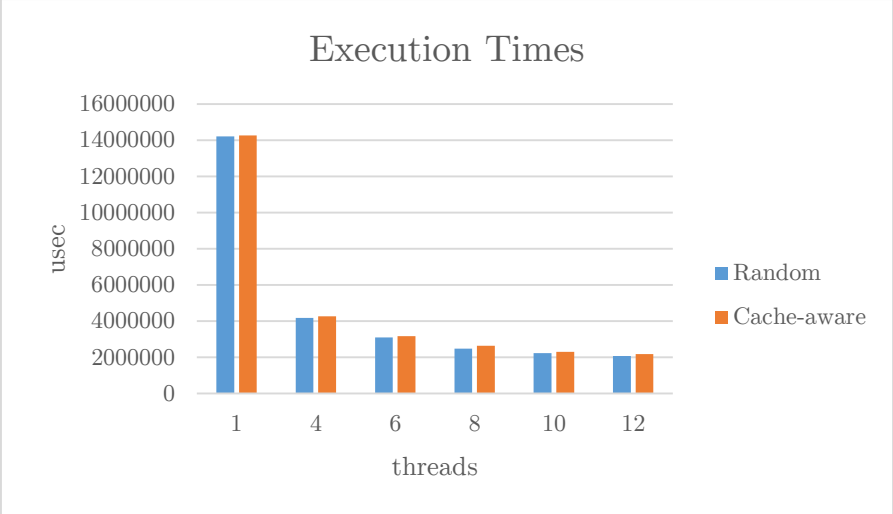

Figure 158. Quicksort execution times on Termi(Cache-aware)

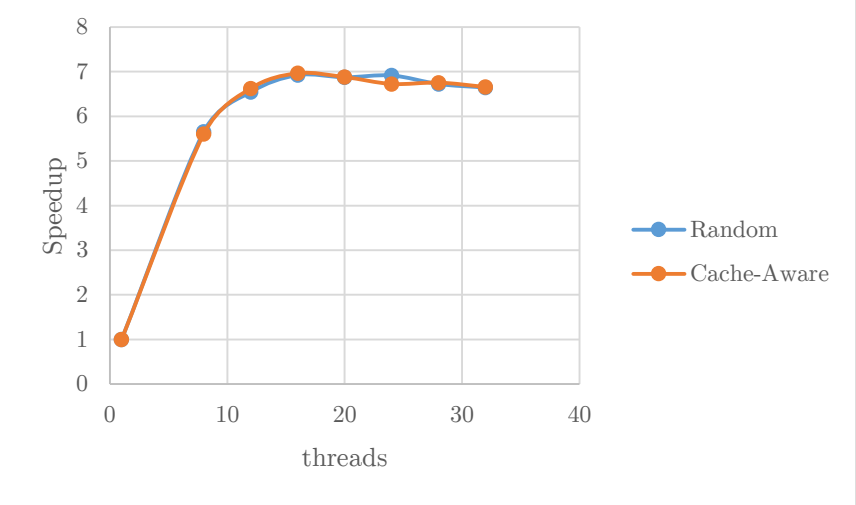

Figure 159. Quicksort speedup on Sandman(Cache-aware)

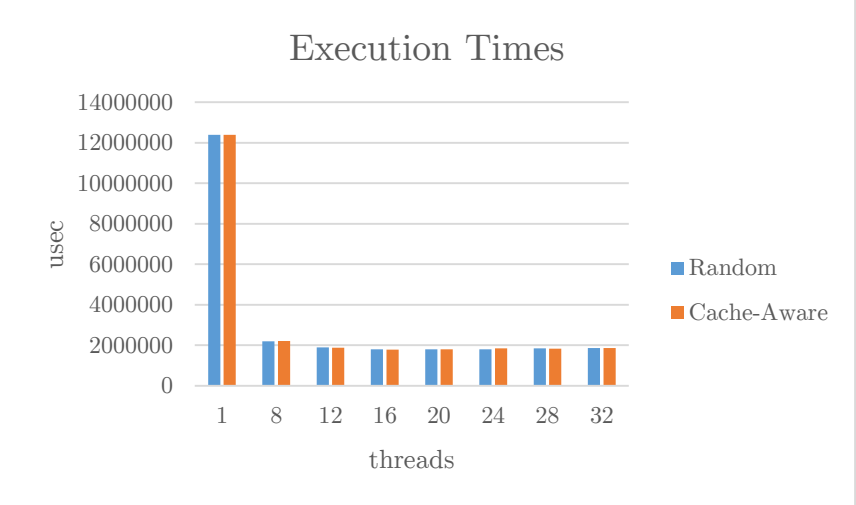

Figure 160. Quicksort execution times on Sandman(Cache-aware)

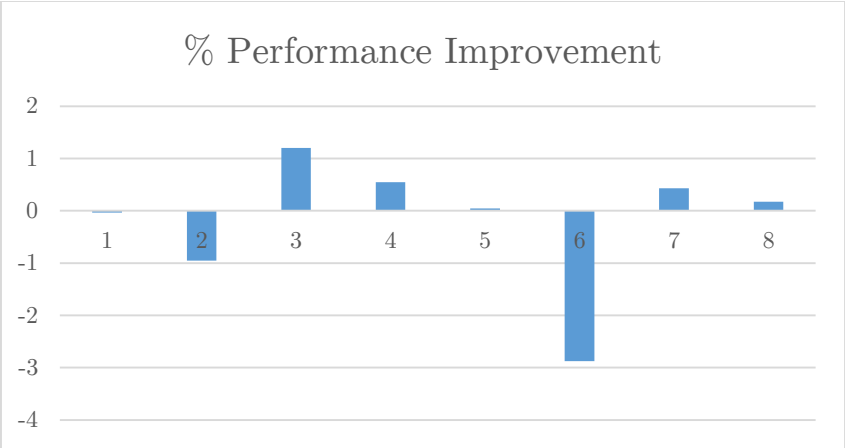

Figure 161. Quicksort performance benefits on Sandman(Cache-aware)

8.1.5 Matrix Multiplication

Matrix multiplication achieved an almost linear speedup even with the random stealing approach. As it can be seen in the following diagrams, our cache-aware mechanism did not cause performance degradation for this application, maintaining its excellent scaling.

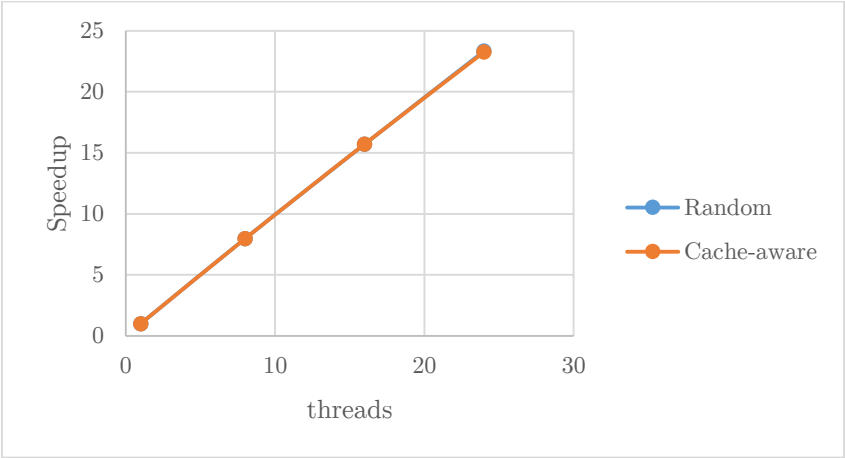

Figure 162. Matrix multiplication speedup on Dunnington(Cache-aware)

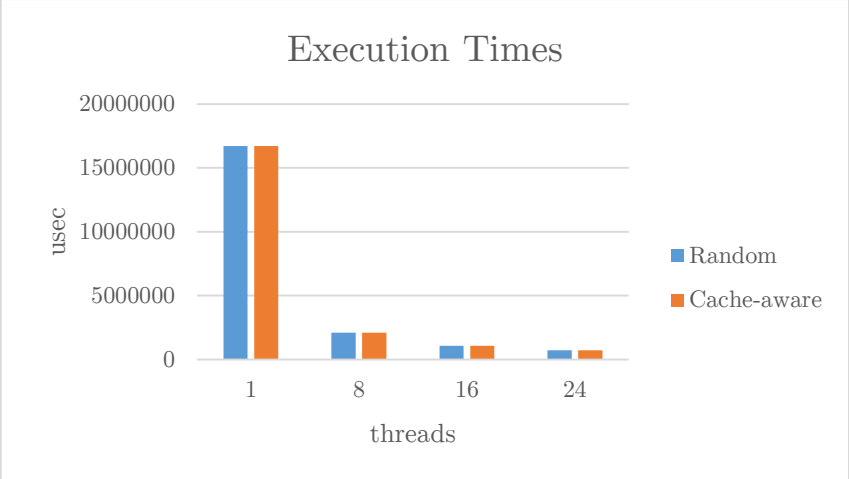

Figure 163. Matrix multiplication execution times on Dunnington(Cache-aware)

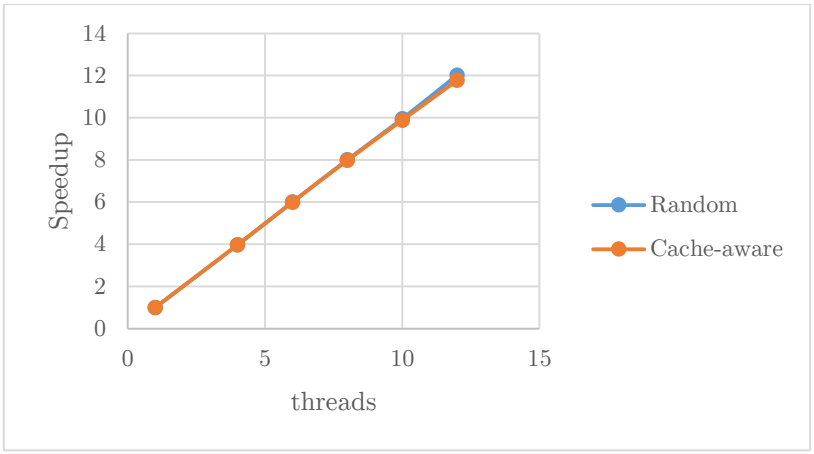

Figure 164. Matrix multiplication speedup on Termi(Cache-aware)

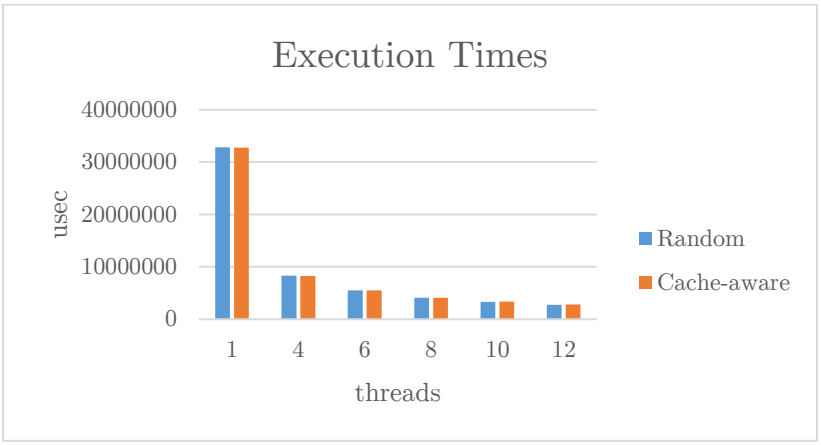

Figure 165. Matrix multiplication execution times on Termi (Cache-aware)

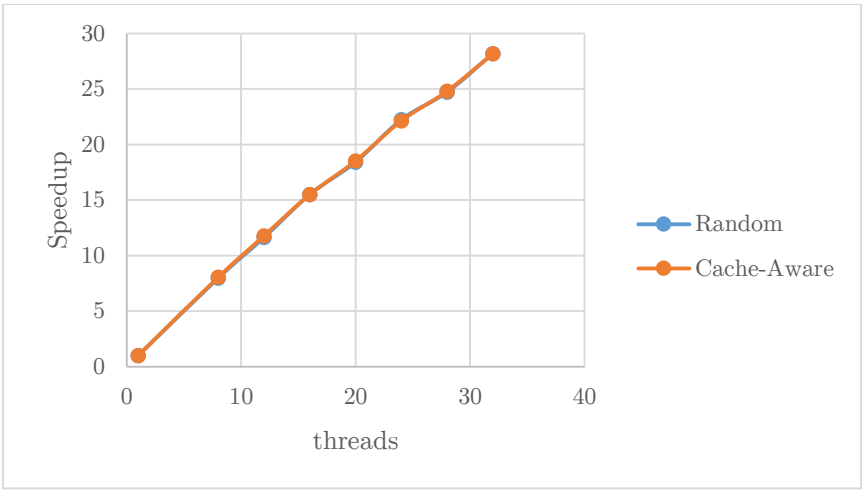

Figure 166. Matrix multiplication speedup on Sandman (Cache-aware)

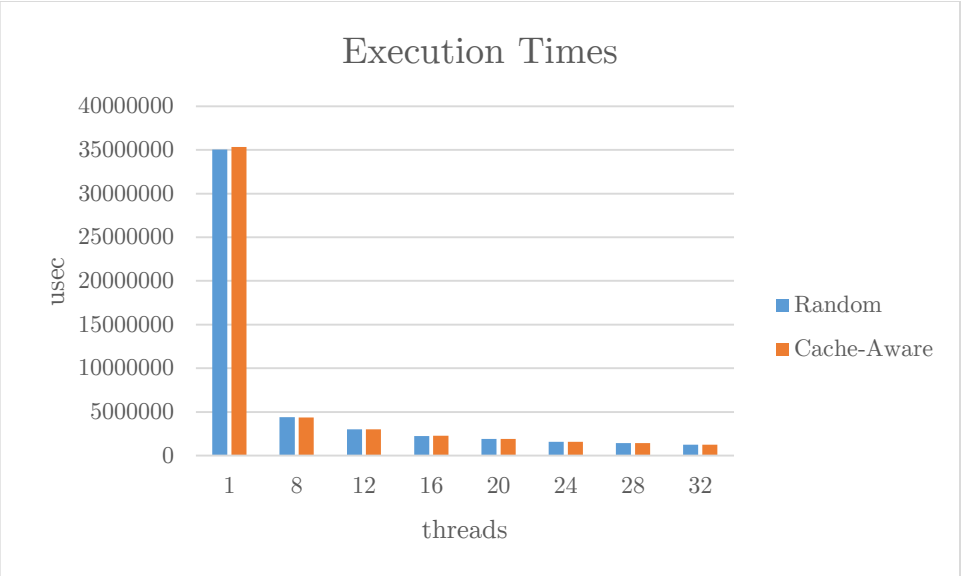

Figure 167. Matrix multiplication execution times on Sandman (Cache-aware)

8.2 Load Balancing Techniques

8.2.1 Searching for the heaviest once in five steals

The next figures show the results of running *blackscholes* on Dunnington and Sandman. We can see that they match the results of *streamcluster* in most cases, as described in Chapter 4, that is gaining performance benefits for an average number of threads, resulting in performance degradation in very large thread counts.

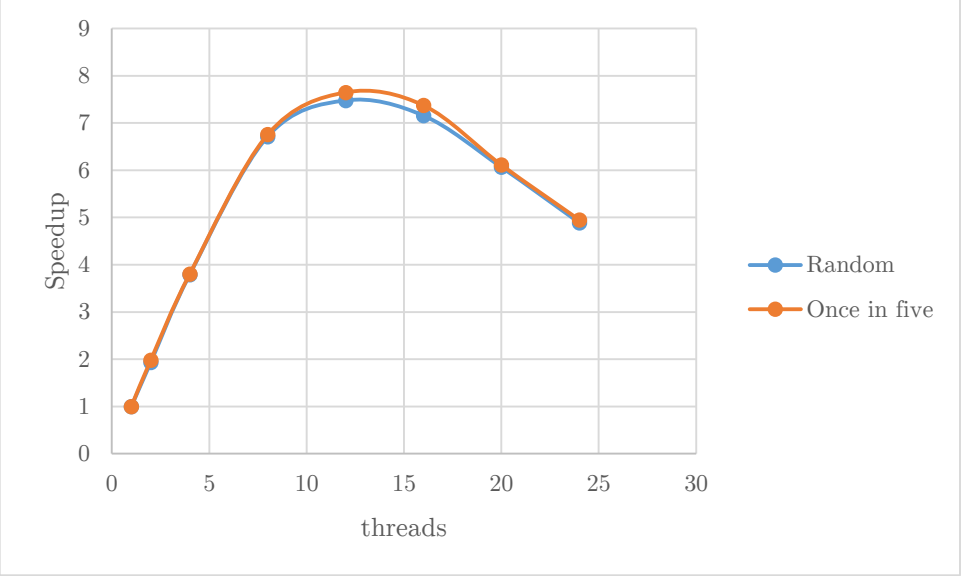

Figure 168. Blackscholes speedup on Dunnington (once in five)

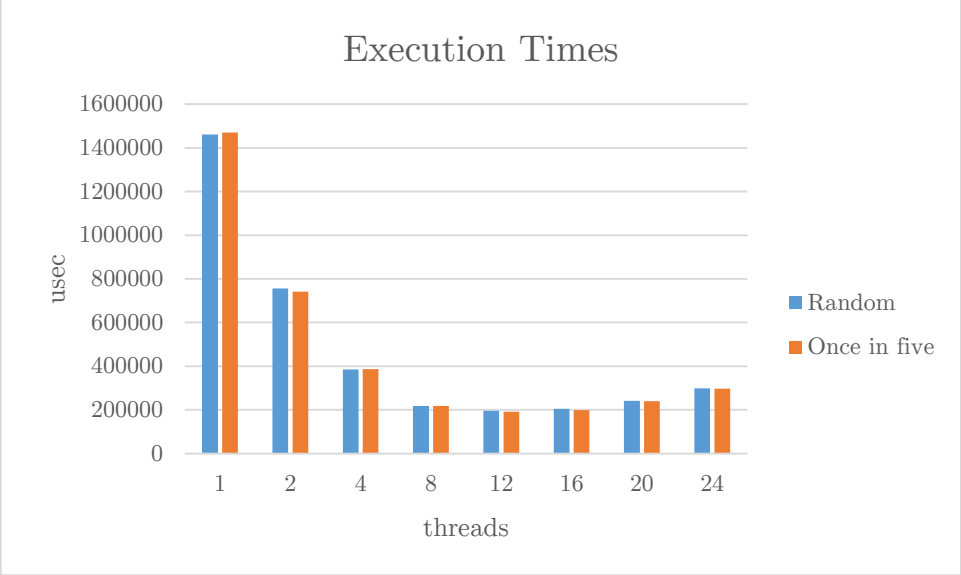

Figure 169. Blackscholes execution times on Dunnington (once in five)

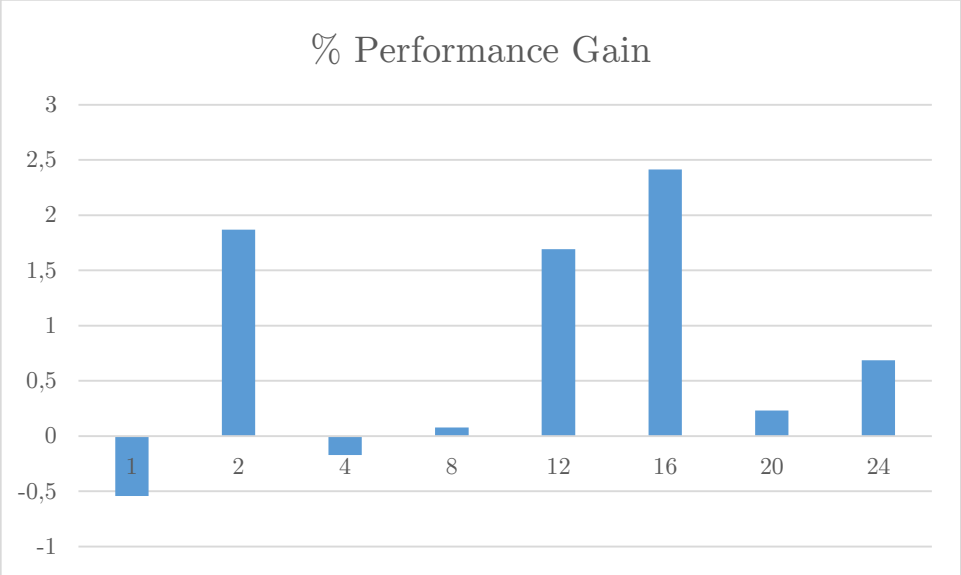

Figure 170. Blackscholes performance gains on Dunnington (once in five)

8.2.2 Global vs Local Sorted List

8.2.2.1 Streamcluster

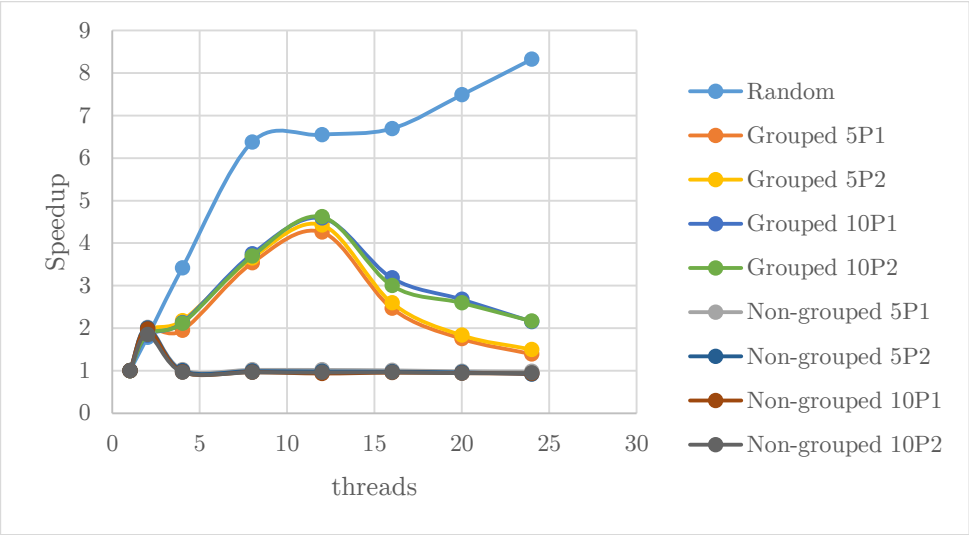

Figure 171. Streamcluster speedup on Termi (sorted list)

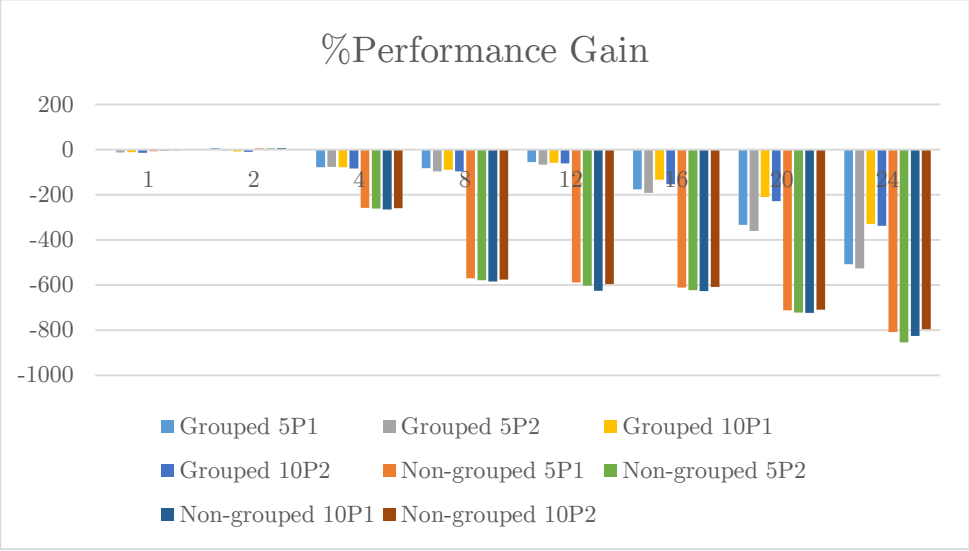

Figure 172. Streamcluster performance gains on Termi (sorted list)

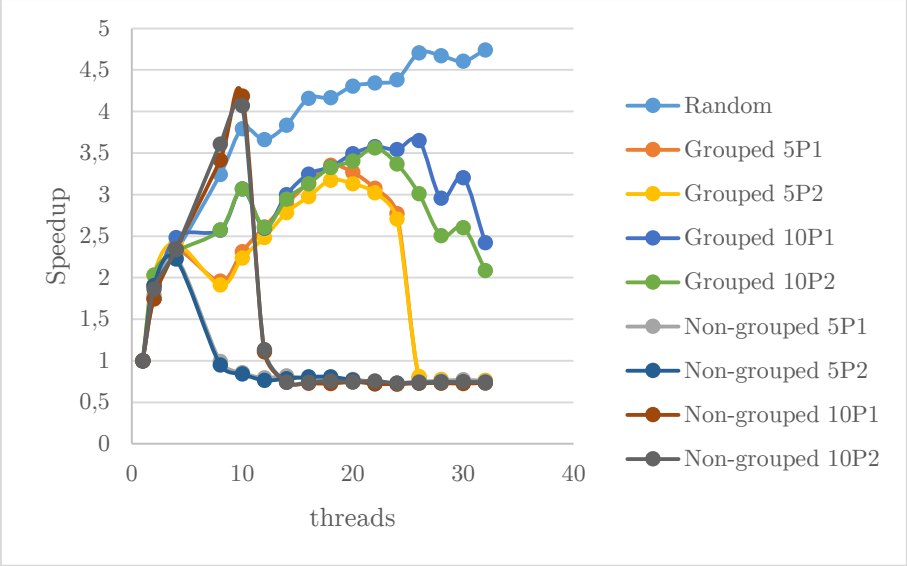

Figure 173. Streamcluster speedup on Sandman (sorted list)

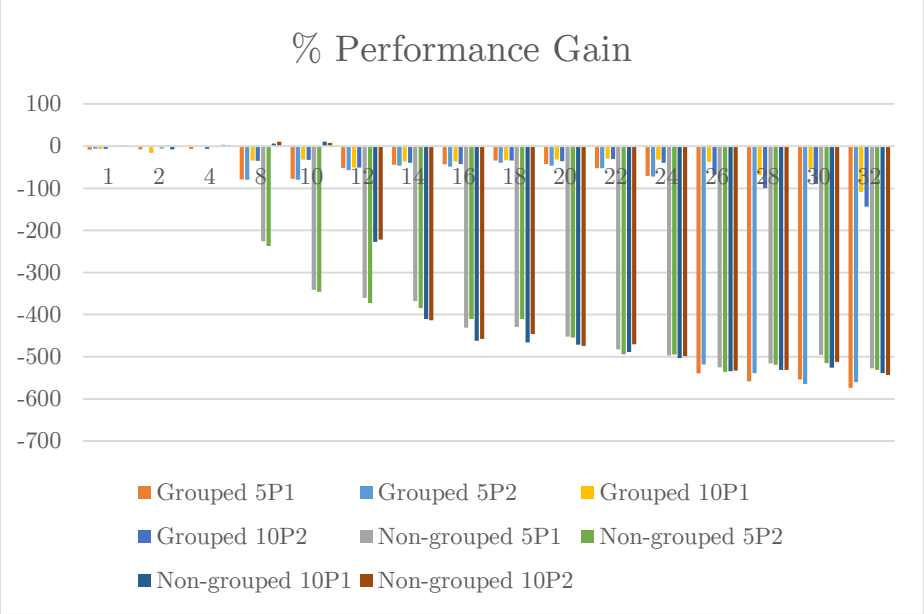

Figure 174. Streamcluster performance gains on Sandman (sorted list)

8.2.2.2 Quicksort

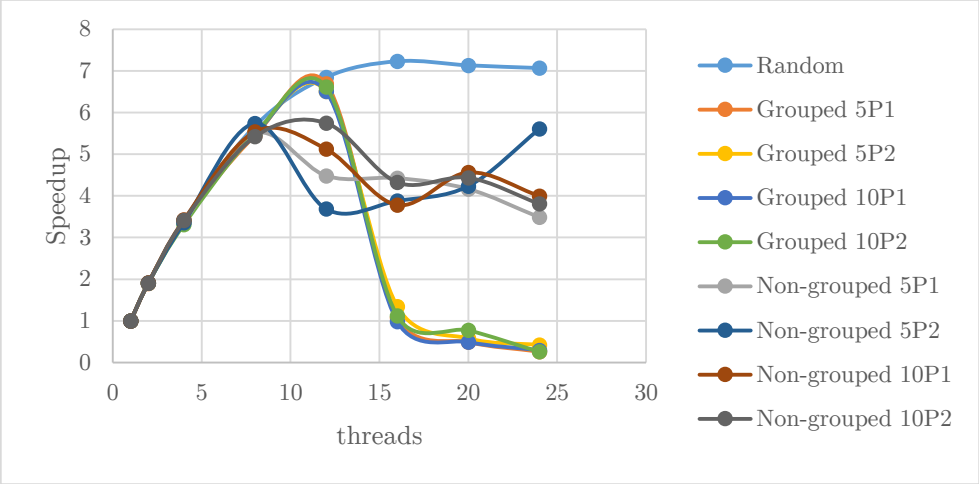

Figure 175. Quicksort speedup on Termi (sorted list)

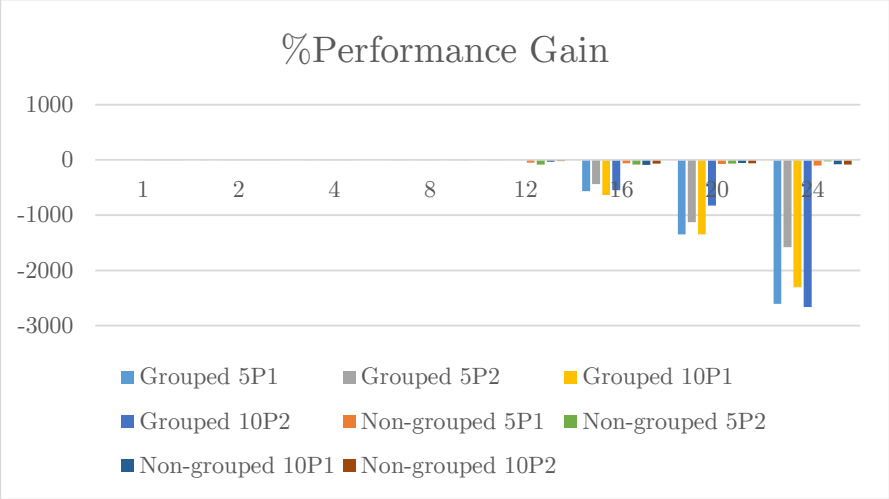

Figure 176. Quicksort performance gains on Termi (sorted list)

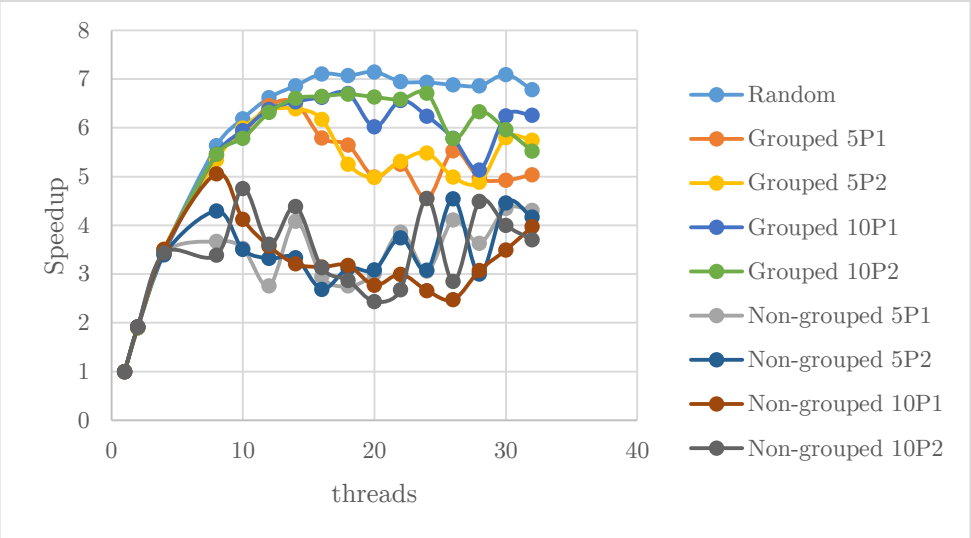

Figure 177. Quicksort speedup on Sandman (sorted list)

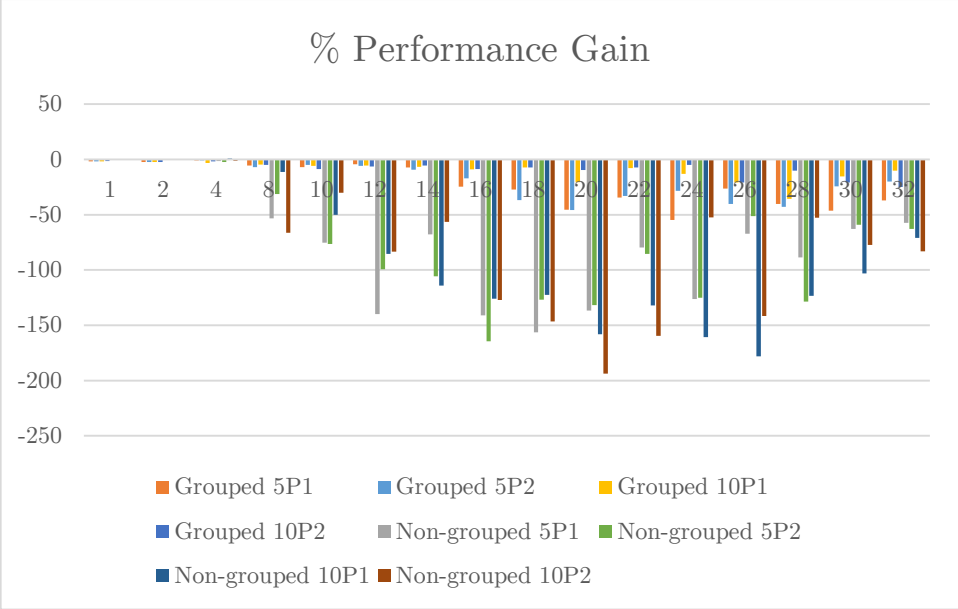

Figure 178. Quicksort performance gains on Sandman (sorted list)

8.2.2.3 Matrix multiplication

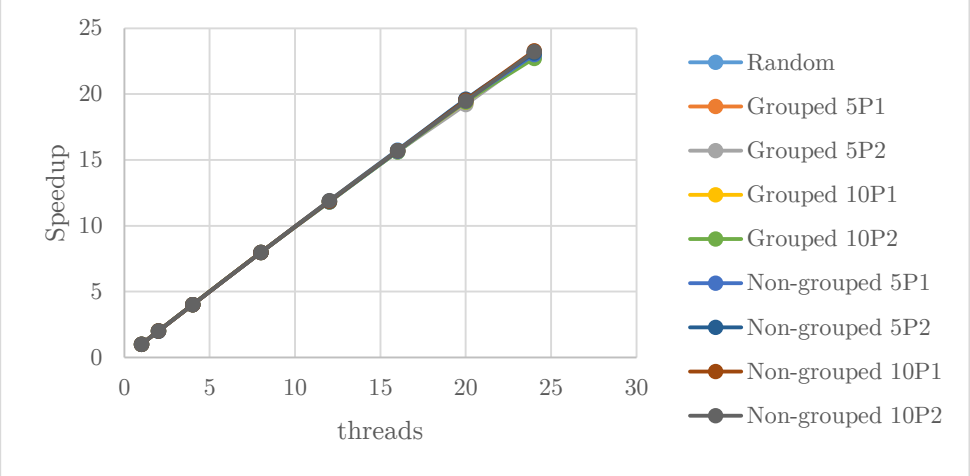

Figure 179. Matrix multiplication speedup on Dunnington (sorted list)

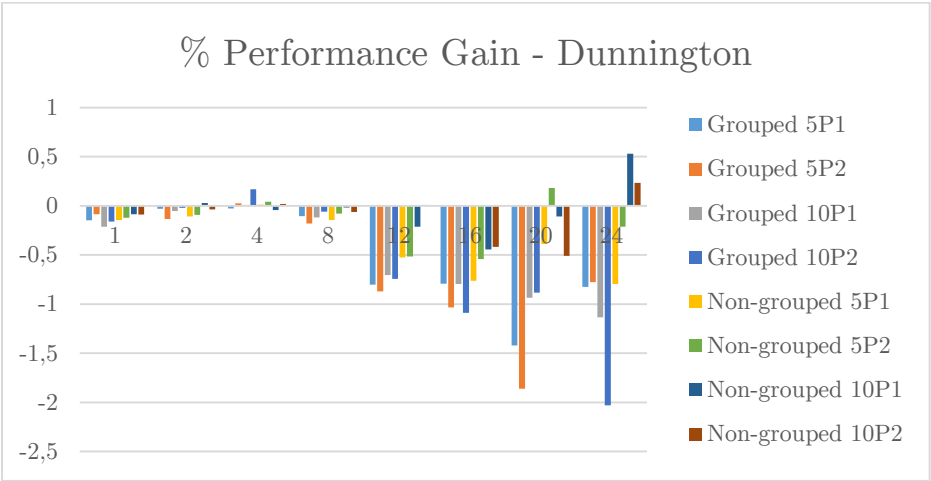

Figure 180. Matrix multiplication performance gains on Dunnington (sorted list)

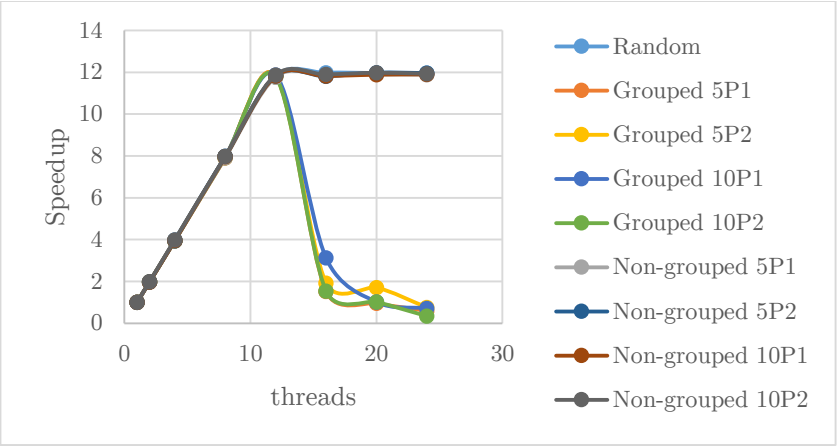

Figure 181. Matrix multiplication speedup on Termi (sorted list)

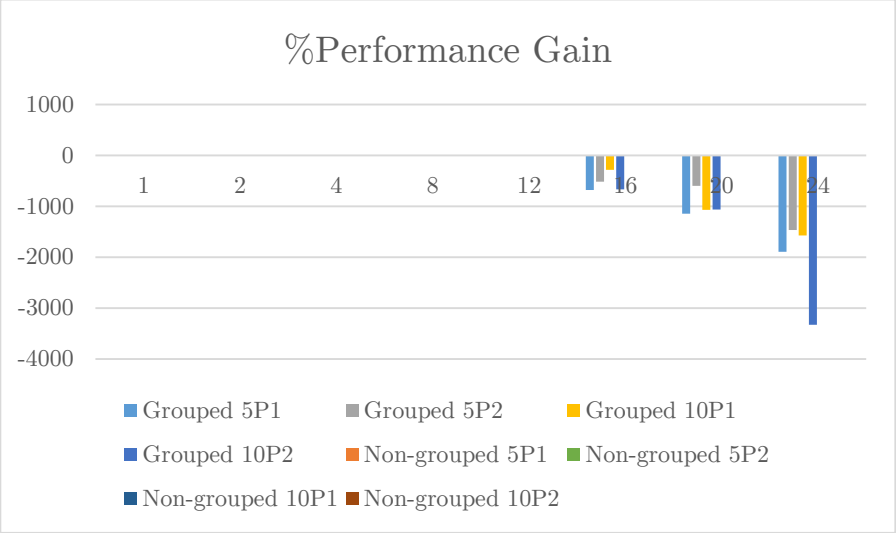

Figure 182. Matrix multiplication performance gains on Termi (sorted list)

8.2.2.4 Strassen

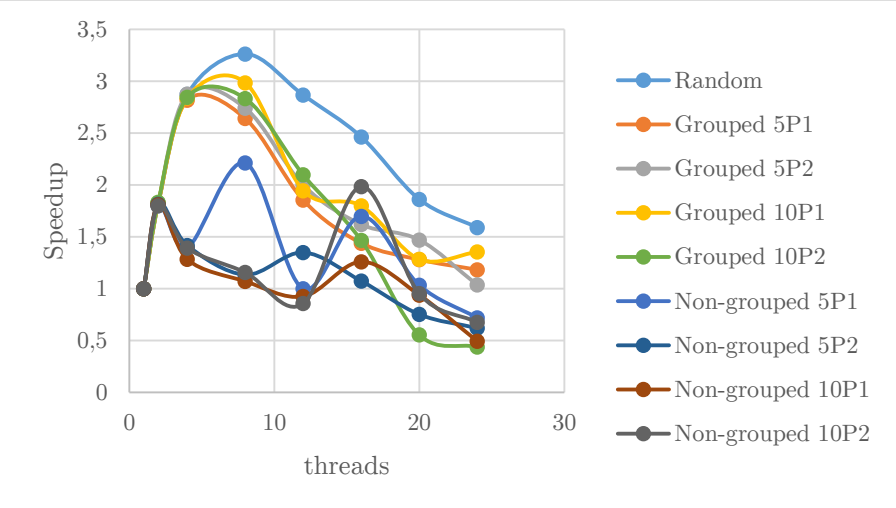

Figure 183. Strassen speedup on Dunnington (sorted list)

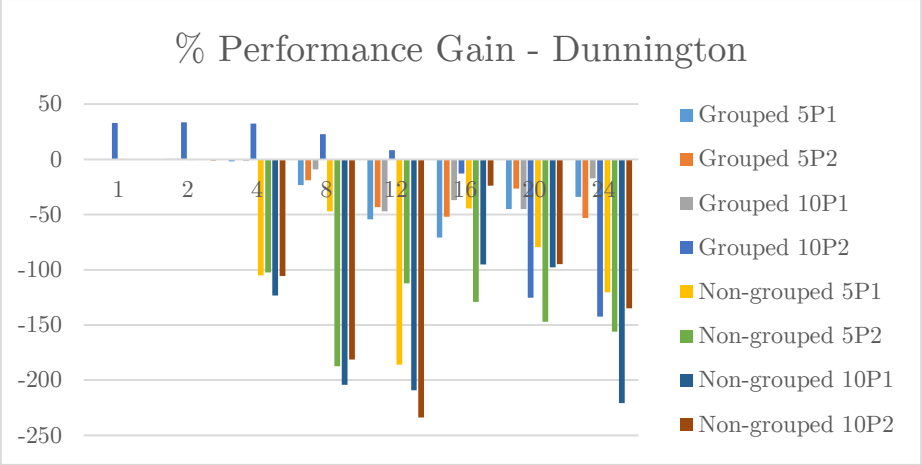

Figure 184. Strassen performance gains on Dunnington (sorted list)

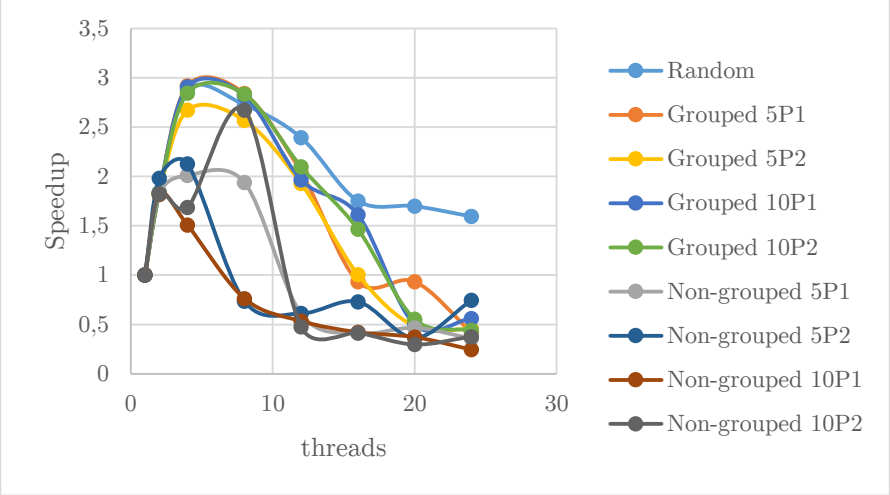

Figure 185. Strassen speedup on Termi (sorted list)

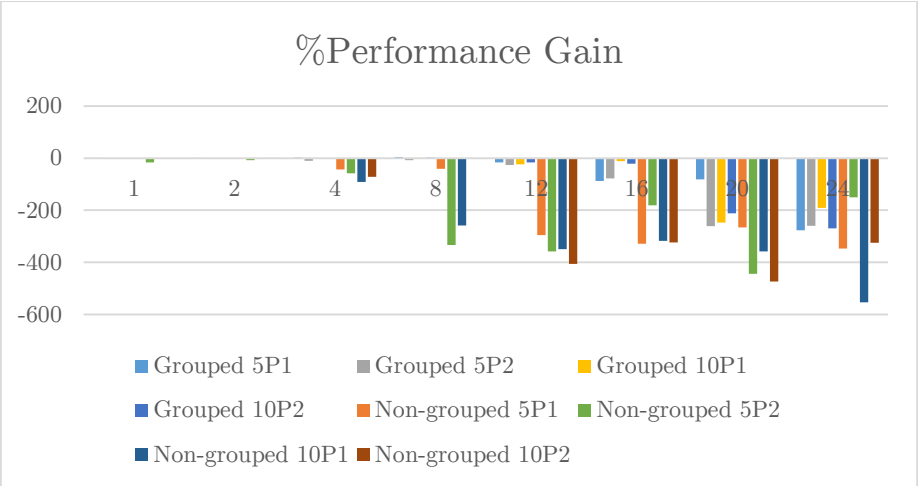

Figure 186. Strassen performance gains on Termi (sorted list)

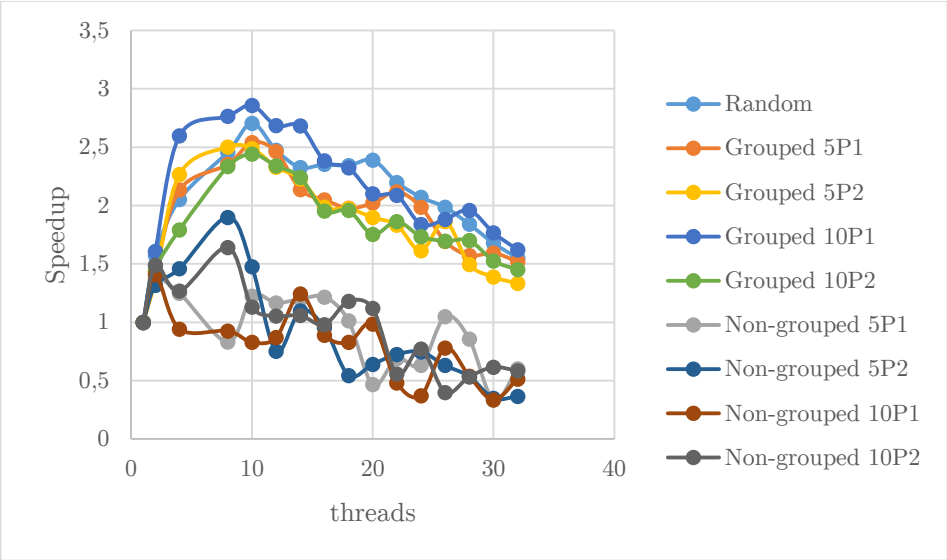

Figure 187. Strassen speedup on Sandman (sorted list)

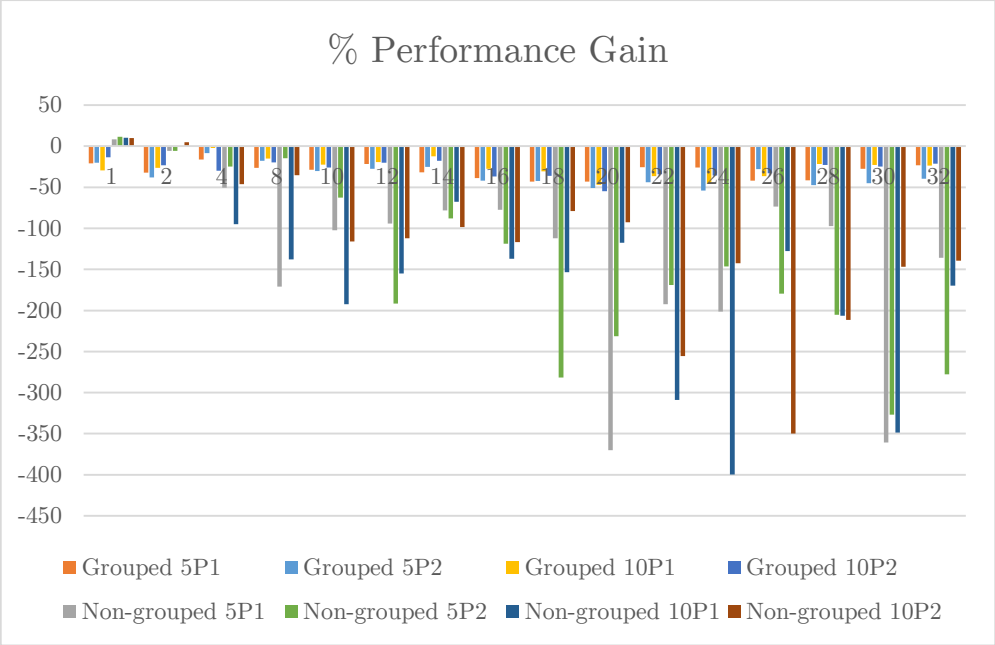

Figure 188. Strassen performance gains on Sandman (sorted list)

8.2.2.5 *Blackscholes*

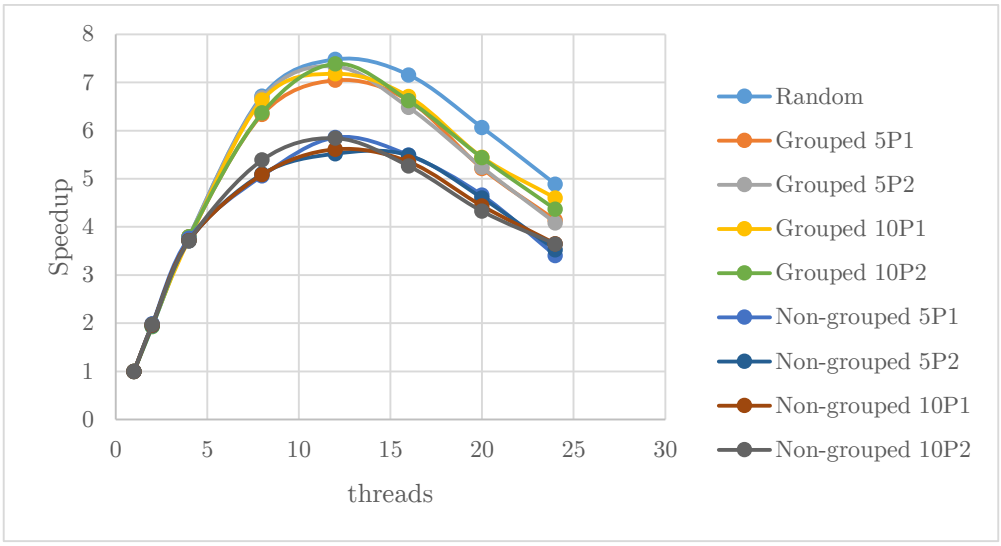

Figure 189. Blackscholes speedup on Dunnington (sorted list)

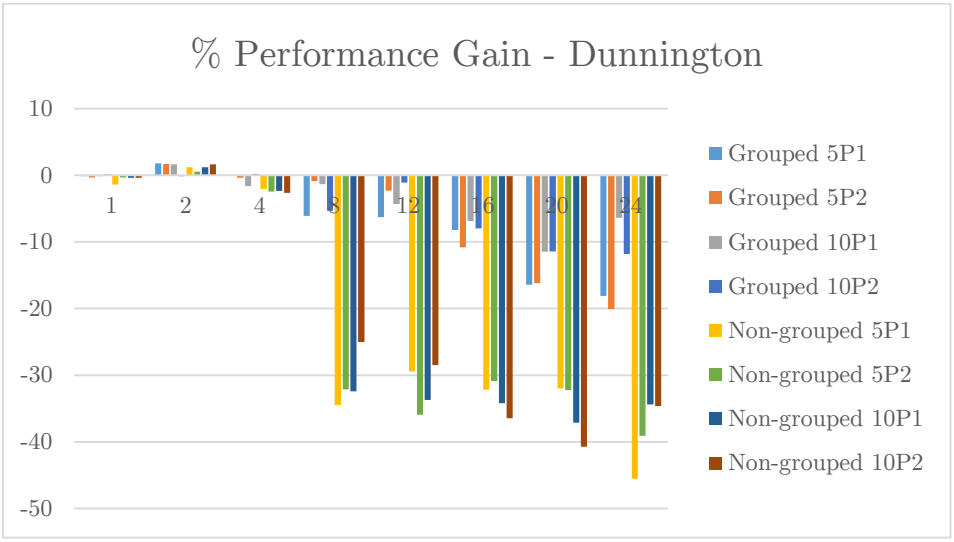

Figure 190. Blackscholes performance gains on Dunnington (sorted list)

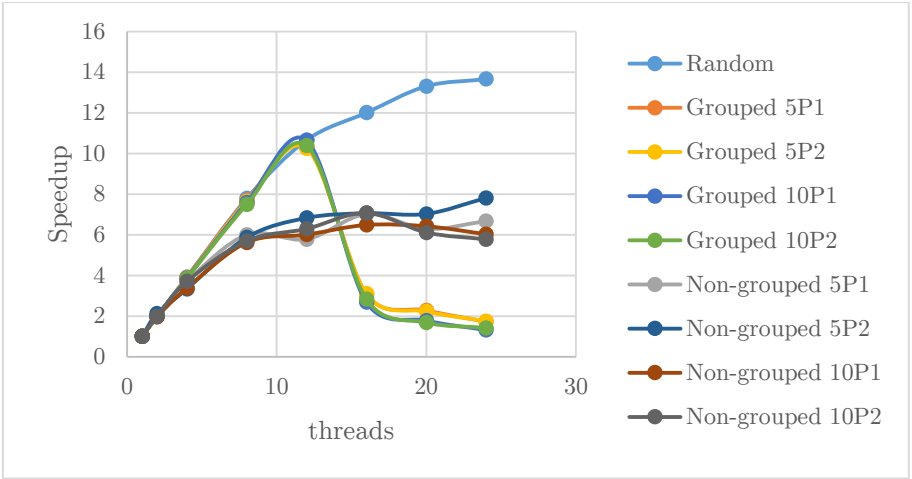

Figure 191. Blackscholes speedup on Termi (sorted list)

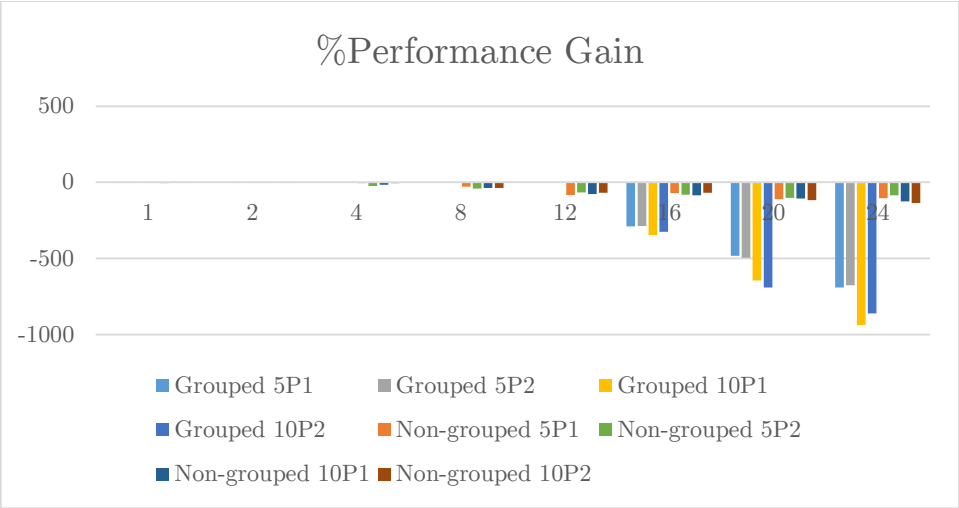

Figure 192. Blackscholes peformance gains on Termi (sorted list)

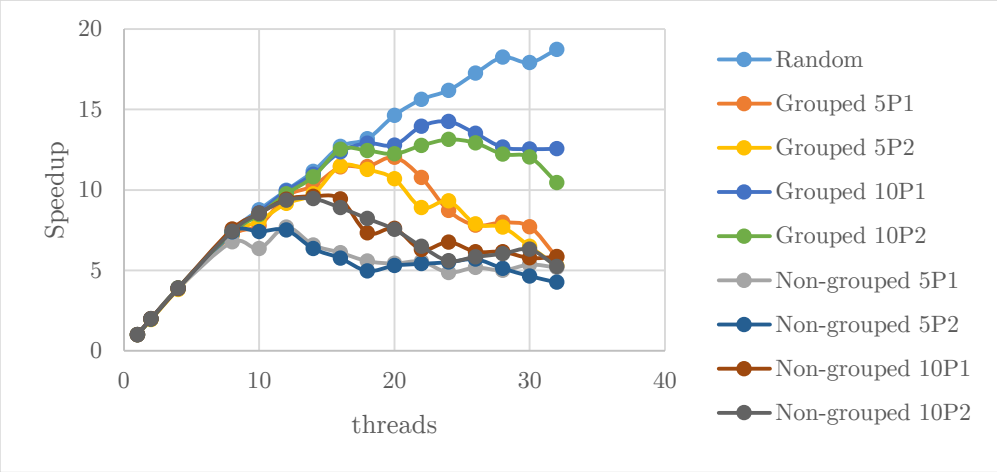

Figure 193. Blackscholes speedup on Sandman (sorted list)

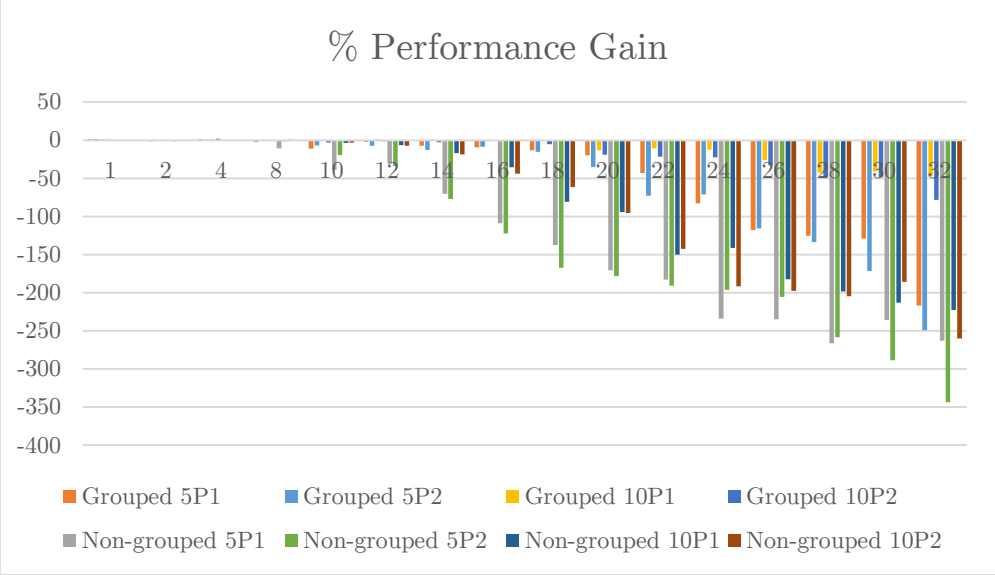

Figure 194. Blackscholes performance gains on Sandman (sorted list)

8.2.2.6 *Swaptions*

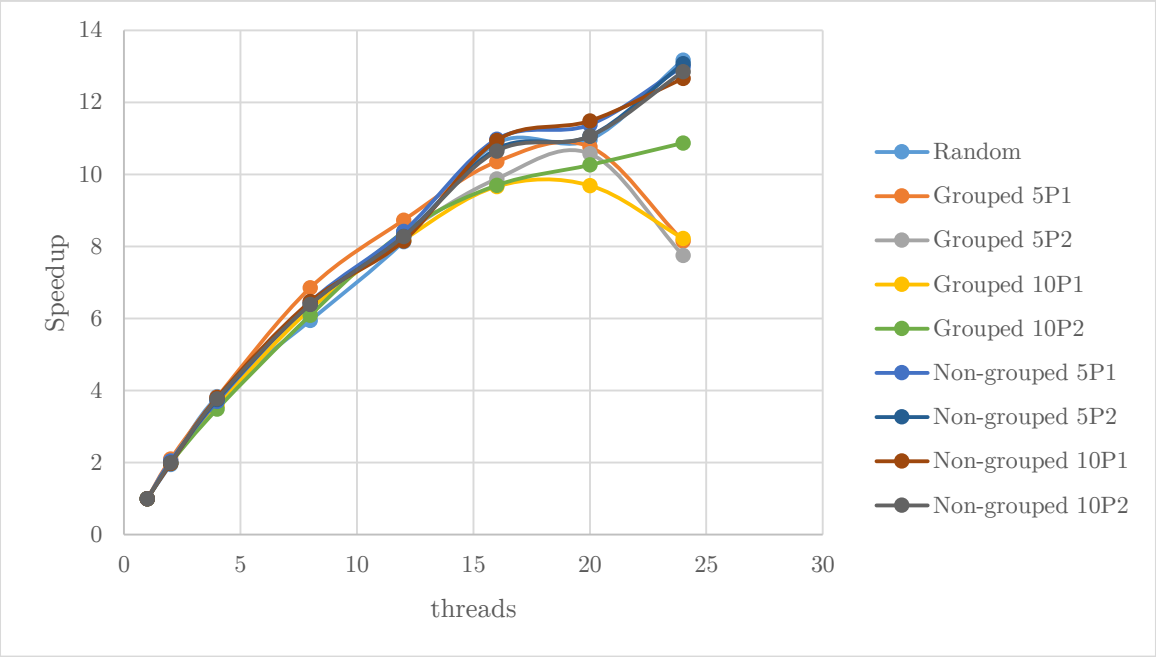

Figure 195. Swaptions speedup on Dunnington (sorted list)

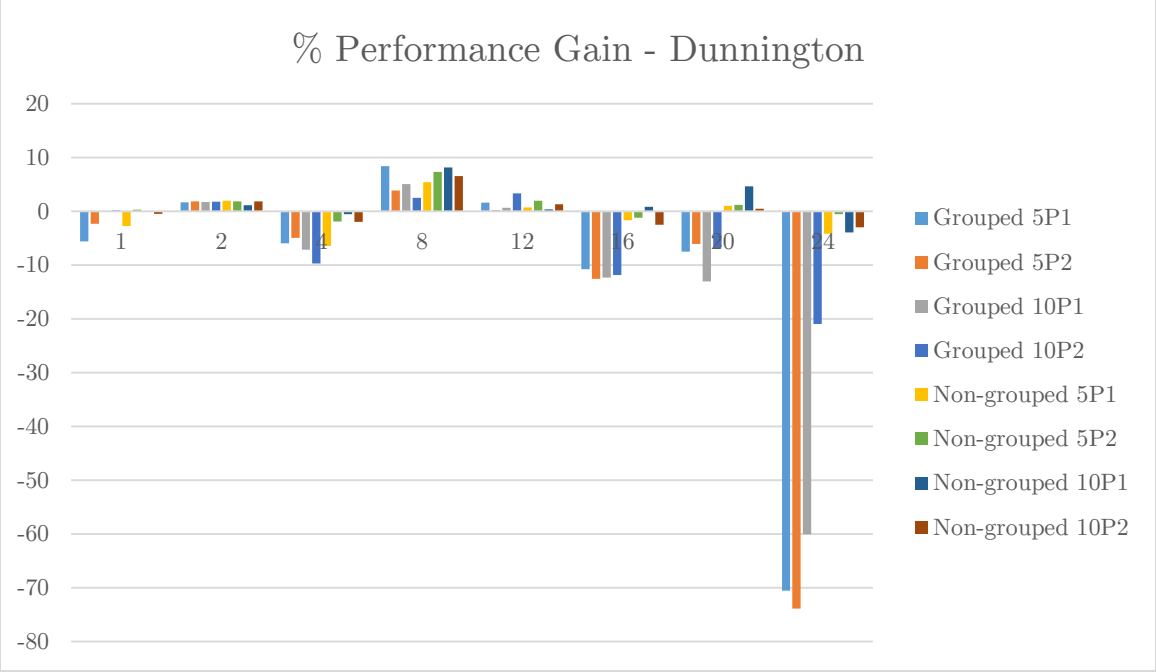

Figure 196. Swaptions performance gains on Dunnington (sorted list)

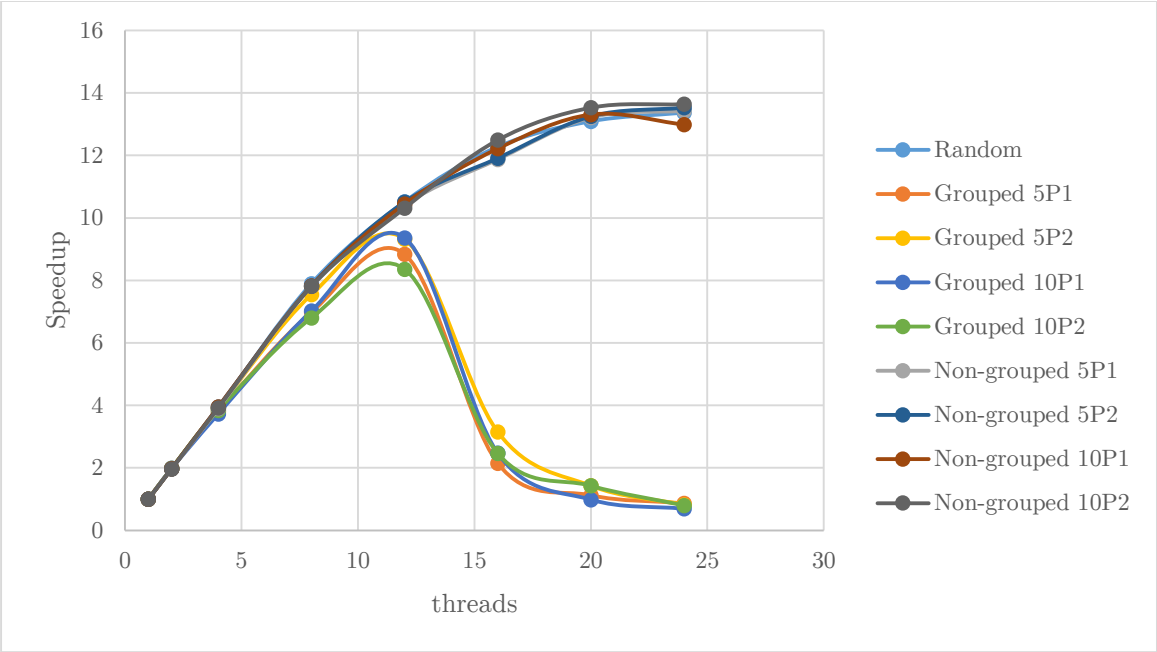

Figure 197. Swaptions speedup on Termi (sorted list)

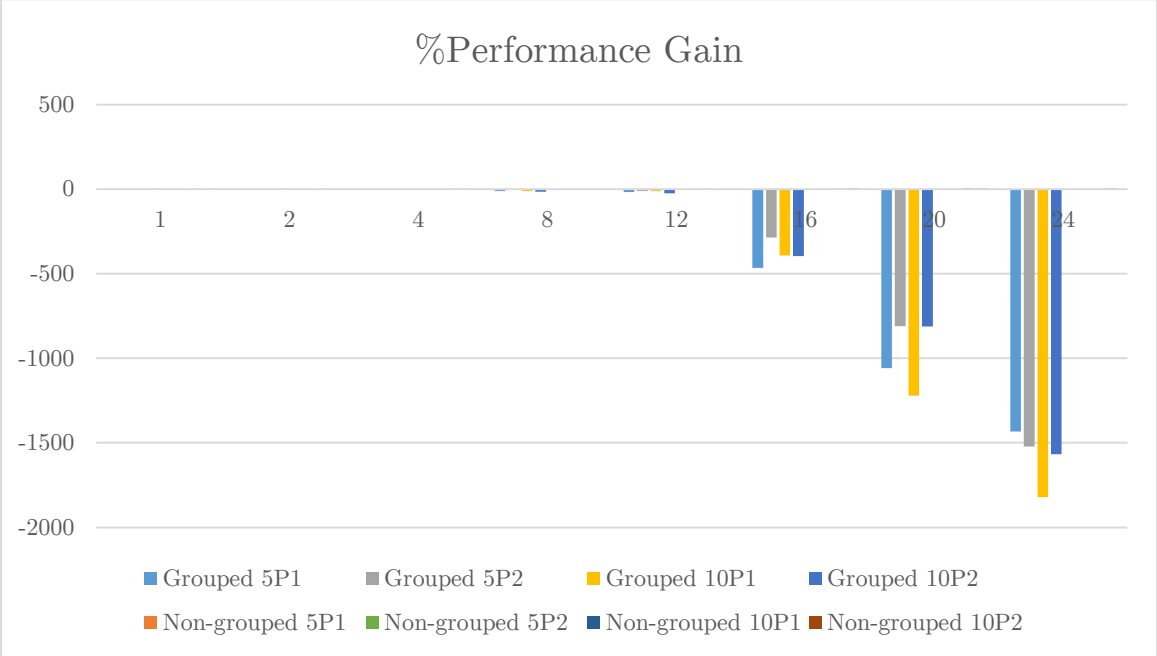

Figure 198. Swaptions performance gains on Termi (sorted list)

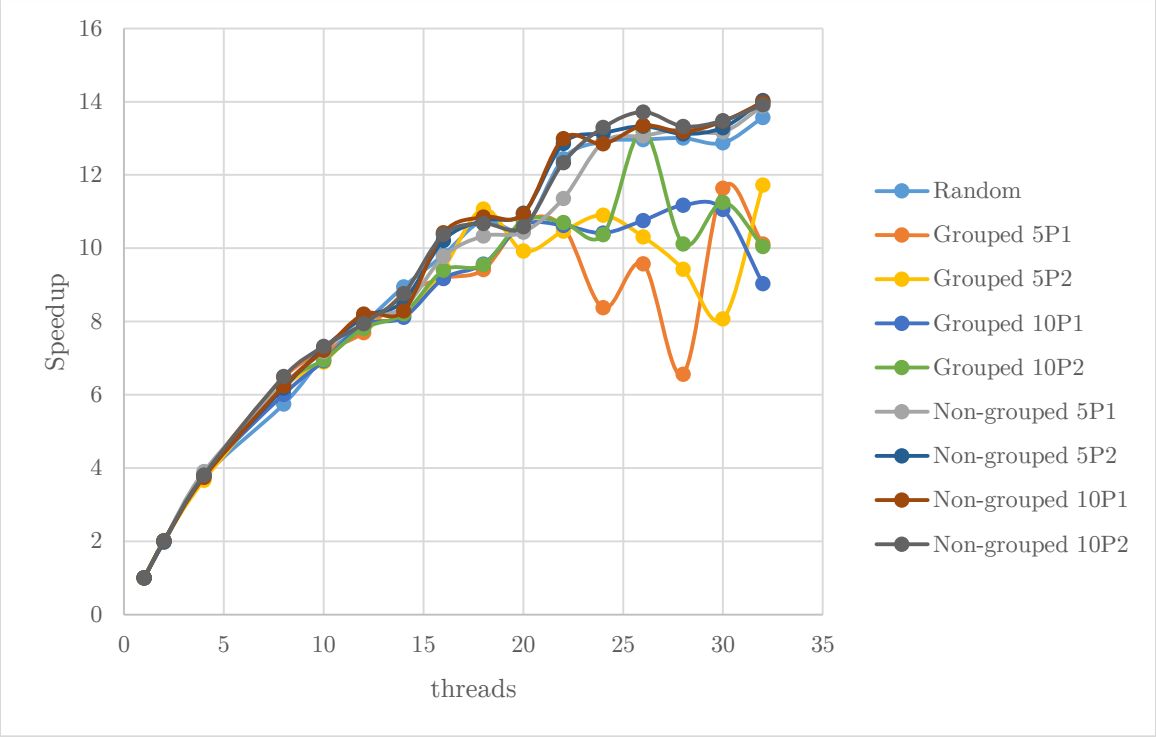

Figure 199. Swaptions speedup on Sandman (sorted list)

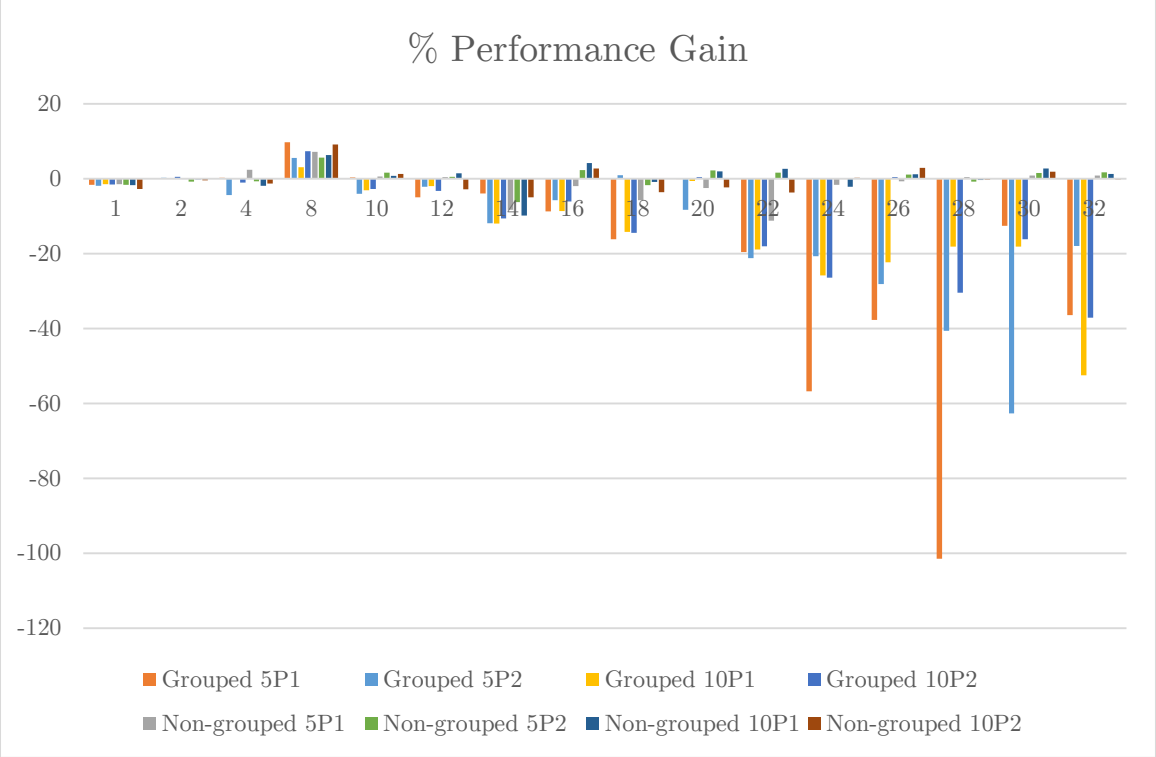

Figure 200. Swaptions performance gains on Sandman (sorted list)