NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Efficient algorithms and architectures for protein 3-D structure comparison

Anuj V. Sharma

European Union
European Social Fund
OPERATIONAL PROGRAMME
EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION & RELIGIOUS AFFAIRS
MANAGING AUTHORITY
Co-financed by Greece and the European Union
NSRF
2007-2013
programme for development
EUROPEAN SOCIAL FUND

ATHENS

DECEMBER 2018

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Αλγόριθμοι υψηλών επιδόσεων και αρχιτεκτονικές υπολογιστών για την σύγκριση πρωτεϊνών με βάση τη δομή τους

Anuj V. Sharma

European Union
European Social Fund
OPERATIONAL PROGRAMME
EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION & RELIGIOUS AFFAIRS
MANAGING AUTHORITY
Co-financed by Greece and the European Union
NSRF
2007-2013
programme for development
EUROPEAN SOCIAL FUND

ΑΘΗΝΑ

ΔΕΚΕΜΒΡΙΟΣ 2018

PhD THESIS

Efficient algorithms and architectures for protein 3-D structure comparison

Anuj V. Sharma

SUPERVISOR: Elias Manolakosr, Professor UoA

THREE-MEMBER ADVISORY COMMITTEE:

Elias Manolakosr, Professor UoA
Ioannis Emiris, Professor UoA
George Panayotou, Researcher A B.S.R.C, Fleming

SEVEN-MEMBER EXAMINATION COMMITTEE

Elias Manolakosr,
Professor UoA

Ioannis Emiris,
Professor UoA

George Panayotou,
Researcher A B.S.R.C, Fleming

Stavros Perantonis,
Researcher A DIMOKRITOS

Dimitrios Soudris,
Assoc. Professor NTUA

Yannis Cotronis,
Assoc. Professor UoA

Evangelia Chrysina,
Assoc. Professor Örebro University

Examination Date: December 19, 2018

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Αλγόριθμοι υψηλών επιδόσεων και αρχιτεκτονικές υπολογιστών για την σύγκριση πρωτεϊνών με βάση τη δομή τους

Anuj V. Sharma

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Ηλίας Μανωλάκος, Καθηγητης ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:

Ηλίας Μανωλάκος, Καθηγητης ΕΚΠΑ
Ιωάννης Εμίρης, Καθηγητης ΕΚΠΑ
Γεώργιος Παναγιώτου, Ερευνητής Α’ Φλέμινγκ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Ηλίας Μανωλάκος,
Καθηγητης ΕΚΠΑ

Ιωάννης Εμίρης,
Καθηγητης ΕΚΠΑ

Γεώργιος Παναγιώτου,
Ερευνητής Α’ Φλέμινγκ

Σταυρος Περαντώνης,
Ερευνητής Α’ ΔΗΜΌΚΡΙΤΟΣ

Δημήτριος Σούντρης,
Αναπλ. Καθηγητής ΕΜΠ

Ιωάννης Κοτρώνη,
Αναπλ. Καθηγητής ΕΚΠΑ

Ευαγγελία Χρυσίνα,
Αναπλ. Καθηγητής Örebro University

Ημερομηνία Εξέτασης: 19 Δεκέμβριος 2018

ABSTRACT

Protein Structure Comparison (PSC) is a well developed field of computational proteomics with active interest since it is widely used in structural biology and drug discovery. Fast increasing computational demand for all-to-all protein structures comparison is a result of mainly three factors: rapidly expanding structural proteomics databases, high computational complexity of pairwise PSC algorithms, and the trend towards using multiple criteria for comparison and combining their results (MCPSC). Despite the sustained interest in the field over the past three decades there are still open challenges in large-scale MCPSC. Firstly, its application using modern many-core and multi-core processor architectures remains unexplored. Secondly, there are few works that apply MCPSC to develop biologically relevant clusters and classify proteins. Finally, there is lack of bioinformatics software tools to support comparative analysis of large protein datasets on commodity computers.

In order to address these challenges, in this thesis we have developed a software framework that exploits many-core and multi-core CPUs to implement efficient parallel MCPSC schemes in modern processors based on three popular PSC methods, namely, TMalign, CE, and USM. We evaluate and compare the performance and efficiency of two parallel MCPSC implementations using Intel's experimental many-core Single-Chip Cloud Computer (SCC) CPU as well as Intel's Core i7 multi-core processor. Further, we have developed a Python based utility, called *pyMCPSC*, allowing users to perform MCPSC efficiently, by exploiting the parallelism afforded by the multi-core CPUs of today's desktop computers. We show how *pyMCPSC*, which combines five PSC methods and five different consensus scoring schemes, facilitates the analysis of similarities in protein domain datasets and how it can be easily extended to incorporate more PSC methods in the consensus scoring as they are becoming available.

Experimental results from our analysis, show that the 48-core Intel SCC NoC processor is more efficient than the latest generation Core i7 CPU, achieving a speedup factor of 42 (efficiency of 0.9), making many-core processors an exciting technology for large-scale structural proteomics. We compare and contrast the performance of the two processors on several benchmark datasets and also show that MCPSC outperforms its component PSC methods in grouping related domains, achieving a high F-measure of 0.91 on the CK34 dataset. We further demonstrate using the Proteus300 dataset, that consensus MCPSC scores form a reliable basis for identifying the true classification of a protein domain, as evidenced both by ROC analysis as well as Nearest-Neighbor analysis. Structure similarity based "Phylogenetic Trees" generated by consensus scores provide insight into functional grouping within the dataset of domains. Furthermore, scatter plots generated by *pyMCPSC* reveal the existence of strong correlation between protein domains belonging to SCOP Class C and loose correlation between those of SCOP Class D within the Proteus dataset. Such analyses and corresponding visualizations help users quickly gain insights about their datasets. Finally we demonstrate that even very large datasets (such as SCOPCATH) can be processed and consensus scores based structural analysis carried out on readily available multi-core processors using our developed methods. *pyMCPSC* has been released to the proteomics community through GitHub and can be accessed at https://github.com/xulesc/pymcpsc.

SUBJECT AREA: Bioinformatics, Protein Structure Comparison

KEYWORDS: Protein, Homology, Machine learning, Sequence comparison, Structure comparison

ΠΕΡΙΛΗΨΗ

Η σύγκριση πρωτεϊνών με βάση τη δομή τους (protein structure comparison, PSC) αποτελεί τομέα της υπολογιστικής πρωτεομικής με ενεργό ενδιαφέρον καθότι χρησιμοποιείται ευρέως στη δομική βιολογία και την ανακάλυψη νέων φαρμάκων. Η ταχεία αύξηση των υπολογιστικών απαιτήσεων για τη σύγκριση πρωτεϊνικών δομών είναι αποτέλεσμα τριών κυρίως παραγόντων: ταχεία επέκταση των βάσεων δεδομένων με νέες δομές πρωτεϊνών, υψηλή υπολογιστική πολυπλοκότητα των αλγορίθμων σύγκρισης δύο πρωτεινών, τάση στον τομέα για χρήση πολλαπλών μεθόδων σύγκρισης και συνδυασμό των αποτελεσμάτων τους (multicriteria PSC, MCPSC) σε ένα σκορ συναίνεσης (consensus methods). Παρά την μεγάλη πρόοδο, εξακολουθούν να υπάρχουν ανοικτές προκλήσεις στην εφαρμογή MCPSC τεχνικών σε ευρεία κλίμακα. Πρώτον, η επιτάχυνση της λειτουργίας MCPSC με τη χρήση σύγχρονων αρχιτεκτονικών επεξεργαστών πολλών πυρήνων παραμένει κατά πολύ ανεξερεύνητη. Δεύτερον, η εφαρμογή μέθόδων MCPSC στη ταξινόμηση νεων δομών πρωτεϊνών είναι περιορισμένη λόγω του υπολογιστικού κόστους και της ανάγκης χρήσης υπερυπολογιστικών δομών. Τέλος, υπάρχει έλλειψη ελεύθερα διαθέσιμων εργαλείων βιοπληροφορικής που να υποστηρίζουν τη συστηματική σύγκριτική ανάλυση και κατηγοριοποίηση μεγάλων συνόλων πρωτεϊνών με βάση τη δομή τους σε κοινούς υπολογιστές.

Προκειμένου να αντιμετωπιστούν αυτές οι σημαντικές προκλήσεις, σε αυτή την διατριβή αναπτύξαμε πλαίσιο λογισμικού που εκμεταλλεύεται σύγχρονους επεξεργαστές (CPUs) για την αποδοτική υλοποίηση παράλληλων MCPSC τεχνικών βασισμένων σε τρεις δημοφιλείς μεθόδους PSC, τις TMalign, CE και USM. Συγκρίνουμε και αξιολογούμε την απόδοση και την αποδοτικότητα δύο παράλληλων υλοποιήσεων, μια για τον επεξεργαστή αρχιτεκτονικής many-core Intel Single Cloud Computer (SCC) με 48 πυρήνες οργανωμένους σε δίκτυο πλέγματος (Network on Chip), και μια και για τον γνωστό επεξεργαστή Intel Core i7 πολλαπλών πυρήνων (multi-core CPU). Επιπλέον, αναπτύξαμε Python εφαρμογή, που ονομάζεται pyMCPSC, και επιτρέπει στους χρήστες να εκτελούν εύκολα υπολογιστικά πειράματα βασισμένα σε MCPSC με μεγάλα σύνολα δεδομένων, αξιοποιώντας τον παραλληλισμό που προσφέρουν οι επεξεργαστές πολλαπλών πυρήνων των σημερινών επιτραπέζιων υπολογιστών. Δείχνουμε πώς το pyMCPSC, το οποίο συνδυάζει πέντε δημοφιλείς μεθόδους PSC για τη δημιουργία πέντε διαφορετικών σκορ συναίνεσης (consensus scores), επιταχύνει σημαντικά και διευκολύνει την συγκριτική ανάλυση μεγάλων συνόλων δεδομένων με δομές πρωτεϊνών. Επιπλέον μπορεί να επεκταθεί εύκολα ώστε να ενσωματώνει στους αλγόριθμους συναίνεση και νέες μεθόδους PSC που μπορεί να προταθούν μελλοντικά καθώς ο τομέας εξελίσσεται.

Τα αποτελέσματ συγκριτικής ανάλυσής δείχνουν ότι ο επεξεργαστής Intel SCC με 48 πυρήνες (Network on Chip) είναι πιο αποδοτικός από την τελευταίας γενιάς Core i7 CPU, επιτυγχάνοντας συντελεστή επιτάχυνσης 42 (απόδοση 0,9), και καθιστώντας τους επεξεργαστές αρχιτεκτονικής many-core τεχνολογία επιλογής για την υπολογιστική δομική πρωτεομική μεγάλης κλίμακας. Επιπλέον, δείχνουμε ότι το MCPSC ξεπερνά τις μεθόδους PSC στις οποίες στηρίζεται ως προς την επιτυχία της ομαδοποίησης νεων πρωτεϊνών, επιτυγχάνοντας F-measure 0,91 στο σύνολο δεδομένων αναφοράς CK34. Επιπλέον, δείχνουμε, με τη χρήση του συνόλου δεδομένων Proteus300, ότι οι τεχνικές MCPSC που αναπτύχθηκαν βελτιωνουν την κατηγοριοποίηση πρωτεϊνών, όπως αυτό αποδεικνύεται τόσο από την ανάλυση ROC όσο και από την ανάλυση κοντινότερων γειτόνων (Nearest-Neighbor). Επιπλεον. τα ”φυλογενετικά δέντρα” που προκύπτουν με τη χρηση MCPSC παρέχουν χρήσιμες πληροφορίες και σχετικά με τη πιθανή λειτουργικότητα νεων πρωτεϊνών. Τέλος, η συγκριτική ανάλυση αναδεικνύει την ύπαρξη ισχυρής συσχέτισης πρωτεϊνικών δομών της κατηγορίας SCOP class C και χαλαρής συσχέτισης μεταξύ εκείνων της κατηγορίας SCOP class D (Proteus300). Τέτοιου είδους ενδελεχείς αναλύσεις δεδομένων και οι αντίστοιχες οπτικοποιήσεις που τις συνοδεύουν βοηθούν τους χρήστες να εξερευνούν και να εξάγουν γνώση από σύνολα δεδομένων που αναλύουν, όσο μεγάλα κι αν είναι αυτά. Δειχνουμε ότι ακόμη και σε πολύ μεγάλα σύνολα δεδομένων, με χιλίαδες domains (όπως το SCOPCATH), μπορεί να εφαρμοστεί αποδοτικά MCPSC επεξεργασία προκειμένου να διερευνηθεί η εσωτερική δομή τους, αξιοποιώντας τους επεξεργαστές πολλών πυρήνων που υπάρχουν σήμερα στους ατομικούς υπολογιστες. Το pyMCPSC που υλοποιεί παράλληλα όλη την υπολογιστική ροή (pipeline) που αξιοποιεί μεθόδους MCPSC οι οποίες αναπτύχθηκαν σε αυτή την διδακτορική διατριβή διατίθεται ελεύθερα στη επιστημονική κοινότητα στο σύνδεσμο https://github.com/xulesc/pymcpsc.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Βιοπληροφορική, Σύγκριση δομών πρωτεϊνών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Πρωτεΐνη, Ομολογία, Μηχανική μάθηση, Σύγκριση αλληλουχιών, Σύγκριση δομών

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Σημείο εκκίνησης τηε διδακτορικής διατριβής ήταν η ενδελεχής ανασκόπηση της βιβλιογραφίας σχετικά με αλγόριθμους σύγκρισης πρωτεϊνών με βάση τη τρισδιάστατη (3D) δομή τους. Πραγματοποιήθηκε ταξινόμηση των δημοσιευμένων μεθόδων και αναζήτηση κοινών μοτίβων στον τομέα της σύγκρισης δομών πρωτεϊνών (PSC). Η μελέτη επεκτάθηκε και στην ανάλυση υπολογιστικών τεχνικών υψηλής απόδοσης που χρησιμοποιούνται γενικότερα στο χώρο της δομικής πρωτεομικής (structural proteomics) και βοήθησε να γίνουν κατανοητές σε βάθος οι προκλήσεις που παρουσιάζουν τα προβλήματα αυτά. Τα αποτελέσματα έδωσαν ισχυρά κίνητρα για περαιτέρω έρευνα και συνοψίζονται παρακάτω:

- Οι μέθοδοι σύγκρισης δομών πρωτεϊνών που αναφέρονται στη βιβλιογραφία μπορούν να ταξινομηθούν σε τεχνικές βασισμένες σε γράφους, Δυναμικό Προγραμματισμό, Ανάκτηση Πληροφορίας (information retrieval), Γεωμετρικές τεχνικές βασισμένες σε Contact maps, τεχνικές Πολυωνυμικού Χρόνου, και τεχνικές Simulated annealing.

- Οι παραπάνω τεχνικές χρησιμοποιούν διαφορετικές μετρικές ομοιότητας για τη σύγκριση δομών πρωτεϊνών. Οι περισσότερες από αυτές ανήκουν σε δύο κατηγορίες: αυτές που συγκρίνουν την απόσταση μεταξύ αντίστοιχων ζευγών ατόμων στις δύο δομές, και εκείνες που συγκρίνουν τις σχετικές θέσεις των ατόμων στις υπερτιθέμενες (superimposed) δομές πρωτεϊνών.

- Οι μέθοδοι σύγκρισης δομών πρωτεϊνών μπορούν να χωριστούν σε δύο γενικές κατηγορίες: α) Σειριακές (sequential), όπου δομικά παρόμοια τμήματα είναι διατεταγμένα στην αλληλουχία της πρωτεΐνης, και β) μη-σειριακές (non-sequential), όπου δομικά παρόμοια τμήματα δεν είναι διατεταγμένα στην αλληλουχία της πρωτεΐνης. Οι πιο σύγχρονες μέθοδοι επικεντρώνονται στη γραμμική ευθυγράμμιση (sequential alignment) δομών πρωτεϊνών. Αυτό οφείλεται στις υψηλότερες επιδόσεις που παρουσιάζουν οι αντίστοιχοι αλγόριθμοι γραμμικής ευθυγράμμισης τόσο σε ταχύτητα όσο και σε ακρίβεια. Παρόλα αυτά, εξακολουθεί να υπάρχει ενδιαφέρον και για αλγορίθμους μη-γραμμικής ευθυγράμμισης σε συγκεκριμένες εφαρμογές.

- Η σύγκριση και ευθυγράμμιση δομών πρωτεϊνών είναι πρόβλημα NP-hard, άρα οι διαθέσιμες μέθοδοι είναι ευριστικές, με διαφορετικές μεθόδους να δίνουν διαφορετικά μεν αλλά συχνά βιολογικά ενδιαφέροντα αποτελέσματα. Επιπλέον, χρησιμοποιείται ποικιλία μέτρων ομοιότητας. Αποτέλεσμα των παραπάνω είναι να ενισχύεται προσφάτως η τάση για σύγκριση πρωτεϊνών με περισσότερα του ενός κριτήρια (Multi-criteria protein structure comparison - MCPSC). Στόχος αυτής της νέας προσέγγισης είναι η παραγωγή ολοκληρωμένης εικόνας ως προς την ομοιότητα δύο δομών πρωτεϊνών με την χρήση πολλαπλών μεθόδων σύγκρισης και το συνδυασμό τελικά των αποτελεσμάτων ώστε να εξαχθεί ένα τελικό σκορ ομοφωνίας (consensus score).

- Οι προσπάθειες επιτάχυνσης της σύγκρισης δομών πρωτεϊνών επικεντρώνονται μέχρι σήμερα στη χρήση κατανεμημένων συστημάτων, π.χ. Υπολογιστικών Πλεγμάτων (Grids) και Συστάδων (Clusters) υπολογιστών. Η σχετική έρευνα επικεντρώνεται στην επίτευξη της μεγαλύτερης δυνατής επιτάχυνσης με χρήση διαμοιραζόμενων επεξεργαστικών δομών μεγάλης κλίμακας. Δεν έχει γίνει ιδιαίτερη έρευνα στην χρήση επιταχυντών (accelerators) υλικού, όπως τα FPGAs, ή μονάδων επεξεργασίας γραφικών (GPUs), ή αναδυόμενων πολυπύρηνων επεξεργαστών (multi- and many-core CPUs) για σταθμούς εργασίας προσωπικών υπολογιστών (desktop workstation PCs) που είναι πλέον ευρέως διαθέσιμοι στους ερευνητές, με λογικό κόστος.
- Η γενική άποψη που επικρατεί στον τομέα είναι ότι καμία μετρική και καμία μέθοδος σύγκρισης δεν είναι ολοκληρωμένη από μόνη της. Υπάρχει ανάγκη για ανάπτυξη στρατηγικών σύγκρισης δομών πρωτεϊνών με συνδυαστική χρήση πολλαπλών μεθόδων και αξιοποίηση των πολυπύρηνων CPUs για υψηλές επιδόσεις χώρις χρήση υπολογιστικών υπερδομών.
- Δεν υπάρχει διαθέσιμη καμία ελεύθερη υλοποίηση λογισμικού για τη λειτουργία MCP-SC.

Με βάση τα παραπάνω, η έρευνα που διεξήχθη στα πλαίσια της εκπόνησης της διδακτορικής διατριβής έθεσε τους παρακάτω στόχους προκειμένου να συμβάλει σε πτυχές του θέματος που δεν έχουν επαρκώς διερευνηθεί:

- Εντοπισμός δημοφιλών μεθόδων PSC που μπορεί να επιταχυνθούν με την χρήση παράλληλης επεξεργασίας σε many-core και multi core επεξεργαστές που είναι διαθέσιμοι σε σταθμούς εργασίας desktop PCs και χρησιμοποιούνται ευρέως σε εργαστήρια πρωτεομικής.
- Ανάπτυξη βιβλιοθηκών αλγοριθμικών σκελετών (skeleton libraries) για τη διευκόλυνση της ανάπτυξης υλοποιήσεων παράλληλου προγραμματισμού, και χρήση των βιβλιοθηκών αυτών στην ανάπτυξη αποδοτικών υλοποιήσεων των PSC μεθόδων για τις επιλεγμένες αρχιτεκτονικές επεξεργαστών.
- Κατάλληλος συνδυασμός μεθόδων και δημιουργία consensus αποτελέσματος σε υψηλής απόδοσης MCPSC σύστημα λογισμικού το οποίο ο μέσος μη-ειδικός χρήστης θα μπορεί να χρησιμοποιεί σε ένα destop PC, χωρίς να απαιτείται πρόσβαση σε κοινούς κατανεμημένους υπολογιστικούς πόρους για υψηλή απόδοση.
- Μελέτη εν τω βάθει της επεκτασιμότητας και των χαρακτηριστικών αποδοτικότητας (efficiency), επιτάχυνσης (speedup),αποτελεσματικότητας (structural classification), του συστήματος MCPSC που θα αναπτυχθεί με την χρήση βάσεων δομών πρωτεϊνών αυξανόμενης πολυπλοκότητας.
- Δημιουργία ενός καλά σχεδιασμένου συστήματος MCPSC υψηλής ευρωστίας και απόδοσης που θα παρέχεται στην επιστημονική κοινότητα ως ανοιχτό λογισμικό για χρήση από τους ενδιαφερόμενους επιστήμονες.

- Δοκιμασία του συστήματος MCPSC που θα αναπτυχθεί σε αναλύσεις πρωτεομικής μεγάλης κλίμακας προκειμένου να εξαχθούν χρήσιμα συμπεράσματα ως προς τις επιδόσεις και τη χρηστικότητά του.

Η έρευνα που ολοκληρώθηκε με επιτυχία οδήγησε στα ακόλουθα πρωτότυπα αποτελέσματα:

Σχεδίαση και ανάπτυξη υψηλής απόδοσης MCPSC συστήματος για many-core επεξεργαστές Τρεις PSC μέθοδοι υλοποιήθηκαν στον επεξεργαστή της εταιρίας Intel Single Chip Cloud Computer (SCC), έναν πειραματικό many-core επεξεργαστή βασισμένο σε αρχιτεκτονική Network on Chip (NoC), με 48 πυρήνες Pentium, οργανωμένους σε δίκτυο τύπου mesh. Αναπτύχθηκε εφαρμογή παράλληλου λογισμικού που ενσωματώνει τις μεθόδους αυτές με δυνατότητα για λειτουργίες σύγκρισης πρωτεϊνών one-to-all, all-to-all MCPSC και μεγάλα datasets δομών πρωτεϊνών. Για την διευκόλυνση της μεταφοράς των PSC μεθόδων στο Intel SCC αναπτύχθηκε βιβλιοθήκη αλγοριθμικών ”σκελετών” που ονομάστηκε rckskel. Η βιβλιοθήκη αυτή σχεδιάστηκε για να παρέχει βασικές λειτουργίες που διευκολύνουν την ανάπτυξη υβριδικών στρατηγικών παραλληλισμού, και χρησιμοποιήθηκε για την επιτάχυνση ανάπτυξης αλγορίθμων σύγκρισης δομών πρωτεϊνών. Η χρήση αλγοριθμικών σκελετών επιτρέπει την διασύνδεση των πυρήνων many-core επεξεργαστών, όπως του SCC, με διαφορετικούς τρόπους, ανάλογα με τις ανάγκες της εκάστοτε εφαρμογής. Η ευελιξία που παρέχει βρέθηκε να είναι σημαντική για τη λειτουργία MCPSC, λόγω των διαφορών πολυπλοκότητας των διαφορετικών PSC μεθόδων που οδηγεί σε άνιση κατανομή των επεξεργαστικών στοιχείων (πυρήνων). Το τελικό αποτέλεσμα είναι μια εφαρμογή παράλληλου λογισμικού που ενσωματώνει με τον βέλτιστο τρόπο τρεις μεθόδους PSC – TMalign, CE και USM – για την αποδοτική εκτέλεση all-to-all σύγκρισης δομών πρωτεϊνών. Η εφαρμογή χρησιμοποιεί όλους τους πυρήνες του SCC (που είναι διαθέσιμοι) για να εκτελέσει παράλληλα πολλαπλές συγκρίσεις δομών πρωτεϊνών ανα ζεύγη. Η βιβλιοθήκη rckskel είναι η πρώτη βιβλιοθήκη αλγοριθμικών σκελετών που αναπτύχθηκε και είναι διαθέσιμη για many-core επεξεργαστές.

Ανάλυση Επεκτασιμότητας και Επιδόσεων

Η υλοποίηση του MCPSC για το SCC χρησιμοποιήθηκε σε σειρά πειραμάτων με σκοπό να μελετηθούν τα χαρακτηριστικά της επιτάχυνσης (speedup) που μπορεί να επιτευχθεί όσο αυξάνονται οι πυρήνες του επεξεργαστή many-core. Κάθε σύγκριση δομής ζεύγους πρωτεϊνών με μια μέθοδο θεωρείται μια εργασία (job), η οποία είναι η μικρότερη υποδιαίρεση εργασίας (grain) που μπορούμε να πετύχουμε, με αποτέλεσμα να έχουμε συνολικά N x N x M εργασίες σε μια λειτουργία all-to-all MCPSC, όπου N είναι ο αριθμός των πρωτεϊνών στο dataset και M είναι ο αριθμός των PSC μεθόδων που θα συνδυαστούν. Στις δοκιμές που πραγματοποιήθηκαν, τα μήκη των πρωτεϊνών του κάθε ζεύγους χρησιμοποιήθηκαν ως παράγοντας εκτίμησης του αναμενόμενου χρόνου για την ολοκλήρωση της σύγκρισης του ζεύγους. Προκαταρκτικά αποτελέσματα με μόνο τη μέθοδο TMalign έδειξαν ότι με την χρήση των 48 πυρήνων του SCC σε ένα κατανεμημένο setting (ο master σε ξεχωριστό πυρήνα) δεν επιτυγχάνεται ικανοποιητική αποδοτικότητα, με την επιτάχυνση να μην ξεπερνά τον παράγοντα 3x. Διάφορα πειράματα που διεξήχθησαν όμως με στρατηγικές εξισορρόπησης υπολογιστικού φόρτου (load balancing) έδειξαν ότι η δυναμική round robin ανάθεση εργασιών στους πυρήνες αποδίδει καλύτερα από την στατική διαμέριση των εργασιών (static job partitioning). Παρατηρήθηκε ότι η εφαρμογή MCPSC μπορεί να επιτύχει σχεδόν γραμμική επιτάχυνση καθώς ο αριθμός των πυρήνων αυξάνεται, κάτι που σημαίνει ότι μπορεί να παρέχει αυξανόμενους παράγοντες επιτάχυνσης σε μεγαλύτερα CPUs αρχιεκτονικής NoC που αναμένεται να γίνουν διαθέσιμα στο άμεσο μέλλον.

Περιορισμοί του fine-grained παραλληλισμού και υλοποιήσεων με FPGAs
Αναπτύχθηκαν FPGA υλοποιήσεις για δύο PSC μεθόδους – TMalign και USM, και έγινε μελέτη σκοπιμότητας για να διαπιστωθεί αν τέτοιες υλοποιήσεις μπορεί να είναι αποδοτικές. Βάσει πληροφοριών που εξηχθησαν από το profiling της υλοποίησης λογισμικού της μεθόδου TMalign, διερευνήθηκε συσχεδίαση υλικού-λογισμικού (hardware-software codesign) για την λειτουργία υπέρθεσης πινάκων (matrix superposition) του TMalign. Επιπλέον, ανάλυση της μεθόδου USM έδειξε ότι οι περισσότερες υπορουτίνες, εκτός αυτής της συμπίεσης, θα μπορούσαν να υλοποιηθούν σε υλικό. Η χρήση όμως FPGAs για την επιτάχυνση αλγορίθμων PSC βρέθηκε να είναι ανέφικτη επειδή: α) είτε τα τμήματα των PSC μεθόδων που μπορούν να εκτελεσθούν παράλληλα αποτελούν πολύ μικρό ποσοστό της μεθόδου, όπως στην περίπτωση του TMalign, είτε β) οι εφαρμογές λογισμικού είναι ήδη βελτιστοποιημένες σε μεγάλο βαθμό και δεν μπορούν να επιτευχθούν σημαντικά περαιτέρω οφέλη από υλοποιήσεις υλικού, όπως στην περίπτωση της μεθοδου USM. Διαπιστώθηκε ότι ακόμα και με το καλύτερο εφαρμόσιμο σχέδιο, μια σχεδίαση υλικού δεν θα μπορούσε να προσφέρει καλύτερη απόδοση από αυτή μιας σύγχρονης CPU λόγω των χρόνων που απαιτεί η μεταφορά δεδομένων από και προς το SoC. Από τα παραπάνω συμπεράναμε ότι δεν είναι δυνατόν να πετύχουμε για το συγκεκριμένο πρόβλημα επιταχύνσεις με χρήση fine-grained παραλληλισμού και FPGAs.

Σύγκριση many-core με multi-core CPU για λειτουργίες MCPSC
Επιπλέον αναπτύχθηκε και πολυ-νηματική (multi-threaded) εφαρμογή του λογισμικου MCPSC. Σχεδιάστηκαν πειράματα για να εκτιμηθεί πώς η επεξεργασία MCPSC κλιμακώνεται όσο αυξάνεται ο αριθμός των πυρήνων σε ένα σύγχρονο multi-core επεξεργαστή και να γίνει σύγκριση επιτάχυνσης με τους many core επεξεργαστές. Η πολυ-νηματική εφαρμογή χρησιμοποιεί το OpenMP για την διαχείριση νημάτων και κοινόχρηστη μνήμη (shared memory), αντί συστήματος ανταλλαγής μηνυμάτων που χρησιμοποιεί το rckskel στην many-core εφαρμογή. Τα πειράματα έδειξαν ότι σε quad-core i7 επεξεργαστή (με 8 hyper-threaded πυρηνες) επιτυγχάνεται επιτάχυνση μέχρι και τα 4 threads, αλλά για περισσότερα των 4 threads παρατηρήθηκε πτώση της επιτάχυνσης (απώλεια αποδοτικότητας). Η σύγκριση αποδοτικότηταςτων many-core και multi-core CPUs με βάσεις δεδομένων διαφορετικών μεγεθών, έδειξε ότι ενώ ο επεξεργαστής multi-core i7 υπερέχει σε καθαρούς χρόνους, επιτυγχάνει μόνο 4x αποδοτικότητα (σε σύγκριση ζευγών δομών πρωτεϊνών ανα δευτερόλεπτο) σε σχέση με τον many-core SCC, παρόλο που λειτουργεί σε 7.5x τη συχνότητα του SCC.

All-to-all MCPSC πολύ μεγάλης κλίμακας (big-data proteomics)
Επιπλέον, πραγματοποιήθηκε επεξεργασία MCPSC με μια πολύ μεγάλη βάση δεδομένων δομών πρωτεινών (protein domains dataset) με 3,213,631 ζεύγη πρωτεινών προς σύγκριση, για να καθοριστούν τα χαρακτηριστικά του consensus-based MCPSC. Τα πειράματα σχεδιάστηκαν να τρέξουν παράλληλα μόνο σε multi-core επεξεργαστή i7 λόγω των περιορισμών μνήμης του πειραματικού επεξεργαστή Intel SCC NoC. Ποιοτική ανά-

λυση με την χρήση RoC τεχνικών έδειξε ότι οι τεχνικές ομοφωνίας (consensus) μπορούν να δώσουν σχεδόν βέλτιστο αποτέλεσμα όταν χρησιμοποιούμε MCPSC, σε σχέση με τις επιμέρους PSC μεθόδους τις οποίες αυτό συνδυάζει. Το αποτέλεσμα αυτό επιβεβαιώθηκε και με τη χρήση της τεχνικής των κοντινότερων γειτόνων (nearest neighbor classification) . Στα πειράματα μεγάλης κλίμακας παρατηρήσαμε ότι διάφορα σκορ σύγκρισης ζευγών PSC έλειπαν από τα δεδομένα, όπως είναι λογικό λόγω εγγενών περιορισμών των PSC μεθόδων, σφαλμάτων στα PDB αρχεία κ.α. Για το λόγο αυτό μελετήσαμε και τον αντίκτυπο διαφορετικών μεθόδων συμπλήρωσης δεδομένων (data imputation methods). Τα αποτελέσματα έδειξαν ότι η συμπλήρωση δεδομένων βελτιώνει σημαντικά την αξιοπιστία των επιμέρους PSC μεθόδων και του MCPSC. Επιπλέον μια σημαντική παρατήρηση είναι ότι το MCPSC τείνει να ακολουθεί την PSC μέθοδο με την καλύτερη απόδοση, και άρα είναι απόλυτα αιτιολογημένη η χρήση του όταν δεν είναι γνωστό το ground-truth (όπως συμβαίνει στη πράξη) και με δεδομένο ότι καμία μέθοδος σύγκρισης δομών PSC από μόνη της δεν θεωρείται ολοκληρωμένη ή ανώτερη των υπολοίπων.

Ανάπτυξη ελεύθερου λογισμικού για MCPSC

Εφαρμογές διαφόρων επιμέρους PSC μεθόδων είναι διαθέσιμες στο public domain ως εκτελέσιμα binaries. Παρόλα αυτά, στις περισσότερες περιπτώσεις οι εφαρμογές αυτές είναι μονο-νηματικές και δεν εκμεταλλεύονται τον παραλληλισμό των σύγχρονων επεξεργαστών. Το παράλληλο λογισμικό που αναπτύχθηκε για MCPSC στο πλαίσιο της διατριβής και παρέχεται ελεύθερα στο GitHub είναι το πρώτο και μοναδικό παγκοσμίως τέτοιο λογισμικό που έχει βελτιστοποιηθεί για multi-core και many-core CPUs.

Τα αποτελέσματα της διδακτορικής διατριβής οδήγησαν στις παρακάτω δημοσιεύσεις:

Αρθρα σε Περιοδικά:

1. **Anuj Sharma**, Elias S. Manolakos, ”Efficient multi-criteria protein structure comparison on modern processor architectures”, *BioMed Research International*, Volume 2015 (2015), Article ID 563674, 13 pages, doi:http://dx.doi.org/10.1155/2015/563674.
2. **Sharma A**, Manolakos ES, ”Multi-criteria protein structure comparison and structural similarities analysis using pyMCPSC.”, *PLoS ONE*, 2018, 13(10): e0204587. https://doi.org/10.1371/journal.pone.0204587

Αρθρα σε Πρακτικά Συνεδρίων με κρίση Πλήρους Κειμένου:

1. **Anuj Sharma**, Antonis Papanikolaou, Elias S. Manolakos, ”Accelerating all-to-all protein structure comparison with TMalign using NoC many-cores processor architecture”, *In Proceedings of the IEEE 27th International Symposium on Parallel & Distributed Processing Workshops*, May 2013, pp 510 – 519, Cambridge, MA.
2. Brueffer, C., Antao, T., Cock, P., Talevich, E., Hoon, M. d., Arindrarto, W., Pritchard, L., **Sharma, A.**, Rasche, E., Rosenfeld, A., Skennerton, C. T., Galardini, M., and Piotrowski, M. (2016). Biopython project update 2016, Bioinformatics Open Source Conference 2016, Orlando, USA, Department of Clinical Sciences, Lund University, DOI:10.7490/f1000research.1112611.1

To the loving memory of my mother

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to all the people who made my PhD years a truly enriching experience. At an academic level, I had the opportunity to interact with some wonderful scholars. At a personal level, I met wonderful people who provided support and encouragement during difficult moments.

I am greatly indebted to my advisor, Dr. Elias Manolakos for being a source of constant inspiration and motivation. Without his guidance this thesis would not have been possible. I would also like to thank the members of my advisory committee, Dr. Ioannis Emiris and Dr. George Panayotou, for their time and valuable feedback.

I would also like to acknowledge the MicroLab - ECE - National Technical University of Athens, Greece for allowing generous use of their SCC infrastructure. In addition I thank Mr. Dimitrios Rodopoulos and Dr. Dimitrios Soudris of NTUA for providing comments and assistance at opportune moments. I would like to acknowledge the European Union (European Social Fund ESF) and Greek national funds, for the Operational Program ”Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) - Research Funding Program ”Heracleitus II”, through which this work was financially supported.

I am grateful to our lab members Evangelos Logaras and Elias Kouskoumvekakis for their insightful feedback on FPGA related aspects of my research work.

Finally, I would like to thank my wife for her constant support and encouragement and my father, brother and sister for always believing in me.

— Anuj Sharma

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This research has been co-financed by the European Union (European Social Fund-ESF) and Greek national funds through the Operational Program ”Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF). Research Funding Program: Heracleitus II. Investing in knowledge society through the European Social Fund.

1. INTRODUCTION

For the last three decades the comparison and alignment of protein structures has been used extensively in computational biology [19], because naturally occurring protein fold in three dimensional space and the resulting structure has a strong correlation to its function [63]. Conservation of proteins is known to be much higher at the structure than at the sequence level, therefore structural similarity is essential in assigning functional annotations to proteins [80]. Function assignment is typically achieved by developing a template of the functional residues of the proteins and then aligning the template with complete known structures [101]. Structural comparison approaches are also increasingly employed in drug repositioning [45]. Protein Structure Comparison (PSC) methods are used to identify proteins with similar binding sites all of which then become potential targets for the same ligand [99, 39]. All these important applications require the structure of one or more proteins (queries) to be compared against a large number of known protein structures (one-to-all or many-to-many type comparison) to identify protein pairs with high structural similarity.

Given that proteins can be very long chains of amino acids the problem of aligning two protein structures has been shown to be NP-Hard [95]. As a result, commonly employed pairwise PSC methods make use of heuristics [123, 24, 56], generating alignments in addition to a similarity score in most cases. Over the years many methods have been proposed for pairwise PSC. These PSC methods vary in algorithmic techniques employed including but not limited to Graph based techniques [140], Dynamic programming based techniques [86, 47], Information Retrieval based techniques [36], Geometric techniques [68] and Contact map based techniques [65]. Further, they also vary in terms of the similarity metrics used and yield different but biologically relevant results [48, 69, 38]. There is therefore no consensus on a single method that is superior for protein structure comparison [13].

A newly discovered protein structure may be used to classify the protein in order to ascertain its functional and behavioural characteristics based on properties of other proteins in its class. While manual curation of the classification is possible this is a slow process and cannot keep up with the increase in the number of known proteins [113]. In such a scenario, the structures of one or more query proteins are compared to those of database proteins whose function, homology etc. are known (one-to-many and many-to-many PSC). Based on the comparison scores, structurally similar database proteins can then be used to determine the properties of a query protein such as drugs that may interact with it. With the advances in high-throughput technologies the number of known protein structures is growing rapidly [128]. This is reflected in the size of the Protein Data Bank (PDB), a repository of 3D structural data of proteins [83], as shown in Figure 1.1. It is clear that the number of protein structures continues to grow exponentially, both in the Structural Classification of Proteins (SCOP) [67] and CATH [121] databases, while simultaneously the number of new protein structures added per year also shows an upward trend.

Computational demands in PSC are therefore a result of three pertinent features. Firstly,

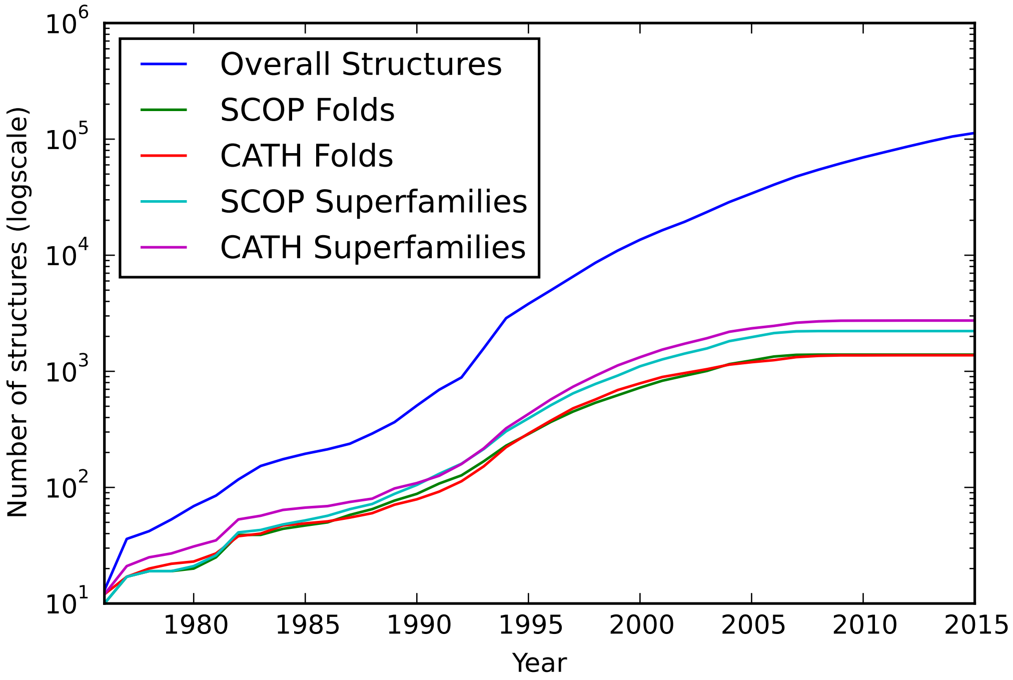


Figure 1.1: Statistics of protein holdings in the Protein Data Bank (PDB), data taken from RCSB PDB. Data for Year 2015 is incomplete.

pairwise PSC has high computational complexity due to the NP-Hard nature of the problem. Secondly, classification of newly discovered protein structures, for the purposes of ascertaining functional properties, requires comparison with large and fast expanding databases [28]. Thirdly, the lack of consensus on a single method has led to a trend in the domain to provide results from more than one structure comparison method [15]. The advantage of such an approach, Multi-criteria Protein Structure Comparison (MCPSC), is that it does not call for determining the superiority of an approach over another, but integration of several protein structure comparison methods into a unified tool. The approach banks on the idea that an ensemble of classifiers is likely to yield better performance than any of the constituent classifiers [61].

In published literature instances of the use of distributed computing platforms, such as clusters of workstations (COWs), computer grids and cloud, can be found for meeting this computational demand [113, 13, 111, 82]. One such implementation, available for use to the community, the ProCKSI server [13], is an online resource for performing all-to-all MCPSC experiments. Results of the experiments returned to the user include individual PSC method scores as well as a consensus MCPSC score. While the ProCKSI server provides an excellent one stop resource for designing and running all-to-all MCPSC experiments, it is limited in the size of the data (upto 250 protein domains) and is not extendable with PSC methods of users choice. In general, distributed solutions, specifically Grids, suffer from problems such as extensibility, maintainability and fault tolerance.

On the other hand, emerging parallel architectures, such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), multi- and many-core Central Processing Units (CPUs) have not been extensively used in the domain. Multi-core and many-core CPUs, as opposed to FPGAs and GPUs, retain backward compatibility to well established programming models [131] and offer the key advantage of using programming methods, languages and tools familiar to most developers. Many-core processors differ from their multi-core counterparts primarily on the communication subsystem. Many-core CPUs use a Network-on-Chip (NoC) while multi-core CPUs use bus-based structures [17]. While multi-core CPUs are ubiquitous and easily available for parallel processing, due to the drive from leading chip manufacturers over the past several years [18], many-core CPUs are not as deeply entrenched or commonly available yet. However, due to architectural improvements many-core processors have the potential to deliver scalable performance by exploiting a larger number of cores at a lower cost of intercore communication [9].

These parallel processing architectures have become more readily available [94, 10], Figure 1.2, and instances of their use are beginning to appear in the broader field of biocomputing [106, 58]. These architectures can in principle be used additively to meet the ever increasing computational demands of MCPSC by complementing already in use distributed computing approaches [112]. It is therefore critical that effort be expended to utilize this desktop scale parallelism [22, 100]. This will allow problems of a scale that could only be tackled by distributed platforms, to be carried out on commodity hardware and perhaps even more efficiently [30]. In particular, any optimisation made for utilising the parallelism of multi-core and many-core CPUs naturally adds to the distributed infrastructure which will soon, if not already, be using combinations of multi-core enabled nodes. Additionally, the familiar shared memory programming model makes it easier to utilize the parallel processing capability of these processor architectures as compared to Graphics Processing Units (GPUs) [131].

As established, protein structure comparison is a well developed field of research with a large body of published literature and active current interest with new techniques being developed continuously. We point out some of what we believe to be open challenges for large-scale MCPSC:

- *Efficient use of modern parallel architectures*: Existing works focus on improving pairwise PSC or on application of distributed resources to the many-to-many PSC scenario. However, the parallelism afforded by modern parallel architectures has not been extensively explored or efficiently utilized especially for many-to-many PSC.

- *Biologically Relevant Clustering and Classification*: Defining and identifying classification of different protein structures is still an unresolved problem. Structured analysis of gain from Multi-criteria PSC towards resolving this problem is needed.

- *Leveraging Structure Comparison*: Generating large-scale MCPSC results currently requires large distributed resources. Since a higher amount of computational power is available locally to researchers, software needs to be made available so that they

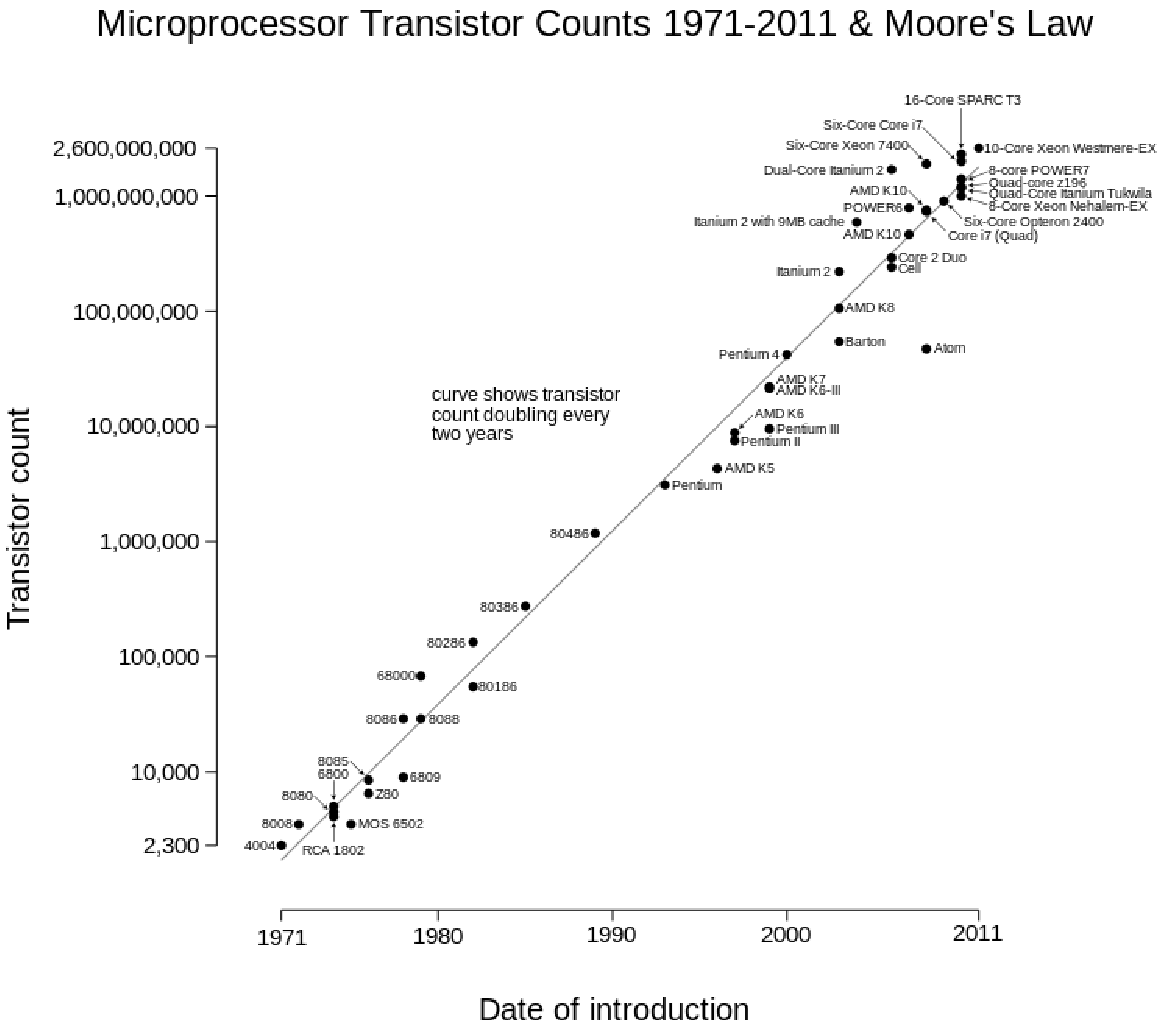


Figure 1.2: Transistor counts for integrated circuits plotted against their dates of introduction. The curve shows Moore's law - the doubling of transistor counts every two years. Towards the right end of the curve it can be seen that leading processor manufacturers are shifting to multi-core and many-core architectures. Taken from [122].

can leverage such results in their every day work. Thus, effort is needed to deliver high quality easy to use software that can utilize desktop parallelism.

Development of solutions driven by these open issues would benefit research focused on structure classification systems such as SCOP etc. by enabling large sets of proteins to be automatically classified and categorised. Another group of researchers who would benefit are those engaged in determining protein function from structure. If biologically relevant groupings of proteins can be identified automatically and with high accuracy, researchers would be able to quickly compare a protein to a large set to determine its function. Further, researchers engaged in protein structure prediction and evaluation would benefit because of the utility of PSC for validating structure prediction results. Finally, availability of software that allows maximum utilisation of local computational power, at the disposal of researchers, will have a broad impact for the community as a whole.

Based on the above observations, with the aim of performing a comprehensive study of efficient solutions for Multi-criteria Protein Structure Comparison, our research design contained several objectives. Firstly, identify PSC methods conducive to parallel implementations and architectures suitable for such implementations. Secondly, implement ports of PSC methods for these architectures and where possible develop reusable libraries. Thirdly, compare and contrast the performance of these implementations for performing all-to-all MCPSC. Fourthly, use the implementations to perform large-scale proteomics experiments and carry out a structured analysis of the qualitative aspects of consensus based MCPSC. Finally, deliver a well designed suite of parallel MCPSC software for use by the broader community.

In this work, three PSC methods – TMalign [143], CE [119] and USM [59] – were ported on the Intel Single-chip Cloud Computer (SCC) [74], a Network on Chip (NoC) based many-core processor with 48 Pentium cores organized in a mesh network. In order to facilitate porting PSC methods to the Intel SCC an algorithmic skeleton [20] library called *rckskel* was developed. The library was designed to provide basic functions needed for exploring hybrid parallelization strategies, for speeding up protein structure comparison algorithms. Use of algorithmic skeletons allows the processing elements of a many-core processor, such as the SCC, to be connected in arbitrary order as per the requirement of the application. This flexibility was found to be important for MCPSC because of differences in the complexity of PSC methods requiring unequal distribution of the processing elements. The result was an application that uses the three PSC methods to perform all-to-all protein structure comparison. The application makes use of all SCC cores (available at run-time) to run multiple pairwise PSCs concurrently.

The MCPSC implementation for the SCC was used to conduct several experiments to study its speedup characteristics. Each pairwise PSC with a single method was considered as a job, which is the most fine-grained work distribution setup that can be achieved, resulting in $N \times N \times M$ jobs, where N is the number of proteins and M is the number PSC methods. In the experimental setup the lengths of the pair of proteins were used as a factor indicating the expected time complexity of performing pairwise PSC. Preliminary results, with TMalign alone, showed that using the 48 cores of the SCC in a distributed setting (master running on a separate processor) is not efficient, giving a speedup lower than a factor of 3x. Several experiments conducted with load balancing strategies revealed that dynamic round-robin job distribution outperforms static job partitioning schemes. It was observed that a near linear speedup can be achieved as the number of slave-cores is increasing, which suggests even higher speedups are possible with bigger NoCs [114].

Prototype FPGA implementations of two PSC methods – TMalign and USM – were also developed. A feasibility study was conducted to ascertain if such implementations are viable. Based on profiling information obtained from the software implementation of TMalign a software-hardware co-design was developed with the matrix superposition subroutine identified for hardware implementation. Analysis of the USM method showed that most subroutines, apart from the compression subroutine, could be implemented in hardware. Using FPGAs for speeding up PSC was not found to be feasible because: a) either the parallelizable parts of PSC methods are a very small factor of the overall method as in the case of TMalign, or b) the software implementations are already highly optimized and no significant gains can be achieved from the hardware implementations, as in the case of USM. However, even with an optimal implementable design, a hardware implementation would not provide gains over using a modern CPU because of the data transfer times to and from the SoC. We therefore concluded that fine-grained parallelism using FPGA is not feasible for achieving high speedups for all-to-all PSC. A byproduct of our initial experiments into the development of an efficient superposition subroutine for the FPGA, was a Python module for structure superposition, QCPSuperimposer, which was submitted and accepted as a module in BioPython v1.66 [29].

A multi-threaded implementation of the MCPSC software was developed for comparison with the many-core implementation. Experiments were designed to assess how the MCPSC problem scales with increasing number of cores on a modern multi-core CPU and to compare this with the speedup observed on a many-core CPU. The multi-threaded implementation makes use of OpenMP for introducing threads and uses shared memory constructs to replace the message-passing communication handled by *rckskel* in the many-core implementation. Our experiments showed that, when running on a quad-core Intel i7 (with 8 hyper-threaded cores), speedup is observed up to the 4 threads configurations thereafter a steep speedup drop (efficiency loss) is observed. This was attributed to the fact that the work-load placement is not suited to taking advantage of the super-scalar structure of the CPU which requires varied workloads to deliver higher performance. Comparison of the many-core and multi-core CPU throughputs, on several datasets of varied sizes, showed that while the i7 is superior in raw times it only achieves a 4x throughput (in terms of pairwise protein structure comparisons per second) as compared to SCC while it runs at 7.5x the frequency [115].

Finally, experiments with a very large protein domain dataset (3,213,631 protein domain pairs) using a utility we developed, named *pyMCPSC*, containing five PSC methods - TMalign, CE, USM, Fast [147], GRalign [72] - were carried out to determine the characteristics of consensus-based MCPSC. The experiments were designed to run only on a multi-core CPU due to the memory limitations of the Intel SCC many-core processor. Qualitative analysis using Receiver Operating Characteristics (RoC) curves revealed that simple consensus schemes, such as using the average score, resulted in MCPSC performing near-optimally in comparison to its component PSC methods. This was also validated in terms of its structural classification ability using a Nearest Neighbor approach. In the experiments we observed that several pairwise PSC scores were missing due to inherent limitations such as errors in PDB files and runtime environment differences (for binary PSC software). We therefore designed experiments to study the impact of imputing pairwise PSC data. Our experiments show that imputing data can improve the ability of PSC methods (including MCPSC) for structural classification of domains in a many-to-many comparison setup. A key observation of these experiments was that MCPSC closely follows the best performing PSC method, which is of importance in the absence of the ground-truth in a domain where no single method is considered to be complete or superior [116]. Finally, we visualize the dataset in domain and topology spaces and show that such visualizations reveal interesting information about presence of correlations and clusters within a large dataset.

Contributions

Following are the main contributions of this work:

- Design and implementation of a parallel computing strategy for the all-to-all MCPSC problem on many-core CPUs. This work included the development of an algorithmic skeletons library *rckskel* for the Intel SCC Network on chip. To the best of our knowledge this is first attempt to use many-core CPUs for this task.

- Scalability analysis of all-to-all MCPSC on the Intel SCC. We compare several load balancing strategies and find that the dynamic round-robin performs the best. Further, it has been shown that a near linear speedup, with respect to number of cores used, to be achieved for the all-to-all comparison task.

- Design and implementation of a multi-core version of the all-to-all MCPSC software and comparison to the many-core implementation using several protein structure datasets of varying sizes. We show that while a modern multi-core CPU outperforms the many-core CPU on raw speedup times, the latter possesses some very desirable qualities making it competitive on efficiency.

- Analysis of using consensus based MCPSC score for clustering and classification of proteins using a very large dataset. We show that MCPSC has near optimal characteristics with respect to its component methods both for clustering and for classification. Further, our results with imputed data can form the basis of dealing with missing data problem to be expected in real world large scale experiments.

- Visual analysis of large domain dataset using MCPSC score. We show that Multidimensional Scaling, Heatmaps and ‘Phylogenetic Trees’ can be used to explore visually the structure of a very large dataset and determine functional correlations between the dataset domains.

- Development of public domain software for parallel MCPSC on multi- and many-core CPUs. All the software that we have developed for both these architectures has been made available via open-source projects including the MCPSC software repository, the large scale MCPSC utility and a module in BioPython. To the best of our knowledge there are no other easily accessible software resources for carrying out large-scale MCPSC experiments available publicly.

1.1 Outline of the Thesis

The remainder of this thesis is organized as follows:

Chapter 2 Introduces several concepts necessary for understanding protein structure comparison. The similarity metrics used, features extracted and representations used are presented. A formulation of the pairwise, one-to-many and many-to-many PSC is also presented.

Chapter 3 Gives a detailed account of the techniques currently found in literature for protein structure comparison. The review includes both serial and distributed implementations. Modern parallel architectures considered in this work are also discussed.

Chapter 4 Presents an algorithmic view of PSC and discuses the characteristics of pairwise PSC and the scope for introducing parallelism. A model for parallel MCPSC is also developed with the help of specific PSC methods.

Chapter 5 Describes in detail the algorithmic skeletons library and the ports of PSC methods developed using the library for the Intel SCC. The programmatic API of the library is described and the overall framework under which multiple PSC methods were ported is discussed.

Chapter 6 Describes the results of load-balancing experiments carried out on the Intel SCC.

Chapter 7 Describes the comparative performance of multi- and many-core implementations for all-to-all MCPSC using several datasets.

Chapter 8 Describes the algorithmic and software architecture of *pyMCPSC*.

Chapter 9 Describes the results of qualitative analysis of consensus based MCPSC using a small and a very large dataset

Chapter 10 Presents a brief overview of the work and possible future direction.

2. PRELIMINARIES

Protein structure comparison is an established research field with a lot of current interest. As such there are several terms and expressions specific to it. In this section we introduce these terms. We also present a brief overview of various metrics used by related methods.

2.1 Protein Structure

Proteins are long polymers containing several atoms which create a repetitive backbone with side-chains attached to each residue, as shown in Figure 2.1. Small portions of a protein, called active sites, are of importance to the function of the protein [64], while the rest of the chain participates in creating the geometry of the protein structure. There are twenty naturally occurring amino acids that make up proteins [54], these are shown in Figure 2.2. In its natural form a protein adopts a unique three-dimensional shape, also known as its *tertiary* structure. The tertiary structure of a protein determines its accessible active sites, thus determining its functions.

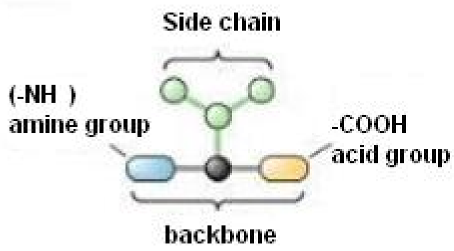


Figure 2.1: Protein backbone and side-chain. Taken from [64]

In three-dimensions the polypeptide chain folds into a curve determined by the order of the residues. Several folding patterns exist which share some common structural features, including *alpha-helices* and *beta-strands*, known as Secondary Structure Elements (SSEs) [64]. Figure 2.3 shows the three most commonly occurring SSEs. These local shapes (SSEs), over the entire polypeptide chain, together result in the tertiary structure of the protein, as shown in Figure 2.4.

2.2 Similarity Metrics

Most similarity metrics can be divided into two types: those that compare the distance between corresponding pairs of atoms in two structures and those that compare the relative positions of atoms of two superimposed protein structures [48]. Most commonly used scoring schemes fall in one of the following three categories: Root Mean Square

Deviation (RMSD) of rigid-body superposition, distance map similarity and Contact Map Overlap. These methods use the distance between residues to determine similarity of protein structures. Following is a list of commonly used similarity metrics used in protein structure comparison:

1. RMSD (Root Mean Square Deviation): is a measure of the average distance between atoms of two superimposed proteins. Methods using backbone comparison rely on RMSD between proteins followed by objective function minimization. These methods are typically computationally expensive. Several algorithms [95, 57] can be found in literature using this measure for protein structure comparison.

2. URMS (Unit vector RMS): is a variant of RMSD. URMS captures the difference between the global orientation of vectors at corresponding alpha-carbons rather than the difference between the coordinates of the alpha-carbons themselves. It has been argued that this metric is more robust in finding global matches than RMSD [141]. In [141] the authors use URMS for protein structure comparison.

3. AFP (Aligned Fragment Pairs): is a metric based on alignment of fragment pairs. Protein sequences may be decomposed into small runs of amino-acids forming what are known as fragments. Each fragment may be an alpha-Carbon or a portion of the sequence that forms an SSE. In some methods [140, 117] for structure comparison similarity, metrics are built based on pairs of such fragments −one from each protein being compared− aligned to reduce RMSD. These aligned fragments are known as aligned fragment pairs.

4. Z-score: represents the accuracy of a match between two protein structures. It is the likelihood of obtaining a similar alignment of a given protein with any random protein structure with the same composition and length. One of the values returned by the popularly used DaliLite [46] algorithm for a pair of protein structures being compared is the Z-score.

5. TM-score: measures pairwise similarity in topologies of protein structures. TM-score is biased towards close matches, as opposed to distant structures, hence it is more sensitive than RMSD for close matching structures. TM-score has been used as an alternative to RMSD [86].

6. Phenotypic Plasticity Measure (PPM): models the evolutionary distance of a pair of proteins at the structural level. It measures the cost of ‘morphing’ one structure into the other. PPM inherently models the naturally observed structural variance of proteins while retaining overall topology. The metric was proposed and used in [34].

2.3 Features of Proteins

Protein structures can be described by features which fall in the following three categories: geometric, topological and physico-chemical. Techniques proposed for protein structure comparison differ in the feature vectors on which the structures are compared [54]. The features themselves capture different aspects of the protein structure and as such have resulted in algorithms with varying degree of success on ”complete” structure comparison. Following is a brief description of the aforementioned feature categories:

- Geometric features: capture the structure of a protein via coordinates, relative positions of atoms, residues, fragments and SSEs [103].
- Topological features: capture the order of the elements along the backbone of the protein [93].
- Physico-chemical features: capture the chemical properties of a protein resulting from its amino-acid sequence. Some methods [34] take advantage of these features to perform structural comparison.

2.4 Representation of Protein Backbone

An aspect that differentiates structure comparison algorithms is the manner in which the proteins are represented. The basic units in protein representation vary from atoms of the chain to ”virtual” geometric cells in space. Each representation can only be used with a subset of the features described in section 2.3. For example consider partitioning the space in which the protein exists into Voronoi cells [16]. Such a division does not lend itself to extracting chemical properties to form feature vectors. Protein structure description methods can be grouped into (a) element based, such as atoms, residues, fragments, SSEs and (b) space based, dividing the proteins 3D space into cells.

The representation of the protein, the assembly of feature vectors to form the protein, can determine the type of comparison algorithms that may be feasible. As an example, a string representation of the structure is more amenable to a string comparison based algorithm. Following is a list of structural representations that can be found in protein structure comparison literature:

- List of unit descriptions: Represent the protein as a list of features that describe position. Techniques in literature can be found that represent the protein as a list of coordinates of alpha-Carbons [127], a list of mean coordinates of side chains [52], a list of two pseudo atoms [8], a list of unit-vectors between alpha-Carbons [26] etc.
- Set of unit descriptors: This representation is similar to the list of unit descriptions, however the ordering of the list is irrelevant. Techniques based on this representation [4] are capable of finding similarity among proteins which are sequence independent.
- Graphs: Represent the protein as a labeled graph with residues as nodes and edges representing relations. Residues can be alpha-Carbons, SSEs etc. Several techniques based on graphs can be found in literature [133, 140].

Table 2.1: Public domain resources for protein structure download, visualization and comparison

Resource	**Description**
PDB	Repository of protein structures
EMBL-EBIMSD	European repository of protein structures
PDBSum	Summary of PDB file structure analysis
SCOP	Manually curated classification database derived from PDB
ASTRAL	Database derived from SCOP
ModBase	Database of 3D protein models generated by comparative modeling
OCA	Integrated database of protein structure and function
PDBWiki	Annotated knowledge base of biological molecular structures
Proteopedia	3D encyclopedia of proteins and other molecules
ProCKSI	Decision support for protein structure comparison

- Contact Maps: They represent the distance between all residue pairs in the protein structure by a binary two-dimensional matrix. For two residues i and j, the ij element of the matrix is 1 if the residues are closer than a predetermined threshold, and 0 otherwise. Techniques based on this representation [98] have been shown to be fast and efficient for protein structure comparison.

- Feature Arrays: Techniques based on this representation [71] make heavy use of matrix algebra to perform comparisons. Typically each protein is represented by a row in the array, with the columns forming the dimensions of the feature vector.

- Strings: Represent the protein structure as a string of alpha-Carbon relative positions. An approach using such a representation can be found in [144].

Several resources for downloading, visualizing and comparing 3-Dimensional structure of proteins, available in public domain, are listed in Table 2.1.

2.5 Problem Formulation

Pairwise protein structure comparison consists of two distinct steps: aligning the structures of the proteins to be compared and generating a similarity score to the aligned pair. Structure alignment can be avoided in some cases where the algorithm is designed generate only the comparison score.

Aligning the structures of the proteins involves finding all rigid-body transformations of the protein structures. Many algorithms exist in literature, as will be seen in later chapters, for performing this step. Generating the similarity score involves finding a correspondence such that the distance between the two structures is minimized. Since the correspondence between individual atoms of the proteins is not known a priori, a variety of algorithms and similarity metrics have been applied to the problem.

As an example consider RMSD based protein structure comparison. An all versus all comparison of alpha-Carbons contained in the two proteins must be performed. The result is a N x M matrix, where N is the number of backbone carbons in the first protein and M is the number of backbone carbons in the second. Using such a matrix it is possible to determine the optimal alignment of the two structures, minimizing the distance between them or maximizing the overlapped carbons. Equation 2.1 shows how coordinate RMSD is calculated where n is the number of aligned residues between the two proteins ($n \leq N$ and $n \leq M$) and $(x(1), y(1)), ..., (x(n), y(n))$ are coordinates of the corresponding residues from the two proteins.

$$RMSD = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x(i) - y(i))^2} \tag{2.1}$$

A more common scenario in practice is to find proteins most similar to one or more query proteins from a database of known proteins based on the structure. Thus in practice researchers are interested in comparing one (or more) protein structure(s) with a database of protein structures, we call this one-to-many (many-to-many) protein structure comparison. The term PSC is typically used to imply one of these scenarios. A succinct representation of this typical scenario is shown in Equation 2.2, where Q is the set of query proteins, D is the set of database proteins and F represents the PSC method being used. The result is a set of pairwise structural similarity scores which can then be sorted to find highly structurally related pairs of proteins.

$$F(q, d) \quad \forall \quad \{(q, d) \quad | \quad q \in Q \quad and \quad d \in D\} \tag{2.2}$$

On occasion a researcher is interested in finding structurally similar clusters within a dataset. This is a specific case of the many-to-many scenario where all dataset proteins are compared (pairwise) against each other (Q = D). We refer to this as all-to-all PSC. It may be noted that the result set of Equation 2.2 forms the cells of a square matrix, with each cell representing one pairwise PSC score for the all-to-all PSC scenario. Depending on the specific requirements it may be sufficient to calculate PSC scores for pairs in the lower or upper triangular part of the matrix because in general $F(q, d)$ = $F(d, q)$. In this work we use both these types of all-to-all structure comparison setups a) where the full matrix is calculated and b) where only the triangular matrix is calculated. We use case (a) where we perform comparative study of our solutions in terms of processing time and case (b) where cluster analysis is performed.

2.6 Datasets

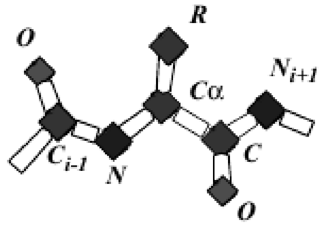
Several datasets, listed in Table 2.2 were used in this work. The table includes statistics about the length of the protein domains and the distribution of the SCOP [67] classifications in each dataset. It can be seen that there is significant variation in the sizes of the datasets, the lengths of protein domains they contain as well as the distribution of the SCOP classifications. For each dataset we retained domains where the PDB file could be downloaded and a classification could be found for the domain in SCOP v1.75. Further, the scripts available for download with the *USM* sources were used to extract the contact maps from the PDB files. The Chew-Kedem (CK34) and the Rost-Sander (RS119) datasets, were used to develop the load balancing schemes for the SCC as well as to study the Speedup (Throughput) on the i7 processor. Further, all the datasets were used to compare the performance of the SCC with that of the i7 on all-to-all MCPCS processing. In all the datasets the structures included, referred to as 'domains', are the leaves of the SCOP hierarchy tree which are structural domains of individual PDB entries.

Table 2.2: Basic statistics of the domains in the datasets used in this work. The table includes the number of SCOP Families, Superfamilies (SpFams) and Folds as well as the Total number of domains (No.), the Minimum (Min), Maximum (Max), Median, Mean and Standard Deviation (Std) of the domain lengths in each dataset.

Dataset	No.	Domain lengths					SCOP classifications		
		Min	Max	Median	Mean	Std	Families	SpFams	Folds
Skolnick [139]	33	97	255	158	167.7	62.7	5	5	5
Chew-Kedem [27]	34	90	497	147	179.5	100.1	10	9	9
Fischer [41]	68	62	581	181	220.6	125.6	56	44	40
Rost-Sander [102]	114	21	753	167	193.2	123.4	93	85	71
Lancia [21]	269	64	72	68	67.9	2.4	79	72	57
Proteus [7]	277	64	728	239	247.6	116.7	53	47	41

The Gold-standard benchmark dataset, introduced in [35] was also used in this work to carry our very large scale MCPSC experiments. The dataset contains protein domains that are consistently defined in both SCOP v1.75 and CATH v3.2.0 (i.e.with domain overlap greater than 80%) and that share less than 50% of sequence identity. Further, the benchmark only considers domain pairs that are consistently classified across the SCOP fold classification and the CATH topology classification. The dataset consists of 6759 unique domains and defined 3,213,631 domain pairs (similar and non-similar sets combined). The dataset consists of 11 (4) Classes, 792 (780) Folds and 1348 (1550) Superfamilies according to the SCOP (CATH) classification databases.

(a) *Geometry of an Amino Acid*



(b) *Amino Acid Side-chains:*

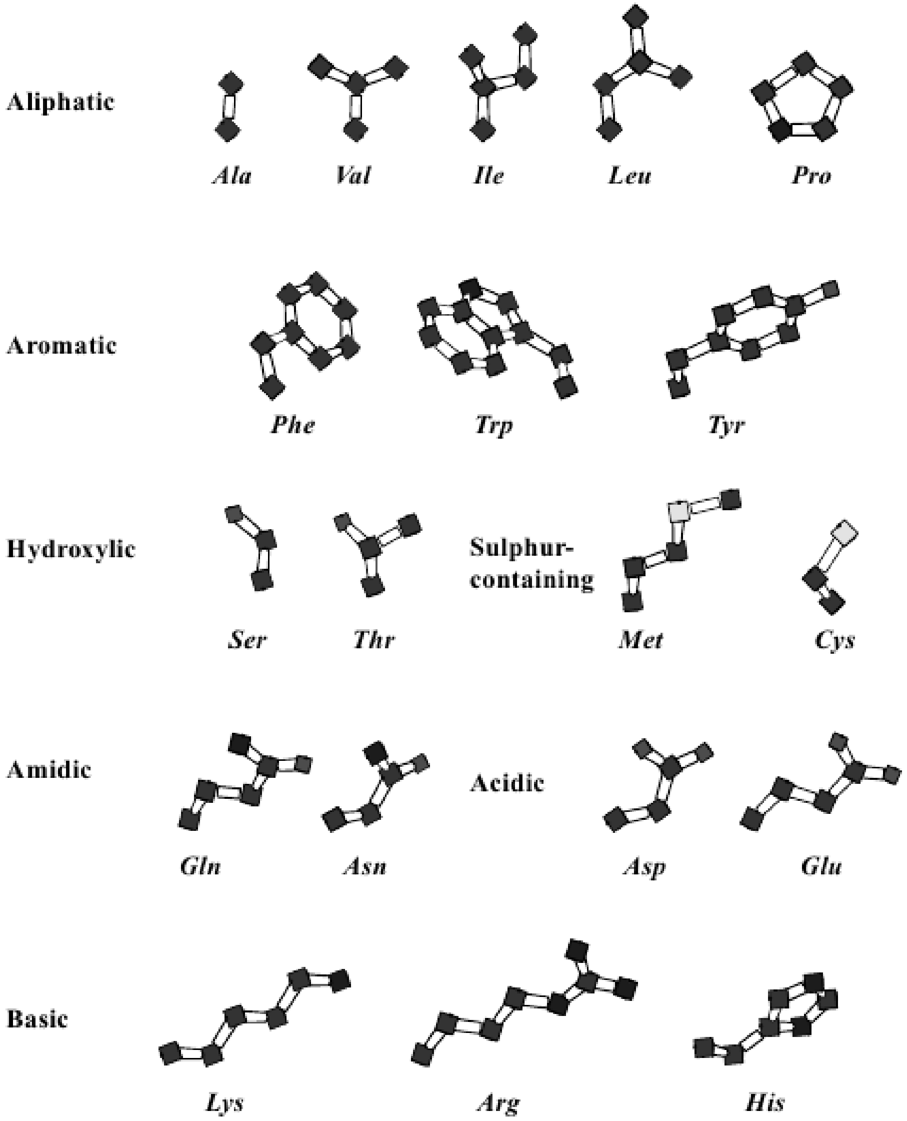


Figure 2.2: The twenty amino acids that make up proteins. Taken from [54], (a) An amino acid consists of a main chain (N, C_{α}, C and O) along with a side-chain R. (b) Classification of side-chain R based on their chemical properties.

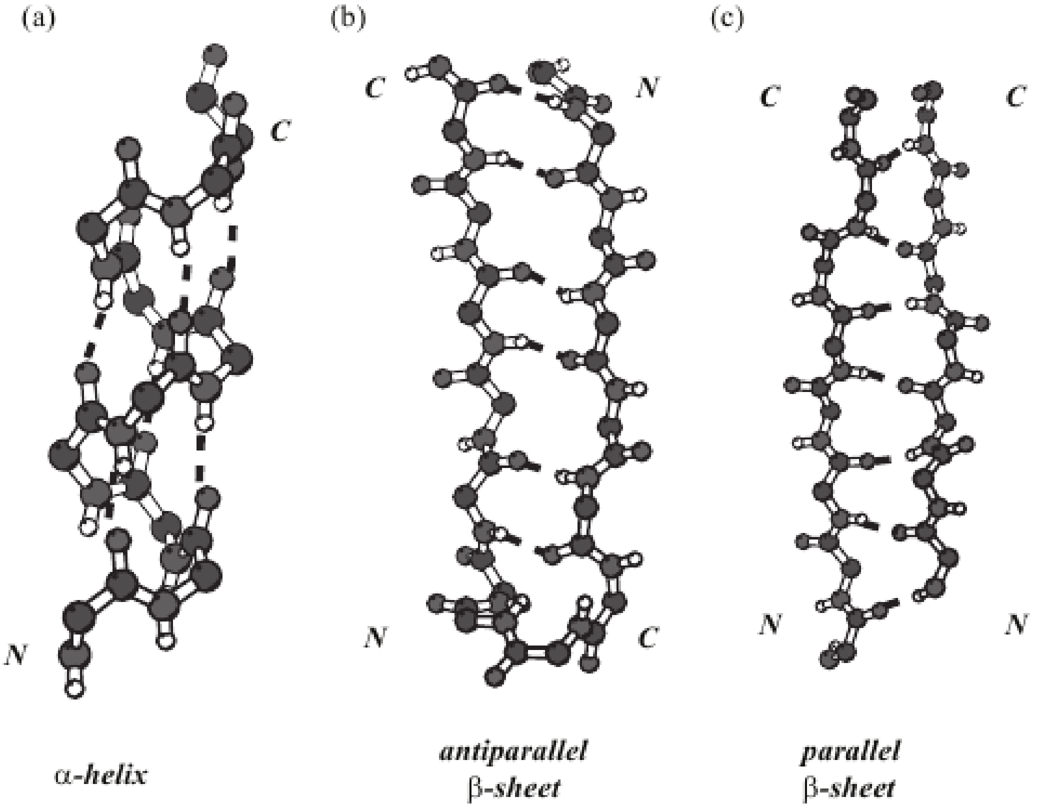


Figure 2.3: Commonly occurring Secondary Structure Elements (SSEs). Taken from [54].

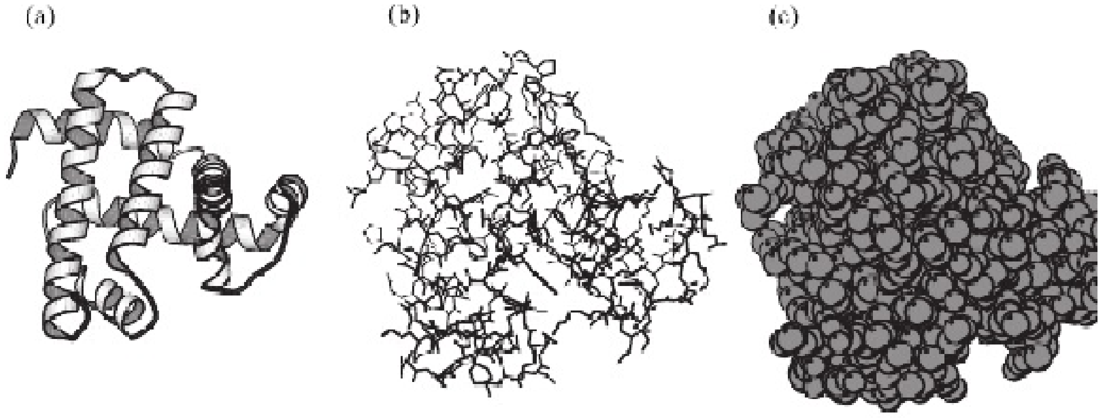


Figure 2.4: Computer aided protein visualization.

Computer aided protein visualization. Taken from [54]. (a) **Cartoon**. This representation also referred to as the ”ribbon” diagram provides a high level view of the proteins secondary structure elements. (b) **Skeletal Model**. This model represents bonds bylines. (c) **Space-filling diagram**. In this model atoms are represented by balls centered at the atom with radii equal to van der Waals radii.

3. BACKGROUND AND RELATED WORK

Several methods have been proposed for performing protein structure comparison. Proposed methods, as can be found in literature, vary vastly in terms of the algorithmic approach, feature selection and comparison metric used. In this chapter a comprehensive review of the published material, both for serial and distributed/parallel architectures, is presented.

3.1 Pairwise protein structure comparison

In this section we present a substantial collection of serial pairwise PSC methods and categorize them based on the algorithmic approach used to simplify the task of reviewing the literature. The list of methods included is by no means extensive but rather is intended to include important contributions in the category.

3.1.1 Graph Based Techniques

In [134] the authors present a method for enhancing the TOPS graph model with structural and biochemical features. The augmented model is reported to perform better than the TOPS graph model. The advanced TOPS+ proposed method gives a 6% increase in accuracy over TOPS and TOPS+ methods on the SCOP dataset. The proposed approach also achieves 98% accuracy on the Chew-Kedem dataset, which is higher than TOPS, basic TOPS+ and SSAP methods.

A greedy non-sequential protein structure alignment algorithm, called FlexSnap, is proposed in [105]. A global alignment is assembled from short, well-aligned fragment pairs. AFPs are generated by pairing alpha-Carbons of the two proteins which satisfy an RMSD based similarity constraint. A high scoring subset of AFPs is then derived by greedily finding the maximum weighted clique in a graph where the AFPs form the vertices. FlexSnap shows the highest agreement at 79%, with the curation in the RIPC dataset as compared to DALI.

In [147] the authors develop an algorithm, called FAST, in which they formulate optimal alignment between two protein structures as a problem of finding the maximal clique in a pair of graphs. Nodes of the graph represent the possible pairings of the alpha-Carbons between the two structures and the edges denote compatibility between the pairs. An edge exists if the distance between the paired alpha-Carbons is less than a cutoff value. To solve the NP-hard problem of finding the maximum clique, a series of empirical rules are used to reduce the graph until a reasonable approximation is possible. In order to test the overall accuracy of FAST, its sensitivity and specificity was determined using the SCOP classification (version 1.61) as the gold standard. FAST achieved higher sensitivities than DaliLite, CE, and K2 at all specificity levels. FAST was also tested against 1033 manually curated alignments in the HOMSTRAD database with an overall agreement of 96%.

A bipartite graph matching framework, for protein structure comparison, is introduced in [136]. The framework is capable of producing sequence-dependent and sequence-independent alignments. Each protein structure is represented as one part of the bipartite graph. The nodes of the graph can be alpha-Carbons or residues or SSEs. Geometric features of the node form its feature vector and the weight of the edge is determined by a similarity metric. However, the authors do not provide any results on the performance of the proposed framework.

3.1.2 Dynamic programming based techniques

In [77] the authors present a dynamic programming based, fragment chaining, global alignment algorithm for protein structure comparison called Matt. Locally aligned AFPs are chained together, allowing local transformations to better align the fragments during chaining. The algorithm outperforms traditional multiple structure alignment methods. Matt was tested against MultiProt, Mustang and POSA using the HOMSTRAD and SABmark benchmark datasets. Matt's performance was competitive on HOMSTRAD and it outperformed the other algorithms on the SABmark dataset.

A fast and flexible, Voronoi-contact based structure comparison algorithm, called Vorolign, is developed in [16]. A residue of a protein is represented by its neighbors as defined by Voronoi tessellation. Similarity between two residues is calculated as the similarity of their corresponding nearest-neighbor sets. Dynamic programming is used to calculate the optimal pairwise alignment. Vorolign was found to accurately determine the correct family, super-family or fold of a protein with respect to the SCOP classification. The authors report that a scan against a database of over 4000 proteins took on an average 1 min per target. Vorolign was also compared to CE in calculating pairwise and multiple alignments and found to have a comparable performance.

In [143] the authors propose an algorithm, called TMalign, for identifying the best structural alignment between proteins using TM-score rotation matrix and dynamic programming. Three kinds of initial alignments are used: dynamic programming based SSE alignment, gap less structure matching and dynamic programming alignment using scoring matrices obtained in the previous two alignments. A heuristic iterative algorithm is applied to the initial alignments in order to obtain the final one. In a benchmark test of 200 x 199 non-homologous protein pairs, TMalign was found to be approximately 20 times faster than SAL with more accurate alignments. TMalign was also found to be faster than CE and DALI, while returning structure alignments of higher TM-score.

Fr-TM-Align, another TM-score based structure alignment algorithm which is a successor of TMalign, is presented in [86]. An initial set of equivalent residues is generated using gapless threading and SSE similarity. Dynamic programming is then used to refine the initial alignment, maximizing the TM-score. For the assessment of the structural alignment quality of Fr-TM-Align, in comparison to other programs such as CE and TMalign, scores such as PSI and TM-score were used. The assessment showed that the structural alignment quality of Fr-TM-Align is better than that of CE and TMalign. On average, the structural alignments generated using Fr-TM-Align have a higher TM-score and coverage in comparison to TMalign.

3.1.3 Information Retrieval based techniques

A tableau based approach, called IR Tableau, is presented in [142]. Relative orientation (angle between axis) of the proteins SSEs are encoded, forming the tableau, and stored on a database. A protein is represented by a 32 dimensional feature vector. These feature vectors are used for comparing a query protein structure with the database of protein structures. Feature vector similarity is assessed using: Cosine similarity, Jaccard Index, Tanimoto coefficient or Euclidean distance. Experiments on the ASTRAL SCOP protein structural domain database (a subset of SCOP) demonstrated that IR Tableau achieves two orders of magnitude speedup over the search times of other tableau based methods.

A two-step methodology is proposed in [36] for fast protein structure comparison. In the first step, screening of protein structures is performed. The second step involves global structure alignment of the query structure with the reduced set. The protein structure is represented as a sequence of local structures. The local structures are codified in a geometric invariant manner. Global alignment is performed using dynamic programming. For a typical protein structure, the method is able to reduce the protein data bank to 200 proteins while retaining the structurally closest neighbors, resulting in 30 to 60 fold improvement in the execution time.

3.1.4 Geometric techniques

In [66] the authors develop a mathematical framework for protein structure comparison by treating them as curves in 3D space. The 3D coordinates of the backbone atoms (N, Ca and C) from the PDB file of a proteins structure are used to derive the curve representing a protein in 3D space. Geodisic distance between two such curves can then be calculated. The approach implicitly makes the structural comparison amino-acid sequence dependent. The authors show that the method performs comparably with CE in protein structure classification on a large manually annotated data set taken from SCOP.

In [76] the authors present a sequence-independent method for aligning protein structures using 3D Spherical Polar Fourier (SPF) representation. A protein structure is represented as a 3D density function. In order to compare two structures, one protein is held fixed and a 6D search over the positions of the second protein is performed. The proposed approach was tested by automatic classification of subsets of the CATH database. Results of the clustering showed that SPF is able to classify protein structures with high precision and accuracy with respect to the expert-curated CATH classification.

Two algorithms for protein structure comparison are presented in [68]. Each one of the two algorithms consists of three steps and the algorithms differ only in the third step. The first step involves identifying local alignments, in terms of consecutive alpha-Carbon pairs, by using the distance matrices of the two backbones. In the second stage the local alignment is used to find an initial rigid body transformation using the least square estimation method. In the third stage dynamic programming (sequential) or maximal matching (non-sequential) is applied to generate an alignment. It is argued that the increase in speed for the matching is because the approach does not involve use of alpha-Carbons to find initial rigid-body transformation. The algorithm is available in public domain supported by a cluster of 6 computers. Compared to Dali, CE and SSM [60] the proposed algorithms result in alignments with smaller RMSD and average length of alignment longer than SSM.

3.1.5 Contact map based techniques

In [92] the authors propose a heuristic approach for solving the maximum contact map overlap (MAX-CMO) problem [5] used in protein structure comparison. A variable neighborhood search (VNS) meta-heuristic is applied with multi-start capability to approximately solve the MAX-CMO problem. The approach was tested by auto-classification of domains from SCOP and CATH. The proposed method performed with an error rate of 3.5% on the Lancia dataset and 1.7% on the Skolknick dataset with respect to the optimum value.

In [72] the authors develop a graphlets based CMO heuristic, GR-align and use it for database searches. Experiments with small and very large datasets show that proposed method is several times faster than most of the popularly used PSC methods. The authors also compare GR-align to other flexible alignment methods and show that the method generates better alignments than its counterparts.

3.1.6 Polynomial time approximations

An approximate polynomial-time algorithm for protein structure matching is developed in [57]. Similarity between pairs of proteins is calculated by finding the Euclidean distance between the structures using RMS as the distance metric. A new scoring function, STRUCTAL, is developed and tested on a small set of protein pairs for its viability. The authors present preliminary experiments studying various features and conclude that it is a well behaved scoring function.

In [62] a method for protein structure alignment, ProSup, which maximizes the number of aligned residues is developed. RMSD is used as a constraint in maximizing the number of aligned residues. The authors show that the approach is able to detect remote structural similarity by using 10 protein pairs used in the Shindyalov and Bourne dataset [119]. The results were compared to VAST, DALI and CE. The results show that ProSup performs most consistently in terms of the RMSD of the aligned structures and out performs VAST in all cases on the length of the alignment.

3.1.7 Simulated annealing based techniques

In [25] the authors propose a method for solving protein structure alignment, based on mean field annealing. They reduce the problem to a mixed integer-programming problem with the inter-atomic distances between the structures as the objective function. Speed up is obtained by avoiding combinatorial computation through reducing the problem to a non-linear continuous optimization. The Shindyalov and Bourne dataset [119] was used for benchmarking the algorithm against Dali, CE and Lund. The results show that the algorithm performs comparably to the other methods in terms of RMSD of the aligned structures.

In [138] a method for protein structure comparison is presented, called Topsalign, where Simulated Annealing is used to optimize an initial alignment. Initial multiple alignment is obtained by a pattern matching algorithm that finds equivalent secondary-structures. Topsalign, was compared with STAMP and DALI and found to be comparable on identifying structurally related proteins, as defined in CATH.

3.1.8 Techniques with special alignment scoring

A new protein structure similarity metric ’phenotypic plasticity’ is proposed in [34]. The authors also present a method called phenotypic plasticity method (PPM) for protein structure comparison. Pairs of core blocks in each of the two proteins structures are compared and their RMSD is calculated. A PPM graph is generated from the core blocks of the two proteins. The optimal alignment can be found by identifying the subgraph, of the PPM graph, with the maximal score. PPM performance in detecting similarities between protein structures was benchmarked using SCOP and CATH. PPM outperformed TMalign and Vorolign in the benchmarking results.

In [97] the authors develop a neural network ensemble (NNE) that uses negative correlation learning (NCL) to find structurally conserved residues in proteins. The NCL-NNE method was applied to 6042 structurally conserved residues (SCRs) from 496 protein domains. The method obtained high prediction sensitivity (92.8%) in identification of SCRs. Further benchmarking using 60 protein domains containing 1657 SCRs showed that the NCL-NNE can correctly predict SCRs with approximately 90% sensitivity.

3.1.9 Other methods

In [141] the authors present a fast pairwise local structure alignment algorithm combining both RMS and URMS metrics [26]. URMS is used to find all viable transformations and then dynamic programming is used to optimize the local alignments using RMS. URMS is used to identify all pairs of similar fragments between the two protein structures. The pairs are then clustered based on similar rotations. Each cluster is then searched for the most viable translation among its members. Finally dynamic programming is used to find the best structural alignment under the rotation-translation transformations identified, using an RMS-based scoring matrix. The method was found to be faster and more accurate than CE in identifying structures belonging to the same family, using the SCOP dataset.

An approach based on principal component analysis of structural data is presented in [146]. Symmetric interaction matrices are constructed from secondary elements for the structures being compared and principal component correlation is applied to compare the structures. 56 protein structures, grouped into 6 different sets according to CATH, were used for testing the proposed approach. The results show the approach assigns strong correlation to structurally related protein pairs.

In [95] the authors rigorously develop a series of algorithms for optimizing protein structure similarity measures, including those commonly used in protein structure prediction. The authors also present an algorithm for near-optimal structure alignment. A fast heuristic implementation of the algorithm proposed, called MAX-PAIRS, was developed for the purpose of providing test results. Testing was performed on a set of 195 pairs of proteins selected from the SCOP database. Performance of the algorithm was compared with CASP LGA, TMalign, MAMMOTH and MUSTANG. The results showed that MAX-PAIRS compares favorably with other algorithms in terms of RMSD of the aligned structures.

CURVE, an algorithm based on representation of the protein as a path in 3D space can be found in [145]. In CURVE, a ”smoothed” representation of a protein structure is used, retaining only information about the shape of the backbone (straight or curved). The structure is thus represented as a series of turning angles. CURVE was compared against CTSS and CE using a set of 100 randomly selected structures. Performance of CURVE was found to be comparable to that of CTSS, while CE was the most superior algorithm of the three.

3.2 Distributed and parallel protein structure comparison

3.2.1 Grid and Cluster based approaches

The predominant approach is to make Grid computing available for Bioinformatics researchers. These computing resources provide an on-line environment for performing protein structure matching. They are supported by large Grid setups that share popularly used structural comparison algorithms. The working environment, provides the user with an interface where the datasets can be uploaded and the user can select which ’services’ to run for the structure comparison. ProCKSI [13] enables the user to select among different similarity measures to be used while performing the structural comparison. Typically, these setups use ontologies for defining the work-flows. The execution of these work-flows uses different structure comparison software in a series, to complete the task. These applications may reside on different nodes participating in the Grid and are automatically used without manual specification.

FROG [88], which uses a Genetic Algorithm, and PROuST [31], which combines indexing and dynamic programming, are two examples of algorithms, for protein structure comparison, designed from scratch keeping in mind parallelism. These algorithms are designed to easily utilize a distributed environment if it is available. Both approaches provide significant speed ups when they can take advantage of their parallel processing capabilities. Parallelized FROG was implemented in C on a 16 node Linux cluster running dual Pentium3 1GHz processors. A comparison of FROG with sequence-based alignment methods, using a subset of the SCOP database, shows it is able to infer phylogenies accurately. In PROuST, geometric features of triplets of secondary structures of proteins are extracted and hashed for fast retrieval. Similar features, extracted from the query protein, are used to identify a subset of the protein structures stored on the database for which dynamic programming is used to generate an alignment. A comparison of PROuST with leading PSC methods showed that it outperforms them in the time taken to generate alignments with comparable RMSD.

In [113] the authors propose a framework for the design of an algorithm that runs on computational nodes, organized as a cluster/grid for protein structure comparison. A high-throughput implementation of a system that allows parallel processing of very large protein structure datasets is developed. Building on ProCKSI the implementation in [14], performs structure comparison significantly faster than the current ProCKSI implementation, while retaining similar accuracy. Tests were preformed on a Linux cluster of 64 nodes containing dual Itanium2, 1.4GHz processors with 4GB main memory, using the RS119 and CK34 datasets, as well as the dataset proposed in [53]. Results of testing on the first two datasets showed that the distributed algorithm performs 30 times (with RS119 dataset) and 26 times (with CK34 dataset) faster as compared to its sequential counterpart.

CEPAR [91] is an extension of the CE algorithm [118] that involves massive parallelism in order to fully take advantage of parallel processing capabilities. The CEPAR algorithm uses a coarse-grain parallel implementation (master/worker strategy) of the CE algorithm. Each worker receives a smaller protein comparison problem from the master which it solves by using the CE algorithm. Further, the algorithm was tailored for the IBM SP parallel computer with a total of 1152 Power3 processors, each running at 375 MHz. Experiments performed, with a C++ implementation of the algorithm, showed that the algorithm does not scale gracefully as the number of processors increases, loosing efficiency as the number of processors increases beyond 500.

SBLAST [79] is an attempt to incorporate parallel access to databases, for speeding up protein structure comparison. Triplets of alpha-Carbons, selected from all proteins in a structure database, are stored on a hash table. Hash table hits, from similar triplets extracted from the query protein, are recursively extended to find the largest common substructure. A ”Master-Worker” paradigm is implemented to parallelize the work done during the pre-processing and searching phases of structural comparison. The pre-processing phase involves building a database of known protein structures, against which a query protein is matched for finding similar structures. The master node controls the division of the jobs among the slaves and collates the comparison results from the different nodes as they come in. The final scoring is done by the master node. This configuration makes use of multiple resources, therefore reducing the load on any individual node. The algorithm was tested using the ASTRAL subset of SCOP. The pre-processing and search modules showed an effective speed up of 70% and 60% for 1000 and 500 processors respectively, as compared to a single processor implementation. No comparison of the result with other existing structure comparison techniques was presented.

3.2.2 GPU based approaches

In [124] a tableau-based heuristic protein structure matching algorithm is presented which uses simulated annealing. The authors develop a fast parallel implementation using the NVIDIA CUDA programming model. The implementation makes use of global, shared and constant memories, to allow fast data access for the multiple threads running in parallel. The authors report a 34 fold speed up, while retaining similar accuracy when compared to the VNS maximum contact map overlap method [92].

In [87] the authors develop a framework for optimal exploitation of GPUs for PSC methods. They evaluate implementations for TMalign, Fr-TM-align and Mammoth on an NVIDIA Tesla C2050 GPU. The framework makes use of a GPU-CPU co-design with the CPU identifying paired fragment pairs and the GPU performing the structural alignment step. The authors report a 36 fold speed up for ported TMalign and a 40 fold speed up for Mammoth while the ported Fr-TM-align achieves a speed up of 65 over the a serial implementations of these PSC methods running on a 1.8 GHz dual-core AMD Opteron CPU.

In [81] the authors report a GPU-based implementation of the CASSERT algorithm. They exploit the parallelism inherent in the two phases of the basic algorithm a) SSE alignment and b) spatial structure alignment. The authors compare the performance of the GPU implementation running on a GeForce GTX 560Ti (384 cores, 2GB RAM) with that of the serial implementation of CASSERT running on a Intel Xeon E5620 2.4 GHz quad-core CPU and report an average speedup of 180-fold.

3.3 Parallel architectures

With the advancement in processor technologies several computer architectures have emerged for use in high throughput computing. Most popular of these are: Field Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs), multi-core CPUs and more recently many-core CPUs [125]. In this section we briefly describe these technologies and compare and contrast them.

FPGAs are made of configurable logic components with programmable interconnections. The logic components can be programmed to behave as the basic logic functions -AND, OR, NOT and XOR- as well as emulate circuits of any complexity. By using this ability, together with the programmable interconnections, very complex circuits can be implemented on an FPGA. Typical applications of FPGAs are signal and image processing, cryptography, specialized routers etc. Application of FPGA based solutions to Biocomputing are also becoming popular, especially for sequence comparison and structure prediction [106].

A relatively new technology, growing in usage in the scientific community, is the Graphics Processing Unit (GPU). GPUs are multi-core processors capable of supporting highly parallel applications, originally designed to accelerate computer graphics. There is a growing interest in tapping into the processing capabilities of a GPU for general-purpose computation [89]. Programming API's, such as VDPAU etc., have become available for utilizing the computing power of GPUs, in a plethora of common programming languages. Typically GPUs find usage in applications that: have large computational requirements, use algorithms that exhibit high parallelism and applications that favor high throughput over low latency [84].

Multi-core processors are ubiquitously available and have been around for a while but there is a lack of good threaded software especially in the proteomics domain. On the other hand, while very similar to multi-core processors, many-core processors are not very commonly available but are likely to grow in availability over time. These two architecture share several characteristics however they also possess some fundamental differences. Multi-core processors typically contain two or more independent processing units integrated in a single chip and typically scale up to 16 cores. On the other hand, Many-core processors contain a larger number of independent processing units (more than 16) organized in a grid. These architectures, also have an impact on software targeted at leveraging distributed architectures such as clusters and grids [22]. With multiple cores appearing on a single chip inter-core communication becomes important. Two broad strategies through which inter-core communication is performed are: a) using a single communication bus and b) using and interconnection network. Each strategy has its advantages and disadvantages and comes with an associated programming model, as shown in Figure 3.1. Single bus communication typically leads to diminished performance when the number of cores exceeds 32 [109].

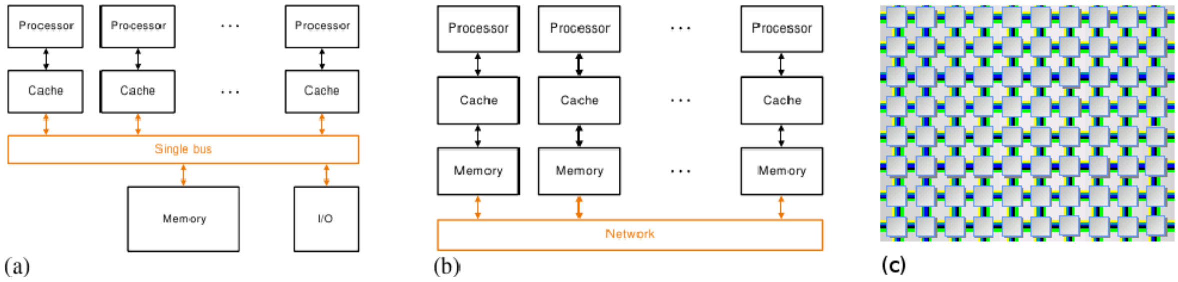


Figure 3.1: Strategies for implementing inter-core communication in a multi-core and many-core architecture. (a) Using a single communication bus, resulting in a Shared Memory Model (b) Using an interconnection network, resulting in a Distributed Memory Model (taken from [109]), and (c) Mesh network interconnection between tiles that carry one or more computing cores and local caches. Such Networks on Chip (NoC) are the modern trend in many-core processors (taken from [42]).

In [23, 43] the authors compare three architectures: FPGAs, Multi-core CPUs and GPUs, within the per-view of the implementation of a sorting application. The authors conclude that programming an FPGA requires more command over digital electronics than computer science. Further, synthesis of the hardware from its description is very time consuming and the tools used for hardware synthesis are unlikely to get faster, in the opinion of the authors. For Multi-core CPUs the authors state that the programming model is close to C/C++ programming, therefore easy to grasp, though experience in parallel programming is helpful. The authors concluded that for GPUs the programming frameworks are unstable and vendor specific. Further, while NVIDIA's CUDA is well developed it is hard to maintain the code produced due to the lack of debugging support. Transfer of data between CPU and GPU memory is cited as a bottle neck, though the authors state there are some workarounds. Further, the authors conclude that the flexibility of FPGAs allows development of more efficient algorithms, while the implementation cost in terms of time and skill, is lower for programming the GPU. Both works conclude that the ease of programming multi-core CPUs makes them an excellent target for parallel program development in general.

By comparison with the other parallel architectures discussed above the many-core processors have several features that make them attractive for parallel MCPSC. They provide a) large number of Processing Elements (PEs), b) the flexibility to combine these elements into banks of serial and parallel PEs, c) high speed interconnection between the PEs and d) a familiar programming paradigm. These advantages of MCPSC apply in all cases except perhaps in comparison with multi-core CPUs. However, due to the technical bottleneck of the bus based architecture the number of PEs on many-core processors will soon outnumber those on multi-core processors by very large factors, giving the former a clear edge over the latter.

3.4 Reviewing the open problems

From the survey of the field of protein structure comparison it is clear that there are three orthogonal dimensions that contribute to the complexity of the methods - size of structure databases, need to support multiple comparison algorithms and complexity of the pair-wise comparison algorithms used. Efforts have been made to harness the computing capabilities of distributed computing systems (clusters and grids) for the task but this tackles only the first two of the three dimensions mentioned above. Due to the diversity of techniques that are used in the field the level of granularity at which parallel implementations can be introduced is specific to the technique. Therefore independent analysis of each technique is required to assess the best way to parallelize it. While at the same time it is crucial to identify common reusable components that can benefit multiple PSC methods as well as simplify targeting MCPSC.

Efficient solutions for MCPSC must focus on using modern processor architectures. The architecture of the platform used for introducing parallelism must provide flexibility in usage of computational elements. Need for flexibility at the platform level is required to find the optimal way of using computational elements to cater both to the algorithmic parallelism as well as multi-method processing. This need for flexibility extends to the software developed in order to allow need based integration of processing elements. The solution developed must allow simultaneous instances of these methods to be available for use. These factors make FPGA, many- and multi-core processors interesting target platforms.

Further, solutions developed must be able to scale to very-large datasets to be of practical use in real world scenarios where researchers are interested in comparing and classifying thousands of protein structures. Of the parallel architectures discussed most are suitable for use with this constraint, given an appropriate implementation can be developed. Such implementations require identification of established, well respected and portable PSC methods. It is readily evident that CPU based solutions are more likely to be suitable for this purpose because current software already makes use of them even if not very efficiently. Finally, in order to aid researchers in leveraging PSC results the solutions developed must be easily accessibly, deployable and usable. These constraints imply that solutions developed should target commodity hardware rather than rely on some esoteric hardware that is unlikely to be available to the community at large. Further, the solutions developed should be made available to the community via mechanisms that make them accessible and usable for the largest possible segment.

With these considerations as guiding principels, in Chapters 5 and 8 of this thesis we present solutions for speeding up MCPSC that target multi- and many-core CPUs. Detailed performance comparison and benchmarking of the solutions developed is presented in Chapter 6 and 7. We also present analysis of development of FPGA based hardware-software codesign for accelerating specific FPGA methods with method specific implementations. We find however that the latter is not a viable solution due to the nature of PSC as discussed in Chapter 4.

4. MODELING PARALLEL MULTI-CRITERIA PROTEIN STRUCTURE COMPARISON

In this chapter we take a close look at the algorithmic model of pairwise Protein Structure Comparison. We identify and discuss levels of parallelism as applicable to PSC (one-to-many etc.) and study the characteristics of pairwise PSC, which will form the basis if load balancing strategies compared in Chapter 6.

In the latter half of this chapter we discuss three specific PSC methods and highlight scope for parallelism in these methods. In Section 4.4, we take a look at fine-grained parallelism for PSC using FPGAs. We analyze the short comings of the approach and establish why it is unlikely to yield good results for PSC in general. Finally, in the last section we present the model for MCPSC which is used for developing solutions presented later in this thesis.

4.1 An algorithmic view

Algorithm 1, shows a high level view of the comparison of the structure of a protein with a database of known structures. A query protein structure q is compared to each protein structure p, in the database D. For each pair of protein structures the best alignment of the structures must be identified. Once all structure pairs have been processed a ranked list $alignments$, of protein structures is generated.

Four functions appear in Algorithm 1: $fragment$, $align$, $bestSubset$ and $sort$. The function $fragment$, partitions a protein into its fragments. The fragments generated could vary from backbone residues to blocks of residues in the protein, such as SSEs, triplets etc., depending on the algorithm. The function $sort$, accepts a list of protein structures with the similarity scores and sorts them from most similar to least similar.

The key functions in the algorithm are: $align$ and $bestSubset$. The function $align$, finds the optimal alignment of a pair of fragments, one from each protein structure. Similarity metrics, such as RMSD, cRMSD etc., are used for determining the optimal alignment. The task of finding all alignments has a complexity proportional to NxM, where N is the number of fragments of p and M is the number of fragments of q. However, finding the optimal alignment of a fragment pair is computationally intensive, because the atoms making up the residues can take an infinite number of orientations in 3D space. The function $bestSubset$, finds the best set of fragment alignments and determines the global alignment of the two protein structures. Definition of the best alignment depends on the algorithm and is determined by metrics such as: RMSD, TM-Score etc. For sequential structure alignment techniques, feasible alignments are limited by the order in which residues appear in the protein structure and the cost of introducing gaps in the alignment.

```
Data: q: structure of query protein, D: database of known protein structures
Result: alignments: list of protein structures ranked by similarity to q
alignments := [];
fq := fragment(q);
for p in D do
    fp := fragment(p);
    paligns := [];
    for fpa in fp do
        for fqa in fq do
            add(paligns, align(fpa, fqa));
        end
    end
    add(alignments, bestSubset(paligns));
end
sort(alignments);
```

Algorithm 1: A general algorithm for protein structure comparison

The loop starting at line 3 in Algorithm 1 explains the growing need for computing power in protein structure comparison as the size of the database grows. This loop also explains the success of distributed processing approaches as each iteration is virtually independent of the next. The nested loops at line 6 and 7 indicate that any increase in speed of generating optimal pairwise fragment alignment is likely to have a significant impact on the overall time requirements. Since the $align$ function executed in each iteration of the nested loop scores the pairwise alignments using a metric, as explained above, faster calculation of metrics should improve the performance times. A third significant aspect is the alignment generated by the $bestSubset$ function. Several algorithms, such as dynamic programming, bipartite graph matching etc., are used to perform the step. Speeding up the alignment process should also have a significant impact on the overall query processing times.

It must be highlighted that Algorithm 1, presents a very simplistic template of algorithms for protein structure comparison. While key operations represented by the four functions must be performed, existing methods for protein structure comparison may not perform them in the order shown in the algorithm or may avoid performing them online. For example, fragments of known structures can be extracted in an offline step and proteins can be indexed by them, thus avoiding time spent on extracting these during query processing. Similarly performing pairwise fragment alignment, between protein pairs, may be avoided by representing protein structures as Contact Maps.

4.1.1 Scope for parallelism in protein structure comparison

Methods used for parallelizing a problem can be classified by the level at which the parallelism is introduced. Based on the discussion of Algorithm 1, presented above, there are the following categories of methods that can be used for introducing parallelism in protein structure comparison:

Sharding

Methods belonging to this category are similar to Database Sharding, a technique that is popularly used in Information Retrieval where large databases are involved. Given the computational, resources it is possible to split the database into several smaller databases (shards), which taken together contain the complete data. The splitting is performed such that the index is distributed across the shards. Suppose N computational nodes are available, the database can then be split into N shards. The overall query time, since each of these node can work independently, is reduced to T/N, where T is the original time to compare the query to the entire database. Clearly this form of parallel processing is tailor made for the Master-Slave computational model. Some evidence can be found in literature of the use of this technique to speedup protein structure comparison [110].

Algorithm specific parallelism

A variety of methods have been used for protein structure comparison. Each class of methods essentially shares a similar algorithmic approach. A more fine grained level of parallelism can be achieved by analyzing these algorithms from first principles. It is easy to see that methods belonging to this category are likely to be algorithm specific.

Major classes of algorithms, such as: dynamic programming, maximum clique finding etc., can each have a parallel implementation, which is likely to be specific to the algorithm class. Attempts have been made to develop a parallel implementation of the Geometric Hashing algorithm, as stated in this thesis. However, more work is required in this direction to develop viable parallel implementations for the other major classes of algorithms. Published literature can be found on independent parallel implementations of some of these algorithm classes . While these parallel implementation have not been utilized in structure comparison, it should be possible to do so.

Parallelizing similarity comparison

The finest level of parallelism can be achieved by parallelizing the similarity metric calculation task of protein comparison algorithms. Several different metrics are used in protein structure comparison. The similarity is calculated between 'units' of each protein (the units can be alpha-Carbons, SSEs, AFPs etc). By developing parallel techniques for calculating these similarity metrics a great amount of time can be saved in the pairwise comparison of protein structures.

4.2 Analysis of PSC methods

4.2.1 TMAlign

Summary

In [143] the authors propose an algorithm, called TMalign, for identifying the best structural alignment between proteins using TM-score rotation matrix and dynamic programming. Three kinds of initial alignments are used: dynamic programming based SSE alignment, gap less structure matching and dynamic programming alignment using scoring matrices obtained in the previous two alignments. A heuristic iterative algorithm is applied to the initial alignments in order to obtain the final one. In the original publication an average alignment time of 0.5 seconds using an Intel Pentium III (1.26 Ghz) was reported.

Implementation Analysis

The TMalign algorithm consists of programs written in C++. An analysis of the program, generated using *SLOCCount* software, revealed that it consists of 2,554 lines of code. The sources are organized as a set of four files, with one main program and no third party dependencies (apart from the standard C library). Further, the makefile supplied in the download does not suggest any specific compiler flag requirements except for the 'fast-math' flag. The call graph for the main program in the implementation of the method is shown in Figure 4.1. Generated using the *Doxygen* the figure shows that the implementation of TMalign has very few dependencies. The program is organized in a set of 4 self-sufficient files with dependencies only the standard C library.

Exploitable Aspects

Based on a preliminary analysis of the algorithm and its implementation, the tasks of a) generating fragment pairs for the two protein structures and b) calculating the similarity matrix for the fragment pairs could be performed in parallel. This would involve distributing the protein into multiple nodes, and calculating the pair-wise similarity of different fragments simultaneously. Subsequently, the similarity scores can be collected to create the similarity matrix which is used in generating the final alignment. The work described can be performed using the standard Map-and-Reduce tasks used in MPI programming.

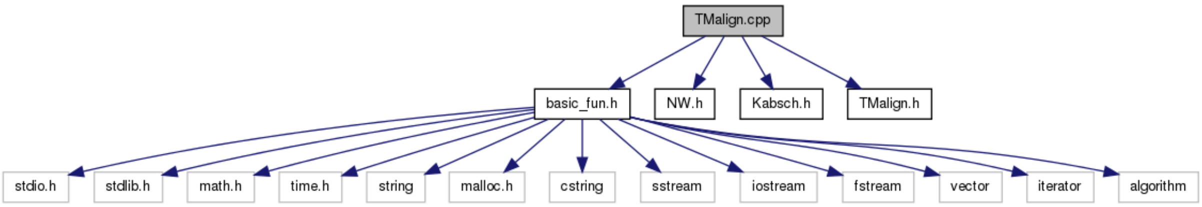


Figure 4.1: TMalign - main call graph

4.2.2 Combinatorial Extension

Summary

In the Combinatorial Extension (CE) algorithm [119] structural alignment of two proteins A and B, of lengths N_a and N_b respectively, is the longest continuous path P of Aligned Fragment Pairs (AFPs) of size m in a similarity matrix S, of size $(N_a - m) * (N_b - m)$, representing pairwise structural similarity score of all AFPs. The algorithm allows generation of gapped sequential alignments of protein structures. Using the similarity matrix S, empirically determined value of m (default value is 8) and distance based heuristic similarity evaluation and path extension formulas, pairwise structural alignments of proteins can be performed. In the original publication an average alignment time of 20 seconds, using a Sun Microsystems Inc. Ultra Sparc II processor (248 Mhz), was reported.

Implementation Analysis

The CE algorithm consists of programs written in C++. An analysis of the program, generated using *SLOCCount*, revealed that it consists of 13,856 lines of code. The sources are organized as the main program and a library on which it depends with no third party dependencies (apart from the standard C library). Further, the makefile supplied in the download does not suggest any specific compiler optimization flag requirement. The call graph for the main program in the implementation of the method is presented in Figure 4.2. Generated using Doxygen, the figure highlights the dependency of the main program on a library which is also included in the download package. The core functions in the main program as well as the library depend only on the standard C libraries including the math library.

Exploitable Aspects

Based on a preliminary analysis of the algorithm and its implementation, the dynamic programming (DP) steps can be performed in a distributed manner. This would involve distributing the similarity matrix over the multiple compute nodes, and performing the DP in a parallel instead of the serial implementation that is currently used. Since DP is performed twice during the program execution, for every pairwise structure comparison, a significant speed-up could be achieved by introducing parallelism.

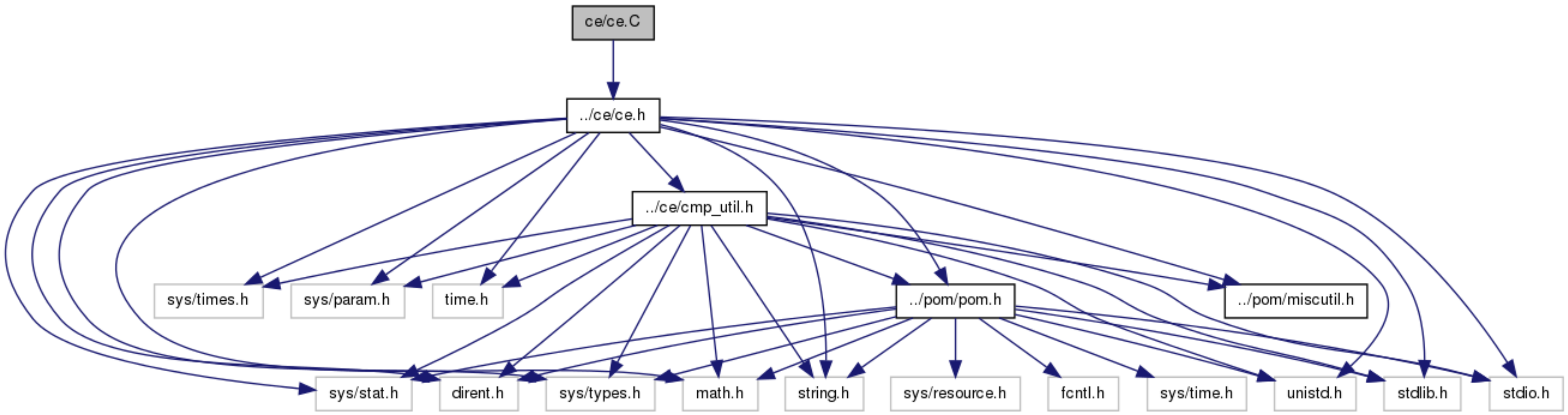


Figure 4.2: Combinatorial Extension - main call graph

4.2.3 Universal Similarity Metric

Summary

In [59] a method inspired by information theory, Universal Similarity Metric (USM), is applied to assess similarity between a pair of protein structures. USM relies on the Kolmogorov complexity [55], $K(.)$ of an object o, defined by the length of the shortest program of a Universal Turing Machine U needed to output o, described in Equation (4.1). Two related formulas pertinent to the derivation of the USM are provided in Equation (4.2) and Equation (4.3). Equation (4.2) shows the calculation of the conditional Kolmogorov complexity of an object o_1 given object o_2, it is a measure of the information required to produce object 1 if object 2 is given i.e. transforming one protein structure into another. The information distance between the two objects then is defined by Equation (4.3).

$$K(o) = min\{|P|, P\, a\, program\, and\, U(P) = o\} \tag{4.1}$$

$$K(o_1|o_2) = min\{|P|, P\, a\, program\, and\, U(P, o_2) = o_1\} \tag{4.2}$$

$$ID(o_1, o_2) = max\{K(o_1|o_2), K(o_2|o_1)\} \tag{4.3}$$

The Universal Similarity Metric for a pair of objects, protein structures in this case, is then defined by Equation (4.4) below,

$$d(o_1, o_2) = \frac{max\{K(o_1|o_2^*), K(o_2|o_1^*)\}}{max\{K(o_1), K(o_2)\}} \tag{4.4}$$

where $o_1^*(o_2^*)$ indicates the shortest program for o_1 (or o_2). The protein structures of each of the two proteins are converted to contact maps. Using the contact map representation of the protein structures along with Equation (4.4) a similarity value for the two structures is produced.

Implementation Analysis

The USM algorithm consists of programs written in Java. An analysis of the program, generated using *SLOCCount*, revealed that it consists of 686 lines of code. The sources are organized as a set of three files.

Exploitable Aspects

Based on a preliminary analysis of the algorithm, the comparison of the contact maps representing the protein structures can be performed in a distributed manner. The calculation of Equation (4.4), which yields the similarity value for the given structures, is performed over two square matrices, each one representing one contact map. The comparison at each individual cell is independent of the others and the result is later combined to produce a similarity value and alignment. The task could be distributed among multiple cores in order to be performed quicker.

4.3 Characteristics of pairwise PSC

We studied the variation in the pairwise PSC processing times with respect to the sum of the lengths and product of the lengths of the pair proteins. The analysis was carried out using processing times recorded on the PC. Figure 4.3, shows the processing times for the pairwise PSC tasks as the normalized sum of lengths and the normalized sum of product of lengths of pair proteins are varied from 0 to 1. The USM PSC method was excluded from this analysis due to the extremely small pairwise processing times. The dotted line in the figures shows the best fit quadratic curve for the observed data points. The figure clearly highlights the processing time requirement differences between CE and TMalign.

A study of the variation of pairwise PSC times with respect to combination of lengths of pair proteins reveals an interesting trend. The time required for completing a pairwise PSC task is clearly a function of the lengths of both proteins forming the pair as can be seen in Figure 4.3. The best fit achieved with the product of lengths of the pair proteins is better than that achieved with the sum of lengths. The relative speed for completing a pairwise PSC task, given a list of pairwise PSC tasks, depends on the properties of the pairs participating rather than the specific PSC method being used. This is also evident from the similarity in slowest pairs for different algorithms as shown in Table 4.1. However, the absolute time required for completing a pairwise PSC task depends on the complexity of the method used. Since the complexity of one PSC method as compared to another cannot be known a-priori, it cannot be used as a factor in the partitioning scheme. Based on the above analysis product of lengths and sum of lengths of the pair of proteins were used individually as parameters for developing load balancing schemes for the ported algorithms. Further the need for balancing across multiple algorithms, in the MCPSC setup, is met by jobs of each algorithm individually across all processing elements.

Table 4.1 lists the 10 slowest pairwise PSC tasks per algorithm and it can be seen that the lengths of the pairs in each case is very high. We performed a statistical analysis using SecStAnT [70] of the proteins participating in the 10 slowest pairwise PSC tasks and the 10 fastest. The parameters on which SecStAnT compares proteins are secondary structure characteristics - Bond Angle, Dihedral Angle, Theta and Psi. Results of the 1-parameter and 2-parameter statistical analysis, distributions and correlations, performed

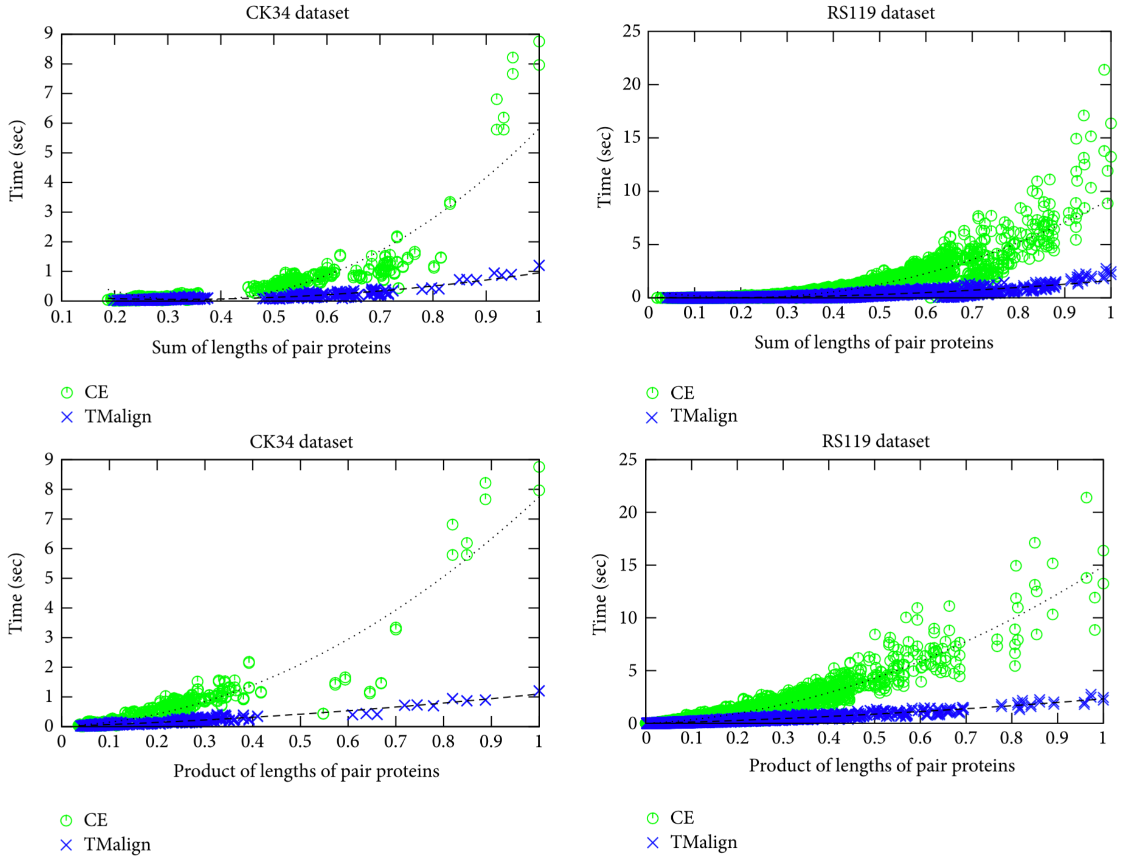


Figure 4.3: Comparison of pairwise PSC processing times using TMalign and CE, with respect to the normalized sum of lengths and normalized product of lengths of pair proteins. The dotted lines show the quadratic best-fit for the data.

Table 4.1: The table lists pairs of proteins (prot) and their lengths (len). The protein pairs included in the table are those that perform slowest in the PSC experiments with both TMalign and CE algorithms. The pairs that are slowest for both algorithms from the two datasets are consistent indicating that the relative speed of pairwise PSC, in a given dataset, depends largely on the pair being compared rather than the method used.

TMalign CK34				**TMalign RS119**			
prot 1	**prot 2**	**len 1**	**len 2**	**prot 1**	**prot 2**	**len 1**	**len 2**
1ct9	4enl	553	436	6acn	6acn	754	754
4enl	1ct9	436	553	6acn	2gls	754	469
2mnr	1ct9	357	553	2gls	6acn	469	754
1ct9	2mnr	553	357	6acn	1lap	754	487
6xia	1ct9	387	553	1lap	6acn	487	754
1chr	1ct9	370	553	7icd	6acn	416	754
1ct9	6xia	553	387	6acn	7icd	754	416
1ct9	1chr	553	370	2aat	6acn	396	754
1ct9	1ct9	553	553	6acn	2aat	754	396
2mnr	4enl	357	436	6cts	6acn	433	754
CE CK34				**CE RS119**			
prot 1	**prot 2**	**len 1**	**len 2**	**prot 1**	**prot 2**	**len 1**	**len 2**
1ct9	6xia	553	387	6acn	2gls	754	469
1ct9	4enl	553	436	6acn	1lap	754	487
4enl	1ct9	436	553	6acn	6cpp	754	414
6xia	1ct9	387	553	6acn	2phh	754	394
1ct9	1chr	553	370	2gls	6acn	469	754
1ct9	2mnr	553	357	6acn	6cts	754	433
2mnr	1ct9	357	553	1lap	6acn	487	754
1chr	1ct9	370	553	6cpp	6acn	414	754
1ct9	1ct9	553	553	6acn	4cms	754	323
4enl	6xia	436	387	2phh	6acn	394	754

did not show any significant statistical difference between the two datasets (slowest PSC domains and fastest PSC domains). However, it was observed that the slowest pairs contained the Hlx310Alpha super secondary structure while this was absent from the fastest pairs. The 310-helix is known to occur rarely in protein domains, because the tight packing of the backbone atoms makes it energetically unstable. This structural characteristic might explain the increased structure alignment times for some domains. Further, comparing the proteins using there UniPort entries we observed that most of the domains in the slowest set are Cytoplasmic while the fastest domains are Membrane proteins.

4.4 Fine-grained parallelism for MCPSC

In this section we present fine-grained parallelism solutions for specific PSC methods. We use the proposed solutions to carry out an analysis of the gain from implementing such solutions with an FPGA.

4.4.1 Parallel TMalign

QCPSuperimposer

During the course of our analysis of TMalign, the residue superimposition was identified as the key parallelizable section, therefore, we looked at fast algorithms for performing this operation. Current implementation of TMalign makes use of Singular Value Decomposition (SVD) to find the best rotation and translation for put two point sets on top of each other i.e. minimize the Root Mean Square Distance (RMSD) between the point sets. The superposition of two sets of 3-dimensional points is equivalent to finding the orthogonal rotation and translation that minimizes the squared Euclidean distance between the rows of two matrices corresponding to the two structures being compared.

The Quaternion Characteristic Polynomial (QCP) based method is the fastest method known for determining the minimum RMSD between two structures and for determining the optimal least-squares rotation matrix [129]. In the QCP method, the RMSD is first evaluated by solving for the most positive eigenvalue of the 4×4 key matrix using a Newton-Raphson algorithm that quickly finds the largest root (eigenvalue) from the characteristic polynomial. The minimum RMSD is then calculated from the largest eigenvalue. The best rotation is given by the corresponding eigenvector, which is calculated from a row of the adjoint matrix. The method has several advantages: a) the time required to calculate the rotation matrix is independent of the system size after a special 3×3 matrix is constructed from the coordinates, b) no special cases need to be handled separately, and c) the approach is extremely fast, straightforward, and robust, since there is no expensive matrix inversion or decomposition.

We therefore used QCP as the basis of our hardware design for the superimposer in the TMalign SoC. Prior to its implementation we developed a Python-C implementation to

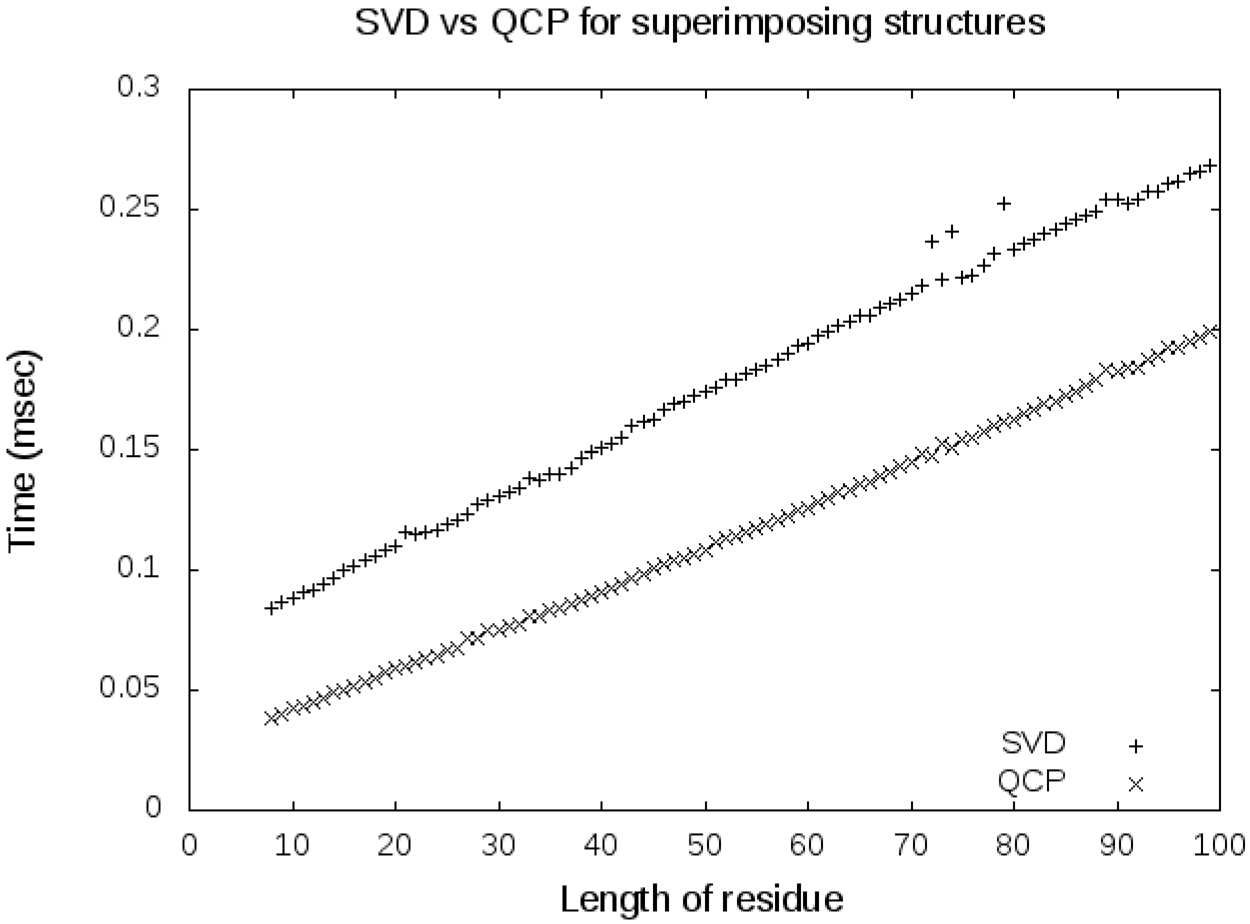


Figure 4.4: Comparative performance of QCPSuperimposer to the existing SVD based BioPython structure superimposer.

study its characteristics with respect to the more standard SVD based implementations. In our implementation, accepted for release in BioPython v 1.66, the most computationally intensive step is implemented in C language. The QCPSuperimposer, that we developed, has three major steps a) centering coordinates of structures being compared, b) calculating the inner product of the centered matrices, and c) finding optimal rotation and translation. We implemented Steps a and b via the fast NumPy Python library and Step c was implemented in pure C with bindings to make it callable from Python.

We compared the performance of our implementation with that of the existing implementation in BioPython. Using randomly selected pair of domains from the Chew-Kedem (CK34) dataset the time for superimposing different length sub-structures was recorded. The length of sub-structures was varied from 10 to 100. Figure 4.4, shows the comparative performance of the two implementations. As can be seen QCPSuperimposer is consistently faster than the existing implementation by a factor of 2.

TMASoC

We developed a software-hardware co-design for parallelism TMalign with the help of an FPGA. Our analysis was based on profiling results obtained for pairwise PSC using TMalign carried out on a PC. Our analysis showed that majority of the processing time in pairwise PSC using TMalign, is spent in superimposing sets of residues from the structures of the pair of proteins. Implementation of a fast routine (QCProt) that performs this function was undertaken and found to be very resource intensive, even for large FPGAs such as the Kintex-7. However, assuming the availability of sufficient resources, such an implementation would still get limited by Amdahl's law, to a speedup of 1.31 since the parallel part of the algorithm forms 24% of the overall. The typical approach to overcoming this situation is to identify other parts that may be parallelizable, however our analysis did not show any other parts of TMalign to be amenable to parallel implementation.

Parallel USM

We also studied the USM PSC method for speedup using fine-grained parallelism. Our, analysis of the method revealed that it is composed of three major parts a) self-residue distance calculation for each protein, b) extraction of contacts for each protein and c) calculation of similarity metric using compression. The first part can be calculated by $N \times N$ processing elements (PE) in unit time given each PE has coordinates of two alpha-carbons of the protein structure. The output from each of these PEs, distance between the alpha-carbons, can then be used to generate a list of 0 or 1 value corresponding to each PE, with a 1 indicating a output higher than some threshold. These two steps can happen in 1 cycle each, given sufficient parallel hardware. Subsequently, four compression values need to be generated to calculate the similarity score. This is the most complex part of the system and requires implementation of a compression algorithm in hardware. Using any such implementation would allow the USM method to be fully implemented in hardware. However, we found that even a very efficient implementation (assuming infinite resources to allow maximum parallelization) provides unreasonably low speedup.

Let us assume a data transfer rate of 40 MB/sec between the PC and the FPGA and a compression rate of the LZA implementation 198.4 MB/sec. Further, we assume that the average length of a protein is 400 alpha-carbons, which gives a total transferable data per protein of 5 KB, corresponding to a data transfer time of 0.25 msec for two protein structures between the PC and the FPGA. The size of contacts data to compress, absence of contact represented by 0 and presence by 1 (step 1 and 2), will correspond to 2 KB per protein giving a total time of 0.06 msec for 4 compressions (one for each protein and one for each ordered combination). Thus the minimum time, assuming fully parallelized parts 1 and 2, for an average pairwise PSC using USM with the hardware implementation will be 0.3 msec. A modern PC running at 3 GHz requires an average pairwise PSC time for USM of 0.53 msec, which results in our optimal FPGA implementation returning a speedup of 1.8.

While parallel implementation for TMalign suffers from a very small parallel portion, the USM implementation suffers from cost of data transfer. Typical, PSC methods make use of heuristic iterations with optimal rotation calculation as in TMalign or contact map generation followed by some method for their comparison. Our parallel implementations therefore represent common shortcomings likely to be encountered in introducing fine-grained parallelism for general PSC methods, rendering it a futile exercise.

4.5 A theoretical model for MCPSC

In [113], the authors developed a framework for high-throughput MCPSC using a cluster of computers. In their work they investigated strategies for distributing PSC jobs over multiple PEs. Given a set of PEs P_1, P_2, ..., P_n (assumed to be homogeneous for simplicity) and the number of pairwise PSC tasks being $Q \times D \times M$, where Q = number of query protein structures, D = number of database protein structures and M = number of PSC methods to be computed, some of the possible partitioning schemes are:

1. Each pairwise PSC per method is treated as a separate job. This results in $Q \times D \times M$ jobs.
2. Each pairwise PSC is treated as a job. This results in $Q \times D$ jobs.
3. All pairwise PSC per method are treaded as a job. This results in M jobs.
4. Comparison of a subset of pairs of proteins with a set/subset of methods to create a balanced number of jobs.

Options 1 and 2 are discarded as being too fine-grained and option 3 as being too coarse-grained [113]. They investigated option 4, developing strategies for optimal performance. The solution developed delivers a high speedup at a high efficiency on a cluster of 64 computers. A key factor in determining which strategy to use is the interprocess communication cost which is high in the case of a standard network of computers. This problem is highly suited to parallelization using NoC based many-core processors, discussed in detail in Chapter 5, due to the low cost of inter core communication and the large number of processing elements.

The low cost of inter-core communication in both multi- and many-core CPUs and the low cost of development (programming effort) for these architectures, makes them ideal candidates for developing accessible solutions for fast MCPSC. On such architectures the most fine-grained job distribution strategy is perhaps ideal to capitalize on their advantages. The focus of the solution developed, therefore, shifts to effective utilization of the processing elements by carrying out load balancing and avoiding costly memory copy operations as much as possible. At the hardware level these architectures provide the flexibility of combining processing elements as needed, which is a much desired feature as discussed in Chapter 3. Solutions developed for these architectures must therefore enable this flexibility to percolate to the software as well. The applications and libraries developed in this work, described in Chapters 5 and 8, are built around this as a basic principle.

5. PARALLEL MCPSC FOR A NETWORK-ON-CHIP PROCESSOR ARCHITECTURE

In this chapter we discuss in detail the Intel SCC many-core processor. The software and hardware architecture associated with the processor is discussed. We describe in detail the MCPSC application developed in this work including the algorithmic skeleton library developed.

5.1 The Intel SCC

The ”Single-chip Cloud Computer” (SCC) , is an experimental Intel architecture research microprocessor containing 48 cores integrated on a silicon CPU chip [74]. It has multiple dual x86 core tiles arranged in a 6x4 grid, memory controllers and 24-router mesh network, depicted in Figure 5.1. The technology used is intended to scale many-core processors to 100 cores and beyond, forming an on-chip network. The novel many-core architecture includes innovations for scalability in terms of energy-efficiency, core-core communication and techniques that enable software to dynamically configure voltage and frequency. The cores on the chip can run separate operating systems acting like independent computational nodes that communicate with other nodes over a packet-based network.

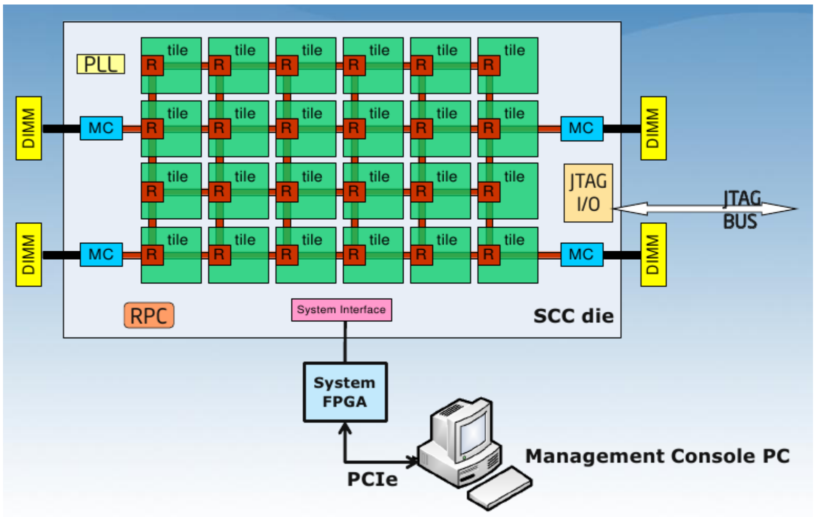


Figure 5.1: Intel SCC System Overview [74].

Networks on Chip

Network on Chip (NoC) is a method to design the communication subsystem between many cores residing on a single System-on-Chip (SoC). NoCs can span synchronous and asynchronous clock domains or use unclocked asynchronous logic [17]. Networking

Table 5.1: Bus-versus-Network Arguments (taken from [17])

Bus Pros & Cons		Network Pros & Cons	
Every unit attached adds parasitic capacitance, therefore electrical performance degrades with growth.	−	+	Only point-to-point one-way wires are used, for all network sizes, thus local performance is not degraded when scaling.
Bus timing is difficult in a deep submicron process.	−	+	Network wires can be pipelined because links are point-to-point.
Bus arbitration can become a bottleneck. The arbitration delay grows with the number of masters.	−	+	Routing decisions are distributed, if the network protocol is made non-central.
The bus arbiter is instance-specific.	−	+	The same router may be re-instantiated, for all network sizes.
Bus testability is problematic and slow.	−	+	Locally placed dedicated BIST is fast and offers good test coverage.
Bandwidth is limited and shared by all units attached.	−	+	Aggregated bandwidth scales with the network size.
Bus latency is wire-speed once arbiter has granted control.	+	−	Internal network contention may cause a latency.
Any bus is almost directly compatible with most available IPs, including software running on CPUs.	+	−	Bus-oriented IPs need smart wrappers. Software needs clean synchronization in multiprocessor systems.
The concepts are simple and well understood.	+	−	System designers need reeducation for new concepts.

theory and methods of on-chip communication are employed in NoC designs to improve communication capabilities for SoCs. These methods have typically yielded significant improvements over conventional bus and crossbar interconnections as the number of cores increases [108]. Further, NoCs also improves the scalability and power efficiency of SoCs. Several leading processor manufacturers such as Intel, IBM etc., have developed processors containing tens of cores on a single chip using networking for inter-core communication [109, 108]. Table 5.1, lists the Pros and Cons of the two main strategies for inter core communication and highlights the benefits of NoCs.

A NoC typically consists of multiple point-to-point data links interconnected by routers. This allows messages to be relayed from any source module to any destination module using the data links [12]. Routing decisions, for data flow, is made at the routers [49]. Thus a NoC is similar to a modern telecommunications network, using packet switching over multiplexed links.

A key aspect in using NoCs to improve performance of programs is understanding the memory hierarchy of the specific chip for which the program is being targeted. References, such as [108], discuss the issue in more detail and present unified models of memory hierarchies found in common NoC based many-core processors.

Attempts have been made to utilize many-core architectures in Bioinformatics application and also in the subdomain of structural proteomics. The majority of the applications attempted belong to pairwise or multiple sequence comparison category. NoCs, however, have not yet been extensively exploited in Bioinformatics [107]. The NoC based implementation of the sequence alignment algorithm, Needleman and Wunsch, developed in [107] which shows significant speedup potential.

5.1.1 Hardware Architecture

Essentially the SCC resembles a cluster of computer nodes capable of communicating with each other in much the same way as a cluster of independent machines. Salient features of the SCC hardware architecture relevant to programming the chip are listed in Table 5.2. The SCC is considered an excellent example of a cloud data center and a very interesting platform due to the following features:

- High-speed network
- Improved communication between cores
- Enhanced performance
- Energy efficiency
- Intelligent data movement between cores

Table 5.2: Salient features of the SCC Chip by Intel.

Core architecture	6x4 mesh, 2 Pentium P54c (x86) cores per tile
Local cache	256KB L2 Cache, 16KB shared MPB per tile
Main memory	4 iMCs, 16-64 GB total memory

Figure (5.2), shows details of individual tiles on the SCC many-core processor. Each core of the SCC has L1 and L2 caches. In the SCC architecture, the L1 caches (16KB each) are on the core while the L2 caches (256KB each) are on the tile next to the core with each tile carrying 2 cores. Further, each tile also has a small message passing buffer (MPB), of 16KB, and is shared among all the cores on the chip. Hence, with 24 tiles, the SCC provides a message passing buffer of size 384KB. The SCC therefore provides a hierarchy of memories usable by application programs for different purposes including processing and communication.

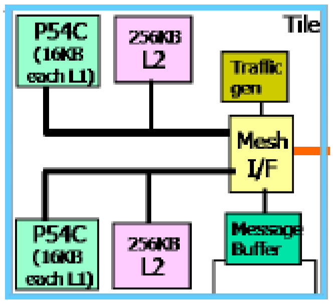


Figure 5.2: Each of the 24 tiles in the SCC processor contains 2 cores with L1 and L2 caches and a message passing buffer (MPB) [74].

A message-passing program sends a message from one core to another by placing the data in the message passing buffers. The SCC also has a Traffic Generator, which is a unit used primarily to test the performance capabilities of the mesh. This is done by injecting and checking traffic patterns and may not be used in normal operation by application programs. The Mesh Interface Unit (MIU), connecting the tiles to the mesh, build packets from data to put it in the mesh and unpacks data coming in from the mesh. It controls the flow of data on the mesh with a credit-based protocol and uses the round-robin scheme to arbitrate between the two cores on the tile [74].

5.1.2 Software Architecture

From a programmers perspective the SCC provides a super computing environment where the paradigm shift from standard programming is small. Programming a standard x86 core and using Message Passing Interface for fast message exchange between processes is currently employed widely for cluster programming. In order to facilitate development, a minimal programming library RCCE, a compact, lightweight communication environment written in C, is available with a basic API for MPI [2]. The main features of the RCCE library are:

- RCCE supports message passing APIs and allows mapping tasks onto many-core chips.
- RCCE usage requires parallel programming experience with an understanding of the underlying SCC hardware.

With the help of RCCE it is possible to build full scale application programs for the SCC chip. The simple message passing environment provided by the RCCE makes it easy to build communication systems with one-sided communication, such as is sufficient for inter-task communication for parallelizing algorithms. Software for the SCC can be compiled using the Intel C/C++ compiler or the GNU C/C++ compiler (gcc). Since the cores on the SCC are x86 type the compilation must be performed for a 32 bit system.

An important aspect of programming the SCC, is the way the system memory is mapped into a core's address space. On the SCC the physical address space (32-bit) of each core is divided into 256, 16 MB, chunks, which are mapped through a look-up table (LUT) to a 34-bit system address and the destination coordinate in the mesh [75]. The core address starts off at 32 bits. The top 8 bits go to the LUT, which puts out 22 bits. The lower 24 bits pass through. Of those 22 bits coming from the LUT, the lower 10 bits are perpended to the 24 bits that passed through, resulting in a 34-bit address. This is the address sent to the memory controller. Each memory controller can address up to 16GB, hence the 34 bits. The LUT configuration determines whether a physical address refers to off-chip DDR3 memory or on-die message passing buffer memory. Coherence between cores must be managed by the programmer. A test-and-set register is provided for each core, which can be used by application programs that require such coordination.

A program written using RCCE is typically expected to belong to the Single program multiple data (SPMD) computing model. From the programmers view each instance of the program runs on a different core of the SCC. Each process receives a different data input resulting in parallel processing being achieved. Message passing is employed, using the RCCE, in order to synchronize the processes. RCCE also provides the programmer with constructs for creating a shared memory space such that multiple cores can access it. These constructs provide a useful method for sharing large data between multiple cores while avoiding the cost of clogging the network with messages.

5.2 The Rckskel library

5.2.1 Overview

In order to facilitate development of PSC algorithms targeted for the SCC, we built a small C library, called *rckskel* (rck Skeleton Library). We found that higher level constructs which hide the details of the inter-process communication, e.g. polling and waiting, would simplify introducing parallelism in PSC algorithms. To retain the flexibility offered by RCCE, in combining processes running on different cores to form a pipeline or to perform parallel execution, we decided to use algorithmic skeletons for building the library.

Rckskel implements algorithmic skeletons in addition to providing wrappers for common RCCE related tasks. The library provides convenient wrappers for common operations, such as environment initialization, testing how many cores are available to the program, setting debug levels and finalization, performed by all applications built with the RCCE.

5.2.2 Operational semantics

The *rckskel* library provides a programmer with both task and data parallel skeletons. The skeletons process a stream of input tasks and produce a stream of output results. Furthermore, the skeletons are stateless and fully nestable.

- SEQ: This is a task sequencing construct where a list of tasks, which may contain sub-tasks, are assigned to a set of processing elements but will be executed only in the order in which they were given. This construct is typically useful for defining the leaf node operations in a hierarchy of operations. Parameters to the function include, *ue_count* (the number of processing elements), *ue_ids* (the specific processing elements), *check_ready* (function to be used for checking if a processing element has been initialized) and *task_count* (the number of sub-tasks). The dots at the end of the function definition denote a variable argument list of tasks. This construct runs the jobs on the corresponding processing elements sequentially. Once the last batch of jobs is submitted it returns to the calling code, without waiting for them to complete.

  ```
  void SEQ(int ue_count, int *ue_ids,
  ```

```
  int (*check_ready)(int),
  int task_count,...);
```

- PAR: This is a task mapping construct where each task, which may contain sub-tasks, is assigned a set of processing elements and the sub-tasks are processed in parallel. This construct assigns the jobs to the corresponding processing elements and returns to the calling code, without blocking till the jobs are completed. The calling code will need to call the COLLECT construct if it needs to wait for the jobs to finish.

```
void PAR(int ue_count, int *ue_ids,
  int (*check_ready)(int),
  int task_count, ...)
```

- COLLECT: This is a task collection construct where a list of processing elements is polled, till all elements return the results of their processing. The function to be applied to the data returned can be specified and may perform operations on the returned data, e.g. storing in an array for later use. The function does not block waiting on processing elements in order but rather performs a busy round-robin loop.

```
void COLLECT(int ue_count,
  int *ue_ids, int (*collector)(int));
```

- FARM: This is a master-slaves task execution construct. A controlling task is created which ensures the execution of the tasks given until they complete. The master process in this setup runs on one of the cores of the SCC. This is the highest level construct currently implemented and takes care of ensuring that all processing elements are available before starting the processing and that all processing is completed when it returns. A task tree is generated from the parameters of the function depending on the sub-tasks. If no tasks have been specified FARM works as a PAR followed by a COLLECT. The tasks in the tree are processed as specified, in parallel or in sequence, using the PAR, SEQ and COLLECT constructs described above.

```
void FARM(int ue_count, int *ue_ids,
  int (*check_ready)(int),
  int task_count, ...);
```

5.2.3 Instantiating rckskel skeletons

In order to describe working with the *rckskel* library we will consider the problem of adding a vector of numbers and discuss implementations of the problem using *rckskel* skeletons. The user must define the basic task, which in this case is addition of a given set of input numbers.

An example of using the seq operation to add the first one-thousand integers is show in Listing 5.1. The master node prepares the vector of data and passes it to a processing node to add. The processing node blocks, waiting from data from the master, and on receiving the data performs the addition and returns the results.

It can be noted that no parallelism is exploited in the seq code in case more than one processing elements are available. To demonstrate parallel addition of the same set of numbers we recreate the scenario with the map function. In order to use the map operation the user must define the split and a merge functions. The main program using the map operation would then be as shown in listing 5.2.

As can be seen from the listing, the change in complexity of the code in order to use multiple cores is small. The number of cores utilized by the map operation depends on the split function supplied by the user and the available cores. In the example described above the split function creates three subsets from the data, if three processing elements are available all three subsets will be processed simultaneously. However, if fewer processing elements are available the parallelism achieved will be smaller. The dynamic allocation of available processing elements is transparently handled by the library.

To conclude we will present an example of instantiating a Master-slaves setup using *rckskel* operations. A template for a master-slaves implementation utilizing *rckskel* is shown in Listing 5.3. The function name *RCCE_APP* is the entry point for applications built with RCCE. The division of master and slave related code, after the common variable declaration and environment initializations, is highlighted by the *if* block, as typical for SPMD-style code. The *MASTER_ID* is defined globally with a default value of 0, which implies that the first core available to the program will be used to run the master process. The master processing starts with creation of a FARM task where application specific methods, *master_send_job* and *master_receive_result*, are supplied. These are application specific because the data structure used by different applications vary.

The jobs to be processed, as well as the SCC cores on which the jobs should be run (*ue_-ids*) are also specified in the task definition. The jobs are run using the *FARM* execution construct. The *check_ready* method supplied is used to start distribution of a job to a slave when the slave becomes ready. The slave processing proceeds by waiting in a busy loop, receiving data from the master process and returning results once the processing is complete, until it receives a terminate message.

The *client_receive_job* method contains a blocking wait on the master and application

Listing 5.1: An example of using the Seq operation. The controlling node prepares a job with some integers and passes it to a client which returns a sum of the integers. Assume two client cores are available

```
int main(int argc, char **argv) {
        // variable declaration & environment setup
        ...
        task_t *task1, *task2;
        if (ue_id == MASTER_ID) {
                task1 = create_task(MAP_rckskel, '0',
                        &master_send_job, &master_receive_result,
                        number_to_add1, ue_ids1);
                task2 = create_task(MAP_rckskel, '0',
                        &master_send_job, &master_receive_result,
                        number_to_add2, ue_ids2);
                SEQ(ue_count, ue_ids, &check_ready, 2, task1, task2);
        } else {
                // local initializations
                ...
                while(TRUE) {
                        // get and process job or terminate
                        if (client_receive_job() == TRUE_) break;
                        // return sum
                        RCCE_send((char *) &result,
                                sizeof(RESULT_BLOCK), MASTER_ID);
                }
        }
        ...
}
```

Listing 5.2: An example of using the Map operation to add a set of integers. Each chunk is sent to a different processing element and the sum's returned by these are collected by the master process using the user defined *collector* function.

```
int main(int argc, char **argv) {
        // variable declaration & environment setup
        ...
        task_t *task;
        if (ue_id == MASTER_ID) {
                task = create_task(MAP_rckskel, '0',
                        &master_send_job, &master_receive_result,
                        number_to_add, ue_ids);
                MAP(ue_count, ue_ids, &check_ready, 1, task);
                COLLECT(ue_count, ue_ids, &collector);
        } else {
                // local initializations
                ...
                while(TRUE) {
                        // get and process job or terminate
                        if (client_receive_job() == TRUE_) break;
                        // return sum
                        RCCE_send((char *) &result,
                                sizeof(RESULT_BLOCK), MASTER_ID);
                }
        }
        ...
}
```

specific processing of data. The results are returned to the master in an application specific data structure of type *RESULT_BLOCK*.

Listing 5.3: An example of using the Farm operation to perform all-vs-all pairwise PSC. User defined functions are used for loading data and saving results. The Farm operation ensures all available cores of the SCC are utilized for processing different pairs of structure comparisons in parallel.

```
int main(int argc, char **argv) {
        // variable declaration & environment setup
        ...
        task_t *task;
        if (ue_id == MASTER_ID) {
                task = create_task(MAP_rckskel, '0',
                        &master_send_job, &master_receive_result,
                        job_indexes, ue_ids);
                FARM(ue_count, ue_ids, &check_ready, 1, task);
        } else {
                // local initializations
                ...
                while(TRUE) {
                        // get and process job or terminate
                        if (client_receive_job() == TRUE_) break;
                        RCCE_send((char *) &result,
                                sizeof(RESULT_BLOCK), MASTER_ID);
                }
        }
        ...
}

int clien_receive_job() {
        // wait to receive instruction from master
        RCCE_recv((char *) &client_in_data,
                sizeof(MasterToClientTransferBlock), MASTER_ID);
        // early exit if master is asking to leave
        if (client_in_data.die == TRUE_) return TRUE_;
        // do PSC and store results in appropriate structures
        ...
        return FALSE_;
}
```

5.3 Software framework for porting PSC methods

In this section we present the framework we used for porting a PSC methods, to many-cores processor technology. Typically PSC involves one-to-all or all-to-all comparisons of protein structures. Each pairwise structure comparison is an independent unit operation. Several pairwise comparison operations can therefore be performed in parallel if the computing resources are available. Many-core processors provide several computing elements, connected by a high speed network allowing distribution of jobs.

We propose the use of a master-slaves parallel implementation of the PSC algorithm, where the master process is responsible for loading the structures to be compared and distributing the pairwise comparison jobs to the slave processes. In a many-core processor system, where the cores are connected by a high speed interconnection network, the data transfer overhead is relatively small. By limiting data loading to a single process we avoid bottlenecks, created due to multiple processes accessing the same data concurrently. The slave processes perform pairwise structure comparison on structure data received from the master and return results of processing to the master. The slave processes continue the cycle until they receive a terminate signal from the master. This simple strategy, where the master process polls the slave processes in a round-robin manner, allows efficient use of the computing resources available to perform pairwise protein structure comparisons.

We used the framework to port a TMalign and CE, to the master-slaves model. Additionally we developed a C++ port of the USM PSC method in order to include it in all-vs-all MCPSC processing. We made use of the $rckskel$ library to combine serial processing of the USM PSC tasks and parallel processing of CE and TMalign PSC tasks, generated for all-vs-all comparison of protein 3D structure.

In order to develop the parallel implementation, we first ported TMalign software to a pure C implementation. The Fortran code of TMalign was converted to C using an F2C converter. The resulting code had a dependency on the F2C library and was therefore cleaned up manually to remove all dependencies. This required altering the data types, as well as implementing four basic math functions: *max*, *min*, *dabs* and *abs*. Additionally, the I/O operations were changed to use C functions. We compared the output of the C implementation with that of the Fortran implementation and found matching results.

In order to develop the parallel implementation of CE we used the C++ sources as the base code. The C++ implementation of CE uses a small structure manipulation library as an external binary. This library was modified in order to link it to the main CE code and deprecate the need for the program to run a binary. A master-slaves parallel implementation of the C/C++ code, using the *rckskel* library was then generated similar to the master-slaves port developed for TMalign in [114]. A C++ implementation of the USM method was also developed. The implementation makes use of C++ $gzip$ compression routines to compare the protein structures by the compression ratio of their contact maps.

The parallel algorithm developed makes use of the master-slaves implementation provided by the $rckskel$ library. The implementation contains a single master process which

```
Data: $Q$: query protein structures, $D$: database of known protein structures
// sequential process
calc_usm_dist(Q, D)
// parallel process
for m in [CE, TMalign] do
    for q in Q do
        for d in D do
            jobs.add(in $n$, using method $m$, protein pair [q, d])
        end
    end
end
par_calc_psc_dist(jobs)
```

Algorithm 2: A pseudo-code of the MCPSC computation implemented in this work. The USM jobs are processed sequentially, while CE and TMalign pairwise computations are parallelized. Each node n performs a pairwise comparison of proteins (q, d) using one of the methods (m). $par\_calc\_psc\_dist$ assigns each pairwise task to a unique processing node. If n is specified the jobs assignment is done to the specific node otherwise an available node is used.

generates a list of jobs, each involving a single pairwise protein structure comparison using a given PSC method. The USM jobs are processed sequentially by the master due to the fast processing speed for these jobs as shown in the results section. CE and TMalign pairwise PSC jobs are then processed in parallel by distributing them to the slave processes available to the master. The first core supplied to the program is used to run the master process and all subsequent cores are used to run slave processes. A pseudo-code of the process implemented by the application is listed in Algorithm 2.

The pseudo-code listed in Algorithm 2 covers the general case of comparing a set of query proteins with a set of database proteins using the three methods ported in this work. When the set of query proteins is the same as the set of database proteins the pseudo-code covers the all-vs-all comparison. Benchmarking of the performance of various strategies for job distribution in this work were carried out by measuring the time for completion of all-vs-all comparisons of proteins. When the query proteins are different from the database proteins the pseudo-code covers the many-to-many comparison which was used in this work for comparing the performance of MCPSC with its component PSC methods.

6. OPTIMAL LOAD-BALANCING FOR MCPSC ON THE SCC

In this chapter we look at the load balancing methods implemented to distribute PSC jobs to the multiple processing elements of the SCC. In this chapter we discuss the basis that was used for developing load balancing strategies. Experiments designed to compare the load balancing strategies in terms of time to perform all-to-all MCPSC are presented and the results of the experiments discussed.

6.1 Load balancing methods

The ability to dynamically adapt an unstructured mesh consisting of processing elements (PE) is necessary for solving computational problems with evolving physical features. An efficient parallel implementation using such a mesh requires load balancing. Dynamic load balancing aims to balance workloads across the PEs at runtime while attempting to minimize the communication. A problem is therefore load balanced when the PEs have nearly equal loads. The objective, of course, is to assign work to the PEs so that the total runtime is minimized.

6.1.1 Static partitioning

Partitioning is a static load balancing method in which the master determines the jobs to be assigned to each slave before the job distribution commences. Determining this assignment (job-to-core mapping) is equivalent to constructing N equal partitions for a list of elements, which is an NP-hard problem [78]. In our case N is the number of cores equal to 47 when all SCC cores are used. We investigated methods for PSC jobs partitioning based on the sum and the product of the lengths of the two proteins to be compared.

6.1.2 Dynamic round robin

Round-robin [120] is a dynamic load balancing method in which the master process maintains a list of jobs and hands the next job in the list to the next free slave process. It is worth noting that if the list is presorted based on attributes that are known to be proportional to the processing time this would naturally result in little to no idle times for the slave processes. However, in such a scheme the master may become a bottleneck as the number of slaves increases.

6.2 Experiments

We have statically partitioned the PSC jobs using as attribute the sum of lengths (product of lengths) of the paired proteins to be compared. The partitions are generated using the Longest Processing Time (LPT) algorithm: The jobs are sorted in descending order by sum of lengths (product of lengths) and then assigned to the partition with the smallest total running sum. These partitions will be referred to as “Greedy” partitions from now on. Since the job partitions are processed on different cores this strategy is expected to balance the processing times by reducing idle times.

Figure 6.1 shows the partitions generated using a random partitioning and a greedy partitioning scheme based on the sum of lengths and the product of lengths as attributes. Partitions created with the random scheme, with equal number of pair proteins per partition, vary greatly in terms of the total normalized sum of the sum (or product) of the pair protein lengths. The size of this normalized sum is expected to be proportional to the overall processing time each partition will require when assigned to a core. Therefore, if each partition of PSC tasks created with the random scheme is processed on a different core, several cores will have significant idle times at the end. On the contrary, partitions created with the greedy scheme are almost equal as to the normalized sum of the sum (or product) of the pair protein lengths. Hence, if each partition of PSC tasks created by the greedy scheme is processed on a different processing element (PE), idle times are expected to be minimized.

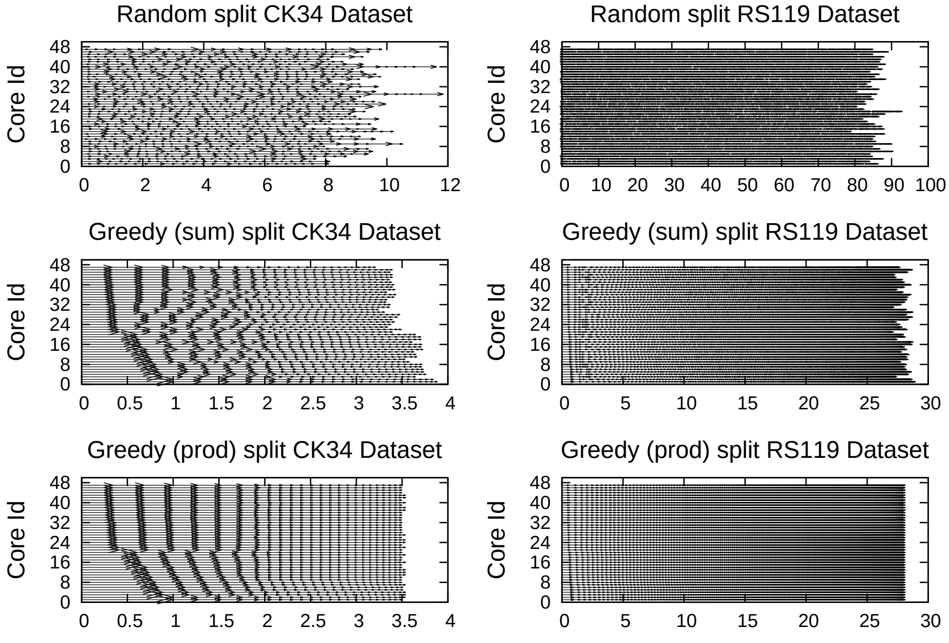


Figure 6.1: The 47 partitions created from list of PSC tasks sorted randomly or by the sum (product) of lengths of pair proteins. Each horizontal line represents the sum of the normalized lengths of the protein pairs assigned to that partition.

6.3 Comparison of load balancing strategies

Figures 6.2 and 6.3 show the space-time parallel execution profiles under different job assignment schemes. In Figure 6.2 the space-time execution profiles of the random and greedy static partitioning schemes are compared. For random partitioning the jobs are randomly partitioned into 47 sets, one set per slave, while in the greedy partitioning the partitions created using the LPT algorithm with the product of lengths of paired proteins as an attribute. In Figure 6.3 the space-time execution profiles of dynamic round-robin job distribution schemes are compared. In the simple round-robin strategy the global job list maintained by the master is filled with jobs randomly, while in the sorted round-robin strategy the list is filled with jobs and then sorted according to the sum (product) of lengths of paired proteins.

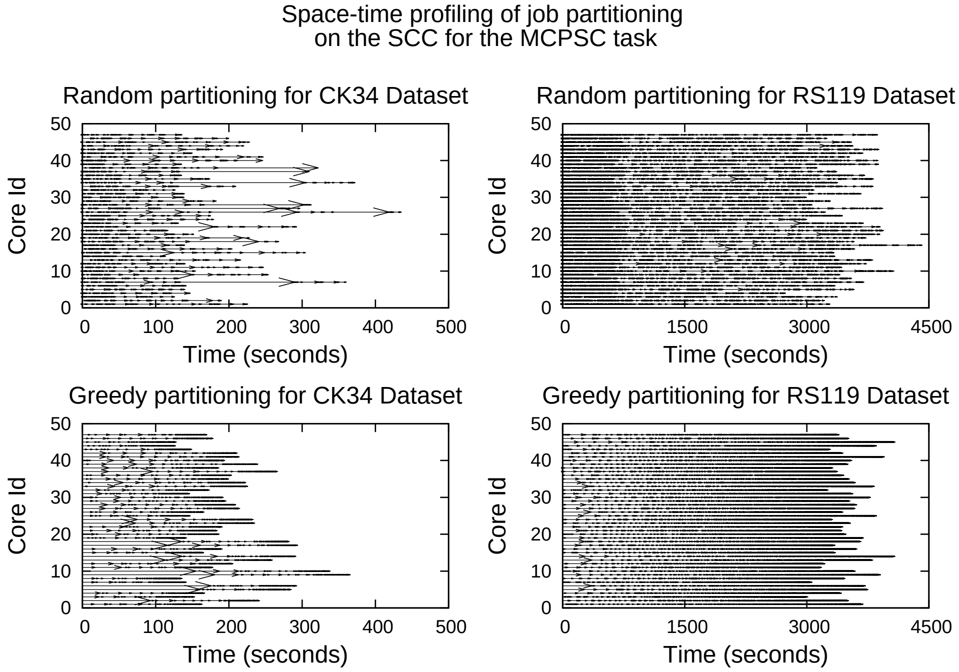


Figure 6.2: Space-time profiling of the partitioning schemes for the CK34 and RS119 datasets. Each horizontal line represents work performed by an individual core of the SCC.

In Table 6.1 we compare the performance of the master-slaves setup, with one master and 47 slaves, to that of a single core of the SCC and show the speedup and the efficiency achieved using the CK34 and RS119 datasets. TMalign and CE jobs were run under the master-slaves model, while USM jobs were processed by the master due to the negligible time required by these jobs.

The greedy static partitioning scheme improves on the performance of the random partitioning scheme, but is inadequate for the SCC where the cost of data transfer between cores is low. This scheme would be of interest in computer clusters where interconnection networks are slow. The greedy partitioning scheme could also be beneficial on larger many-core processors, where the master may become a bottleneck, making off-line cre-

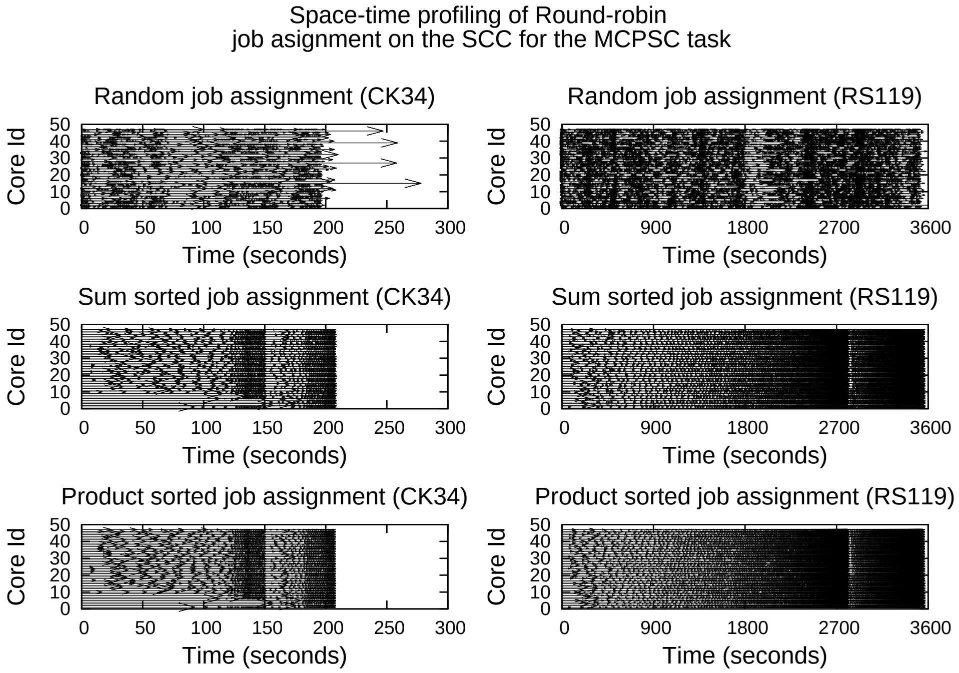


Figure 6.3: Space-time profiling of the round-robin job assignment for the CK34 and RS119 datasets. Each horizontal line represents work performed by an individual core of the SCC.

Table 6.1: Comparison of speedup and efficiency achieved by the load balancing schemes in processing the all-to-all MCPSC task vis-a-vie the single core (SCC) processing times. The efficiency is calculated assuming 47 processing elements (one PE serves as master). All times are in seconds.

Load Balancing scheme	**Dataset CK34**			**Dataset RS119**		
	Time	**Speedup**	**Efficiency**	**Time**	**Speedup**	**Efficiency**
Serial (1 SCC core)	9698	-	-	166000	-	-
Random	479	20	0.43	4765	35	0.74
Greedy Partitioning (sum)	409	24	0.50	4300	39	0.82
Greedy Partitioning (product)	396	28	0.59	4433	38	0.80
Round Robin (sum sorted)	239	41	0.86	3930	42	0.90
Round Robin (product sorted)	238	41	0.86	3930	42	0.90

ation of batches of PSC tasks essential. In such a scenario however, other techniques such as work-stealing, would need to be assessed before selecting the most appropriate method.

The round-robin dynamic job assignment strategy outperforms the best offline partitioning scheme. Furthermore, the sorted round-robin flawlessly balances out the cores processing times on both datasets, making it the best approach for distributing jobs to the cores of the SCC. In this approach, the protein pairs are sorted in descending order of product (or sum) of pair protein lengths. Specifically, for the one-to-many PSC case, we simply retrieve the database proteins in descending order of length and create pairwise PSC tasks with the query protein. For the many-to-many PSC case however, we need to determine the correct order of the proteins when the query proteins are received.

The sorted round-robin strategy distributes the jobs most efficiently for the proposed master-slaves setup on the Intel SCC with 47 slaves. This setup achieves a 42 fold (40 fold) speedup for the RS119 (CK34) dataset as compared to a single core of the SCC. The speedup is almost linear, which suggests that high efficiency can be achieved even when the many-core processor has more nodes. A bigger many-core processor, however, may require using a hierarchy of masters to avoid a bottleneck on a single master node. In such a scenario, a combination of partitioning schemes and round-robin job assignment could be used to distribute jobs and minimize idle times. For instance, on larger NoC processors a well balanced solution may require the use of clusters of processing elements, concurrently processing PSC jobs with different methods. All PEs computing jobs of a PSC method would receive jobs from a specific submaster.

7. PERFORMANCE BENCHMARKING: MULTI- VS MANY-CORE PROCESSOR

In this chapter we compare the performances of the Intel SCC and the Intel i7 for all-to-all MCPSC. In the first section we discuss the results of using the SCC in a master-slaves setup to using it in a distributed setup. Comparison of the two setups is carried out by performing all-to-all PSC using TMalign in the distributed setup and its master-slaves port rckAlign.

In subsequent sections we present results of comparing the performance of the Intel SCC with that of the Intel i7 using several protein structure datasets of varying sizes. Finally, in the last section we present a qualitative comparison of MCPSC with its component PSC methods.

7.1 SCC Usage

We compared *rckAlign* running on the SCC and the existing TMalign software used in a distributed manner, to perform all-vs-all comparison for the CK34 dataset. In the distributed TMalign version, a controlling master process is run on the SCC host machine (MCPC). The host process creates a list of jobs and distributes them to individual cores of the SCC. Each process is responsible for loading its own structure data. Issuing a job to a core is performed using the *pssh* remote execution command available on the MCPC. On the other hand, when using *rckAlign* the data is loaded by the master process and slave processes receive the data for pairwise structure comparison from the master process using the SCC network. Results of the experiment are shown Figure 7.1.

As observed in Figure 7.1, *rckAlign* achieves faster processing times than when the master process is running on the MCPC. There are two main reasons for this behavior: (a) disk access through the Network File System (NFS) creates a bottleneck when multiple processes are trying to access the data, and (b) high environment setup costs incurred while issuing remote processing. The SCC-MCPC setup provides NFS access to disks installed in the MCPC for the Linux system running on each individual core. When processes running on several cores try to access the data stored in the shared partition of the MCPC disk a bottleneck is created, by the MCPC disk controller, resulting in overall increase in processing time. This situation is not encountered when all the protein related data is loaded by a single process as is the case for *rckAlign*. Further, the master process running on the MCPC starts a new process for each pairwise comparison, which has its environment setup cost, thus increasing further the total processing time. Results of the comparison thus validate the superiority of the approach where the master process runs on one of the SCC cores rather than on the host PC machine. Additional overhead cost is also avoided in *rckAlign* because all processes, master and slaves, are initialized once for a given number of slaves.

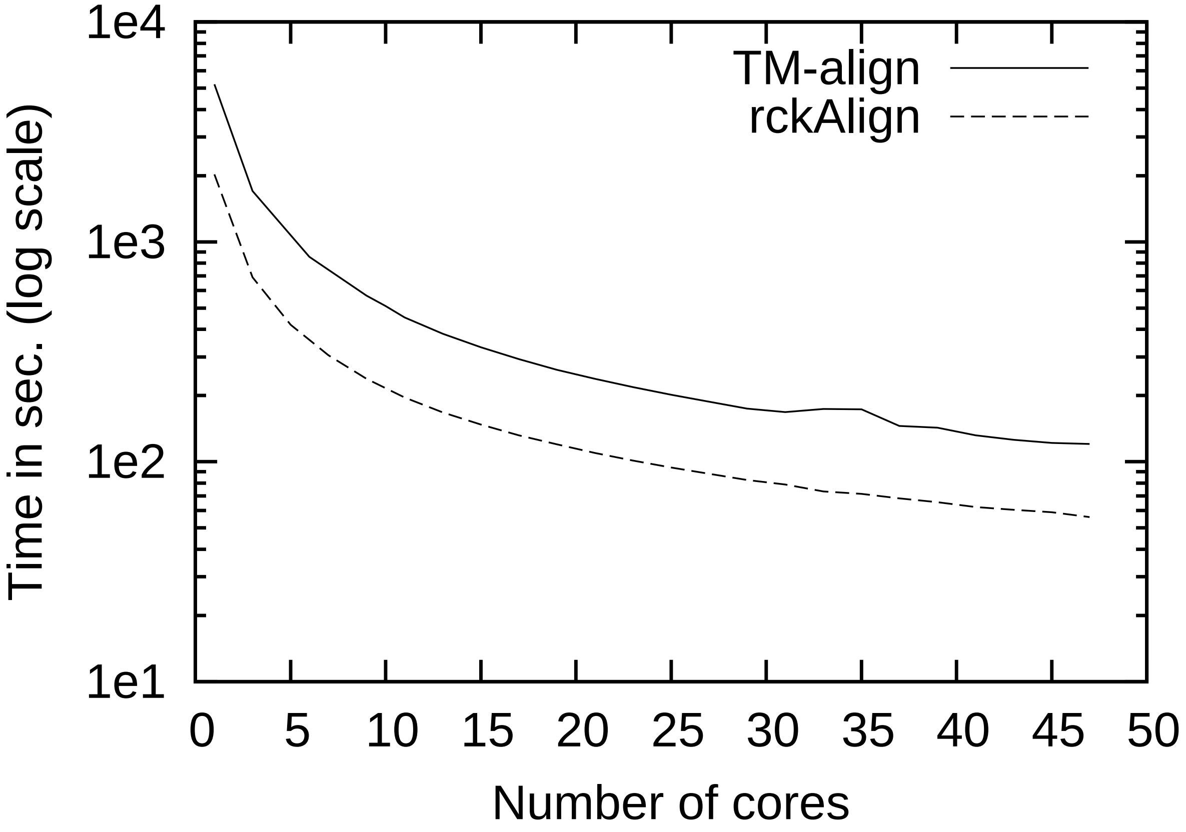


Figure 7.1: Performance comparison of parallel *rckAlign* with that of distributed TMalign software (C port) for the Chew-Kedem dataset (CK34) as the number of slave cores used is increasing.

Table 7.1: Time required for the baseline all-vs-all PSC task using TMalign (C port) on two different processors and datasets. All times are in seconds.

Processor	**Datasets**	
	CK34	**RS119**
AMD Athlon II X2 250 2.4 GHz	406	7298
Intel P54C Pentium 533 MHz	2029	28597

The times required for performing the all-vs-all comparisons for the two datasets using the serial implementation of TMalign were measured, both for the AMD Athlon II X2 250 2.4 GHz processor and for the P54C Intel Pentium SCC Core at 533 MHz. The times obtained for the SCC core were used as the baseline for calculating the speedup achieved by *rckAlign* running in parallel on the SCC. For running on a single SCC core, the TMalign program was modified slightly, to load all the protein structures to be compared at the start in order to be equivalent to the way *rckAlign* works. Results of the experiment are presented in Table 7.1. When comparing baselines performance (using one core) the faster AMD CPU outperforms as expected the much slower Intel P54C core.

The speedup achieved by the parallel *rckAlign* implementation on the SCC was mea-

Table 7.2: Performance of rckAlign in an all-vs-all PSC task on the CK34 and RS119 datasets.

Slave Cores	CK34		RS119	
	Speedup	Time (sec)	Speedup	Time (sec)
1	1	2029	1	28597
3	2.94	689	2.96	9654
5	4.82	420	4.91	5818
7	6.66	305	6.95	4114
9	8.52	238	8.94	3195
11	10.34	196	10.97	2605
13	12.09	168	12.95	2208
15	13.74	148	14.88	1921
17	15.36	132	16.76	1705
19	16.89	120	18.64	1534
21	18.53	109	20.59	1389
23	20.03	101	22.52	1270
25	21.56	94	24.52	1166
27	23.02	88	26.49	1079
29	24.52	83	28.45	1005
31	25.72	79	30.37	941
33	27.68	73	32.32	885
35	28.43	71	34.21	836
37	29.75	68	36.14	791
39	30.97	65	38.01	752
41	32.60	62	39.74	719
43	33.59	60	41.49	689
45	34.45	59	43.40	659
47	36.17	56	44.78	640

sured for both datasets CK34 and RS119. This experiment was designed to measure the speedup achieved as a function of the number of slave processes used as well as the size of the datasets. The master process loads all the domains to be processed and creates a list of jobs with all pairs (all-vs-all). The master process then distributes N jobs among the N slaves and the results are gathered by polling the slaves in a round-robin manner. The distribution of jobs and collection of results is carried out until all jobs have finished. Communication between the master and the slaves is carried out using functions available in the *rckskel* API. The number of active slaves was varied from 1 to 47 in order to assess the impact of increasing the number of cores available for parallel processing. Results of the experiment are shown in Figure 7.2 with detailed values presented in Table 7.2.

Figure 7.2 shows that the speedup achieved by *rckAlign* is increasing almost linearly with the number of cores available for running slave processes. This is a result of the low cost of exchanging data between processes running on cores connected by a high speed interconnection network. If the data transfer times were high, the master process would

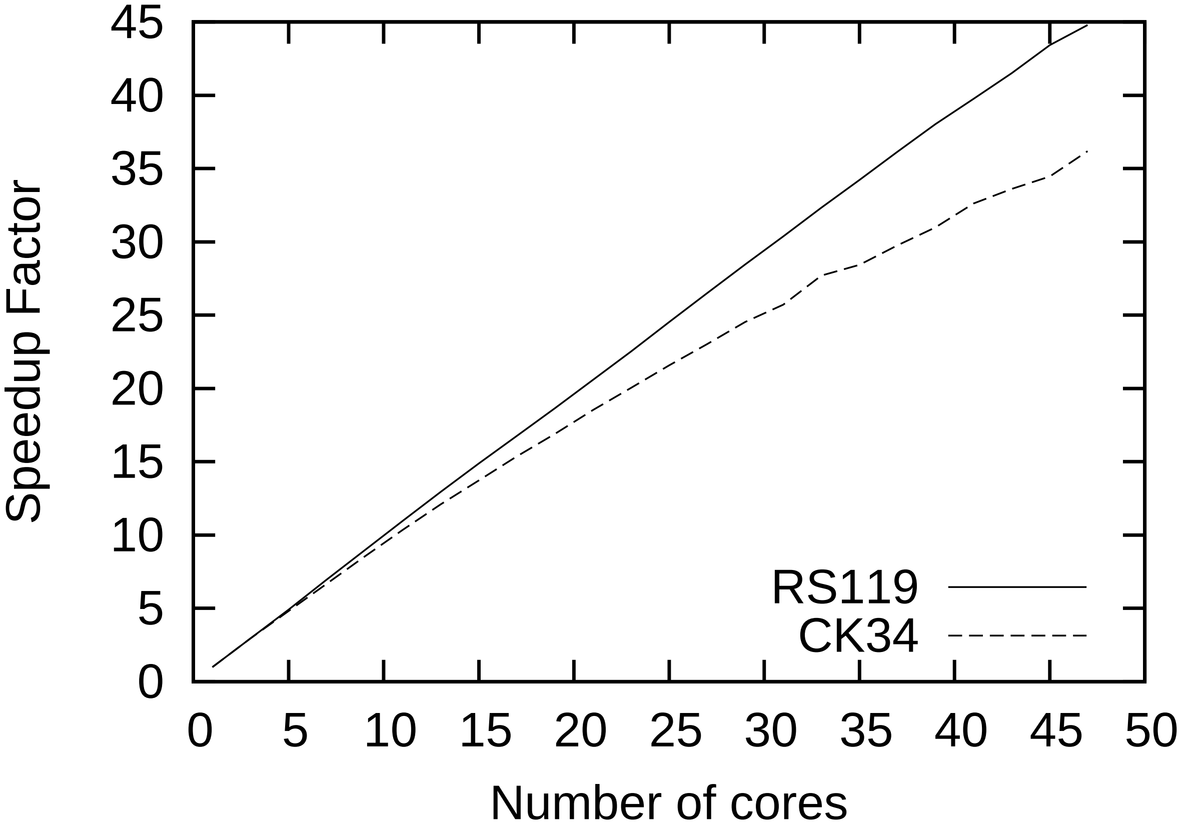


Figure 7.2: Speedup achieved by *rckAlign* as the number of slave cores is increasing (from 1 to 47) for the Chew-Kedem (CK34) and the Rost-Sanders (RS119) datasets. The speedup reported is relative to the performance on a single core of the SCC.

become a bottleneck and core utilization would be reduced, resulting in larger overall processing times. Since an almost linear speedup is observed, the simple master-slaves implementation appears sufficient to exploit the parallelism offered by many-core processors to this problem. This observation suggests that further speedup can be achieved on many-core processors with a greater number of cores. It should be mentioned that no load balancing was applied to the allocation of jobs to slaves in our implementation. It has been suggested that good load balancing approaches can improve the performance of all-vs-all PSC [113].

From the results in Table 7.3 we observed that *rckAlign* running in the SCC at 533 MHz achieves an 11 fold speedup over the AMD 2.4GHz processor and a 44 fold speedup over a single Intel P54C 533 MHz processor when using the RS119 dataset. We also observe that the larger the dataset the higher the speedup observed. These results suggest that many-core NoCs with fast interconnection networks and faster processor cores than the SCC will be ideal candidates for delivering high performance for all-to-all PSC tasks applied to large size protein databases, as needed for combinatorial drug design.

Comparison of the performance of *rckAlign* with the serial implementations run on a sin-

Table 7.3: Comparison of times required by TMalign and *rckAlign* for performing all-vs-all PSC on the CK34 and RS119 datasets. All times are in seconds.

Dataset	TMalign AMD@2.4GHz	TMalign Intel@533MHz	rckSkel SCC(all cores)
CK34	406	2029	56
RS119	7298	28597	640

gle core of the SCC and the use of the faster processor, suggest that there is scope for achieving higher overall speedups, if the many-core processor provides faster cores. It is possible that the single master strategy would become the bottleneck, if slave processes were running on faster cores or faster network. However, this can be tackled by implementing a hierarchy of master processes such that a master does not become a bottleneck for the slaves it controls.

7.2 Comparison with serial implementation

Table 7.4 provides the times taken to perform all-to-all comparison on the PC and on a single core of the SCC, with the CK34 and RS119 datasets. The software took longer to load data and process jobs on a single core of the SCC as compared to the PC. The longer data loading times are due to the need for network I/O, since the data is stored on the Management Console PC (MCPC) and accessed by the SCC via NFS. In long running services, however, data is loaded once and used when needed, therefore, this time is disregarded in our performance comparisons. The differences in processing times is due to the architecture difference of the processor cores (x86-64 vs. x86) and their operating frequencies (3GHz vs. 533MHz) on the PC and a single SCC core respectively.

Table 7.4: Time required for the baseline all-to-all PSC task using the TMalign, CE and USM PSC methods on the PC and a single core of the SCC. The table also shows the time required for the all-to-all MCPSC task (where all three PSC methods are used). All times are in seconds.

Method	Ported software on PC				Ported software on SCC 1 core			
	CK34		RS119		CK34		RS119	
	Load	Processing	Load	Processing	Load	Processing	Load	Processing
TMalign	0.01	127	0.05	1725	0.30	2514	1	33452
CE	0.80	374	3	6459	14	7152	50	132205
USM	0.01	0.60	0.04	7	0.70	30	2	345
All	0.80	502	3	8191	15	9697	53	166000

7.3 Comparison with multi-core implementation

The MCPSC implementation running on the SCC with 47-slaves achieves a speedup of 42 (41) with an efficiency of 0.9 (0.86) on the RS119 (CK34) datasets as shown in Table 6.1. In order to perform this experiment the MCPSC software framework was re-targeted to the Intel Core i7 multi-core processor, using *openmp* to implement multi-threading. This experiment allowed us to assess how the MCPSC problem scales with increasing number of cores on a modern multi-core CPU readily available for scientists and engineers. This multi-core software version uses shared memory to replace the RCCE based message passing (*recv* and *send*) calls used in the many-core version developed for the Intel SCC. As shown in Table 7.5 we observe speedup on this multi-core processor when running all-to-all MCPSC with the CK34 and RS119 datasets up to the 4 threads configurations. Thereafter a steep speedup drop (efficiency loss) is observed.

Table 7.5: Speedup, Efficiency and Throughput in pairwise-PSC tasks per second for performing MCPSC on a Intel Core i7 multi-core CPU using the CK34 and RS119 datasets.

Threads	Dataset CK34			Dataset RS119		
	Speedup	Efficiency	Throughput	Speedup	Efficiency	Throughput
1	1.00	1.00	4.92	1.00	1.00	3.49
2	1.88	0.94	9.27	1.46	0.73	5.09
3	2.04	0.68	10.01	1.98	0.66	6.90
4	2.54	0.63	12.47	2.60	0.65	9.08
5	2.70	0.54	13.25	2.70	0.54	9.42
6	2.71	0.45	13.32	2.71	0.45	9.45
7	2.74	0.39	13.47	2.69	0.38	9.38
8	2.64	0.33	12.98	2.67	0.33	9.31

This is attributed to the fact that this is a quad-core processor that implements Hyper Threading (HT) [104]. Since all threads operate on exactly the same type of instructions workload (SPMD) it cannot take advantage of each core's super scalar architecture that needs varied workload placement (e.g. integer and floating point arithmetic at the same time) to show performance advantages when utilizing more threads than cores. Further, an SMP Operating System (O/S) - like the one we run on our test system - uses all system cores as resources for swapping threads in and out. A thread executing on core PE_1 for some number of cycles, may get swapped out and later resume execution on core PE_2. Thus the thread cache on core PE_1 becomes useless and it gets a lot of cache misses when restarting on PE_2. This O/S behavior impacts cache performance and reduces the overall application performance. In terms of pairwise-PSC tasks per second (pps) the Intel Core i7 achieves a throughput of 13.47 pps (9.45 pps) on the CK34 (RS119) dataset as compared to 5.53 pps (3.63 pps) achieved by the SCC. This is due to the fact that the multi-core CPU contains latest generation cores, featuring a highly superior, out-of-order micro-architecture and clocked at almost 7.5x the frequency of the SCC. We believe that even more performance can be exploited from next generation multi-core processors, should they start introducing hardware memory structures like the Message Passing Buffer (MPB) of the SCC NoC processor alongside their cores, for increased communication efficiency among them, without resorting to shared memory and its unavoidable locks or cache-coherency protocol overheads.

Finally, Table 7.6 shows the comparative performance of the SCC (with 48 cores) and the i7 (running 7 threads) on all-to-all MCPSC using several datasets. It can be seen that the i7 outperforms the SCC consistently in terms of Throughput. The two larger datasets - Lancia and Proteus - were not processed on the SCC because of the limited per-core memory (512 MB) which was not sufficient to load the domain structure data for the full

Table 7.6: Comparison of the SCC and i7 CPU in terms of throughput delivered on all-to-all MCPSC.

Dataset	Pairs	SCC Throughput	i7 Throughput	Ratio
Skolnick	1089	6.05	27.92	4.62
Chew-Kedem	1156	5.03	13.29	2.64
Fischer	4624	1.98	9.37	4.74
Rost-Sander	12996	3.30	9.38	2.85
Lancia	72361	-	222.65	-
Proteus	76729	-	6.77	-

dataset. A decrease in Throughput was observed as the size of the dataset increases with some exceptions. Both the SCC and the i7 deliver lower than expected Throughput (based on its size) on the Fischer dataset which we believe is due to the higher complexity of the dataset. The Fischer dataset has domains belonging to 5 times more SCOP Super Families as compared to CK34 and also has the second highest median length of protein domains among all the datasets used in this work. Conversely, the Throughput delivered by the i7 on the Lancia dataset is significantly higher than that delivered for the similar size (in terms of number of domains) Proteus dataset. We attribute this difference to the large difference between the mean lengths of domains in each dataset.

7.4 Qualitative analysis

The acceleration capabilities offered by modern processors (many-core and multi-core) enables performing all-to-all PSC using different methods on large scale datasets and compare results. For example, it becomes easy to perform a qualitative analysis on a set of protein domains and use consensus score (GP’S) to explore relation between the protein domains, categorize them into biologically relevant clusters [134] and automate generation of databases such as *Structural Classification of Proteins* (SCOP).

The MCPSC comparison method outperforms the component PSC methods (TMalign, CE and USM) and groups 34 representative proteins from five-fold families into biologically significant clusters. In Figure 7.3 each protein domain is using the format ”domainName_-foldFamilies”. The tags of the ”foldFamilies” belong to one of: i) tb = TIM barrel, ii) g = Globins, iii) ab = alpha beta, iv) b = all beta and v) a = all alpha protein families. The clusters obtained from the MCPSC method show that most domains are grouped correctly according to their structural fold. Figure 7.4, shows similar clusters generated by the three component PSC methods used in this work. In comparison all the component PSC methods produce clusters in which there are many wrongly grouped domains. Misclassification of domains is the highest in USM. While CE performs better, TMalign is the best performing PSC methods. However, it can be seen that the TMalign based score results in misclassification of the TIM barrel and Alpha beta containing protein domains. This is not entirely incorrect because *tb* domains are considered as a type of *ab* domains but containing a specific topological structure known as *toroid*. However, MCPSC score based clustering is able to uncover this underlying dissimilarity, which was an interesting observation from these results. The quantitative analysis evaluating the Recall vs. Precision tradeoff based on the F-measure [44], results in a value of F=0.91 for the MCPSC method, which is higher than TMalign (F=0.82), CE (F=0.71) and USM (F=0.62).

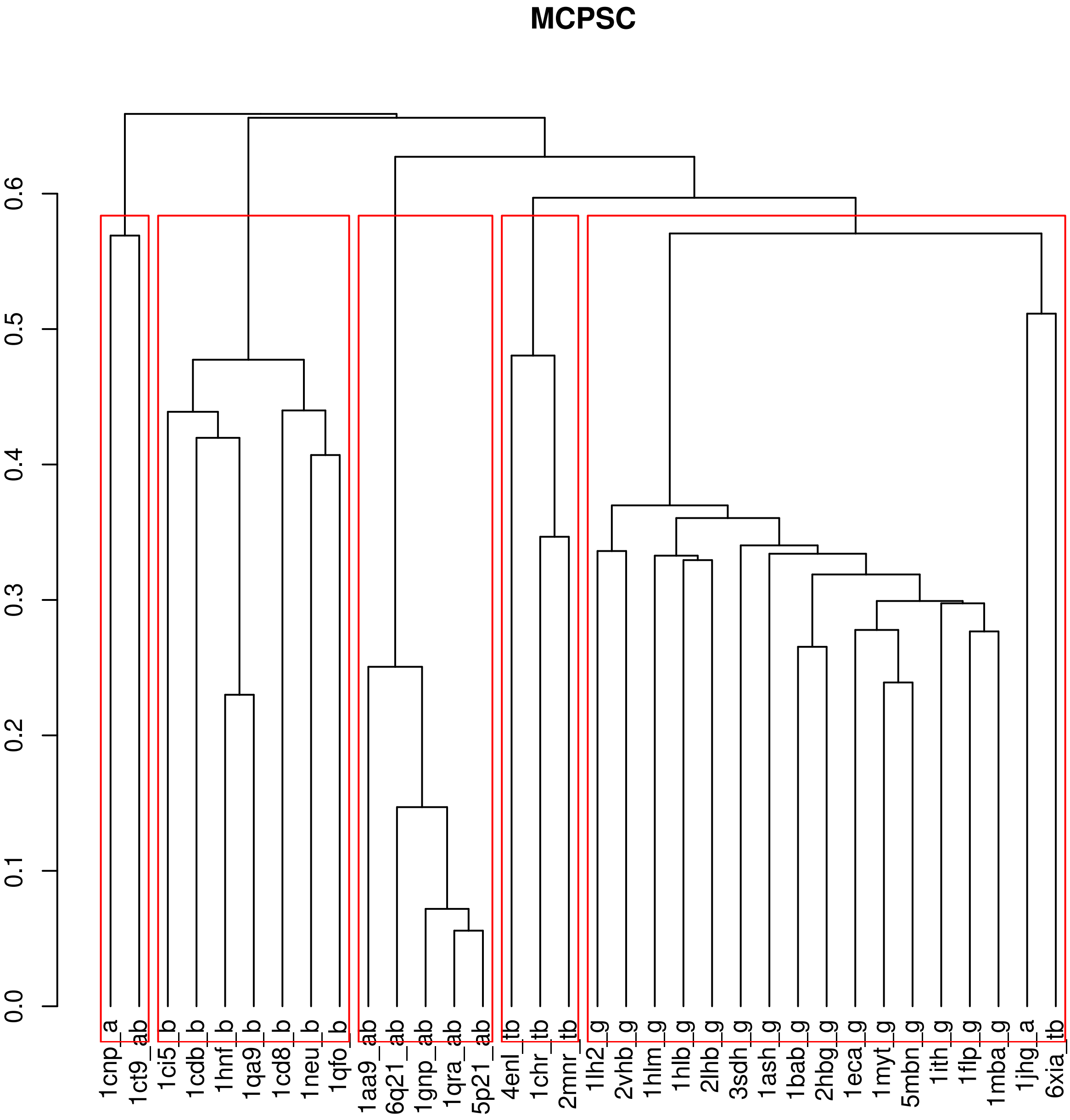
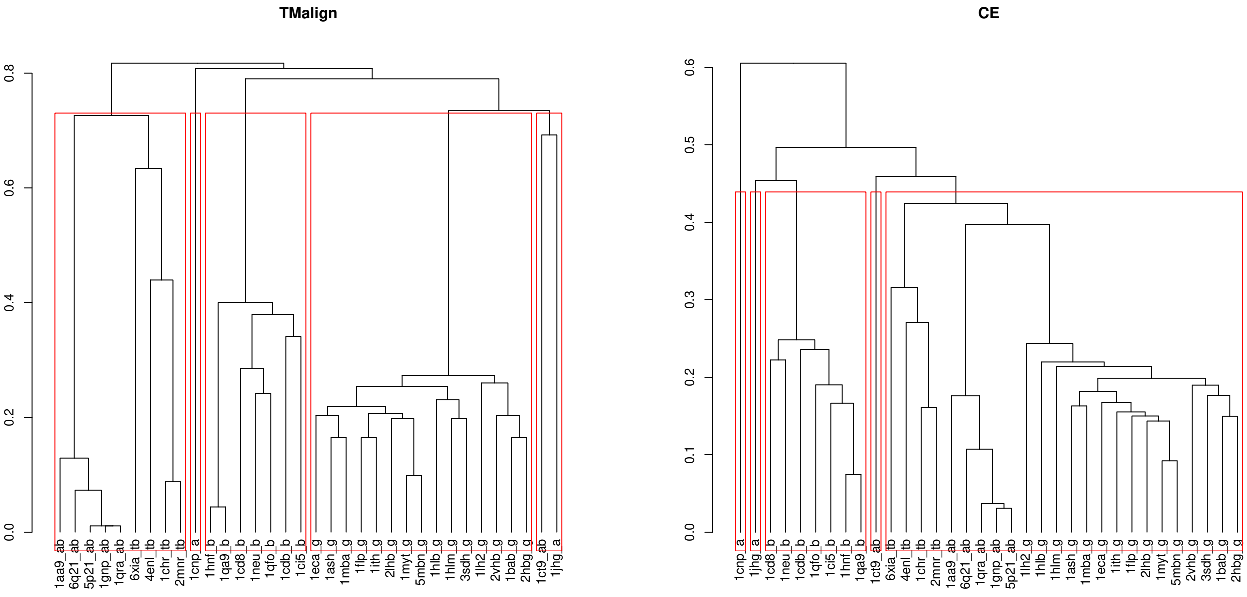
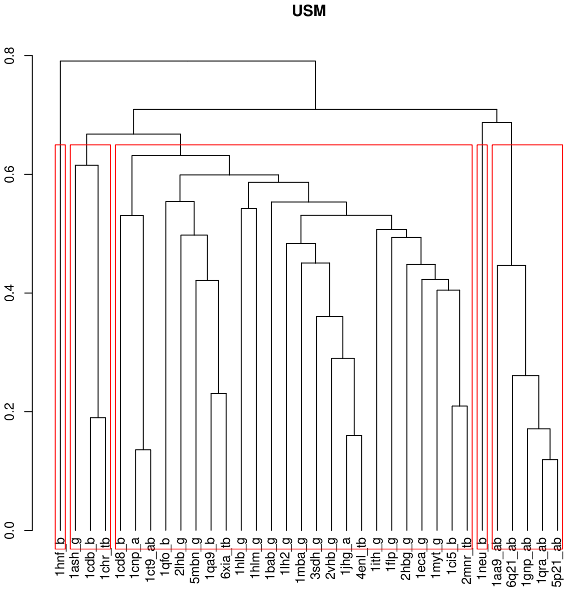


Figure 7.3: Hierarchical clustering result using the Chew-Kedem dataset and MCPSC score as distance metric between domains. Each box represents a cluster and the domains belonging to it. The Average linkage method was used to build the dendrogram.



(a) Classification with TMalign

(b) Classification with CE



(c) Classification with USM

Figure 7.4: Results of Hierarchical clustering using the Chew-Kedem dataset and the three component PSC methods.

8. LARGE SCALE CONSENSUS BASED MCPSC ON MULTI-CORE PROCESSOR ARCHITECHTURES

With the growing size of protein databases, structural analysis in real world scenario requires comparison of structures against large datasets. Tools and utilities to handle such large scale MCPSC must therefore make efficient use of commodity hardware. In this chapter we present the design of one such utility, developed during the course of our work, which allows efficient use of multi-core CPUs for large scale MCPSC. The utility is easy to extend and packaged for ease of installation and use. To the best of our knowledge this is the first utility of its kind that is made openly available to the scientific community [116].

8.1 MCPSC on commodity hardware

One-to-many, many-to-many and all-to-all PSC jobs with one or more PSC methods can be distributed in multiple ways depending on the unit of work sent to the processing elements [113]. On a multi-core machine with shared memory the cost of transferring data is negligible hence it is the ideal architecture for using fine grained job distribution. If there are N pairwise comparisons to be made and M is the number of PSC methods then the total number of fine grained PSC jobs are $N \times M$.

To this end, a useful cluster computing shared resource available to the community is the ProCKSI server [13]. Given a dataset of protein domains, it supports *all-to-all* MCPSC experiments, returning to the user individual PSC method scores as well as a consensus average score. While ProCKSI is an one-stop resource, it is limited in the size of the data that a user is allowed to submit (upto 250 protein domains). Moreover, the users cannot add new PSC or MCPSC methods of their choice. In general, distributed solutions, implemented using shared resources, suffer from limitations such as extensibility and maintainability.

We developed a utility, called *pyMCPSC* which we have created using the popular Python programming language [132] and make available to the community. *pyMCPSC* generates pairwise structure comparison and consensus scores using multiple PSC and MCPSC methods. In addition, the resulting similarity scores are used to generate multiple insightful visualizations that can help a) compare and contrast the structure comparison methods, and b) assess structural relationships in the analyzed dataset. Such comprehensive analysis allows researchers to gain quick visual insights about structural similarities existing in their protein datasets, simply by exploiting the power of multi-core CPUs of their computers.

pyMCPSC allows pairwise structure comparison tasks to be distributed over the multiple cores of the CPU and provides a simple Command Line Interface (CLI) for setting up and running all-to-all MCPSC experiments in a standard PC. Our utility wraps available executable PSC method binaries with a user specified class, thus making it easy to incorporate new PSC methods in MCPSC analysis while hiding the details of parallel job distribution from the user. As distributed today, *pyMCPSC* contains wrappers for the executable binaries of five well known PSC methods: *CE* [119], *TM-align* [143], *FAST* [147], *GRALIGN* [73] and *USM* [1]. These implementations can also serve as examples of how to quickly extend the utility with new PSC methods as soon as their binaries become available to the community. In addition, *pyMCPSC* generates consensus (MCPSC) scores using multiple (five) alternative schemes. Finally, *pyMCPSC* uses the computed similarity scores (PSC and MCPSC) to generate several insightful visualizations.

8.2 Consensus scores introduced

Given a set of protein domains, we generate similarity scores using the supported PSC methods, for all protein pairs (all-to-all) that can be formed using the dataset. One-to-many, many-to-many and all-to-all PSC jobs with one or more PSC methods can be distributed in multiple ways depending on the unit of work sent to the processing elements [113]. If there are P pairwise comparisons to be made and M PSC methods to be used, the total number of fine grained pairwise PSC jobs is $P \times M$. By default *pyMCPSC* creates a list of pairwise comparisons (P) corresponding to the all-to-all setup (all pairs of domains in the specified dataset). Pairwise similarity scores are then generated by calling third party external binaries for each of the supported PSC methods. The pairwise PSC processing is distributed over p threads (a configurable parameter), equal to the number of cores in the processor. The user may include specific pairs of interest in the ground-truth data (one-to-many or many-to-many setups) to limit the pairwise comparison results used in consensus calculations and performance analysis. Once all the pairwise similarity scores have been generated, the MCPSC consensus scores are also computed for the domain pairs. The consensus scores calculation involves several steps, as indicated in Fig 8.1.

8.2.1 Data imputation scheme

A “local average fill” scheme is used to compensate for potentially missing data for each PSC method. Missing PSC score for pairs of domains can be a result of PSC method executable or PDB file errors and can be problematic for classification/clustering analysis that rely on these values. Assuming that pairwise PSC scores were successfully generated for s domain pairs (out of the total P pairs in a dataset), the number of missing pairwise scores is $P - s$, with the value of s being different for different PSC methods. To impute the missing data for each PSC method, the following steps are repeated for all domain pairs (d_i , d_j) with a missing score:

- find the set of PSC scores where d_i is the first domain in the pair
- find the set of PSC scores where d_j is the second domain in the pair

- merge the two sets and use the mean value of scores in the set union as the PSC score for that domain pair
- if the two aforementioned sets are empty then use the global average of scores for that PSC method to supply the missing score's value.

8.2.2 PSC Scores

Base PSC scores calculation Pairwise scores for all PSC methods are first converted to dissimilarities (with value higher when domains in the pair are more different).

PSC scores scaling A Logistic Sigmoid scheme is used to scale scores to ensure equal contribution of PSC methods towards the consensus MCPSC scores calculation. Given the base dissimilarity score (X) for a PSC method, its scaled version (S) is obtained using Equation (8.1) below, where μ and σ are the mean and standard deviation respectively over all scores X for that method. Effectively, the dissimilarity scores are first autoscaled (to make the different PSC method scores comparable) and then the logistic sigmoid is applied. As a result, at the end we obtain similarity scores (S) in the range 0 to 1.

$$S = 1 - \frac{1}{1 + e^{-\frac{X-\mu}{\sigma}}} \tag{8.1}$$

8.2.3 MCPSC consensus scores calculation

We have introduced five different MCPSC consensus scores as discussed below:

- **M1** - It is the Generalized Mean of the available PSC scores and is computed as shown in Equation (8.2) below, where m is the number of non-null PSC method scores available for a given domains pair. In the current implementation $q = 1$, hence $M1$ is essentially the average of the available PSC scores for the pair.

$$M1 = (\frac{1}{m}\sum_{i=1}^{m} {S_i}^q)^{\frac{1}{q}} \tag{8.2}$$

- **M2** - It is a weighted average of the PSC scores of the different methods. For each domain pair we weight the available PSC method scores by the percentage of pairs successfully processed by each PSC method in the whole dataset (coverage based weighting).
- **M3** - Similar to **M2**, but here we also allow domain expert knowledge to play a role in the method's relative weighting e.g. we lower USM method's weight to one half since it is a domain agnostic method (domain expert knowledge based weighting).

- **M4** - For each domain pair, we weight each PSC method by the mean RMS distance of its scores from those of the other PSC method scores, as shown in Equations (8.3), (8.4) and (8.5) below, where S_k^i is the scaled PSC score for the k^{th} domain pair $(k = 1, 2, \ldots, P)$ and the i^{th} PSC method $(i = 1, 2..., \ldots M)$. If scores S_k^i or S_k^j are missing, the corresponding k^{th} term is excluded from the summation in (3) (divergence driven weighting).

$$RMSD_{ij} = \sqrt{\frac{1}{P}\sum_{k=1}^{P}(S_k^i - S_k^j)^2} \tag{8.3}$$

$$r_i = \frac{1}{M}\sum_{j=1}^{M} RMSD_{ij},\ i, j \in \{1, 2, ..., M\} \tag{8.4}$$

$$w_i = \frac{r_i}{max(r_i)}, i \in \{1, 2, ..., M\} \tag{8.5}$$

- **M5** - For each domain pair, we weight the PSC methods by user supplied relative weights.

For a domain pair with m available PSC method scores, where in general $m <= M$ ($M = 5$ currently), the m weights are first normalized to sum up to one and the consensus score (for schemes **M2**-**M5**) is then calculated as the weighted average of the available m scores. MCPSC schemes **M1**-**M4** leverage different properties of their component PSC methods, while weighting them in different ways to generate a consensus score. Finally, a *median* MCPSC score per domain pair is generated using the **M1**-**M5** scores. As *pyMCPSC*sources are made available, it is also entirely possible for the user of the utility to experiment with new consensus score generation schemes.

Supervised learning of weights

In order to learn the weights for the supported PSC methods using supervised learning a Logistic Regression scheme can be used as follows:

- Read pairwise PSC scores generated by $M = 5$ PSC methods for all domain pairs. Each pair is represented by a feature vector containing all PSC method scores.
- Each pair is categorized as belonging to Class 1 - meaning pairs from the same SCOP classification - or Class 0 - meaning pairs belonging to different SCOP classification.
- Perform 10-fold cross-validation with Unimputed (intersection) and Imputed datasets evaluating the performance of a binary (0/1) Logistic Regression (LR) model.

  - 3 metrics - Sensitivity (Recall), Specificity and Precision are recorded at each iteration (10 in all).

This cross validation procedure was repeated several times, each time using a different percentage (varied from 1% to 100%) of the full data used to learn the coefficients. The lowest percentage for which the learnt model performed well on the 3 metrics for the Proteus domains dataset was 10%.

8.3 *pyMCPSC* software design

8.3.1 Architecture

As a software architecture, *pyMCPSC* is organized into several modules called in sequence by the main entry point. An overview of the processing sequence is shown in Fig 8.1. The modules are functionally independent and the interface between them is via files. Each module receives a set of parameters, including the files used to read data and write the output results. In a typical scenario, the user sets up an experiment, using command line parameters for supplying information such as the location of protein domain structures data and ground-truth classification (if available). The ground-truth data required by *pyMCPSC* to perform the analysis steps is the SCOP/CATH [37] classification of the domains in the dataset being analysed. The information is expected to be provided to the utility in a specific format. *pyMCPSC* first generates pairwise similarity scores for all domain pairs, using the supplied PSC methods and the implemented MCPSC methods, and then generates results to facilitate structure based comparison and analysis.

pyMCPSC is organized into several modules, each one implementing a specific functionality. The main entry point of the utility drives the sequence of activities shown. Similarity scores are generated for all protein pairs using the executable binaries of the included PSC methods. Subsequently the scores are scaled, missing data (similarity scores) are imputed and consensus MCPSC scores are calculated for all domain pairs. If the user has supplied ground-truth domain classification information, then comparative analysis results are also generated based on the similarity scores. The modules where the respective functionalities are implemented are specified in parenthesis.

Supported PSC methods

The current version of *pyMCPSC* contains wrappers for five well known pairwise Protein Structure Comparison (PSC) methods: a) *CE* [119], b) *TM-align* [143], c) *FAST* [147], d) *GRALIGN* [73] and e) *USM* [1]. These implementations can serve as examples of how to quickly extend the utility with more PSC methods, as their binaries become available in the future. Download links for software corresponding to these methods are listed in Table A. USM is not included in the Table because, the program is not available for down-

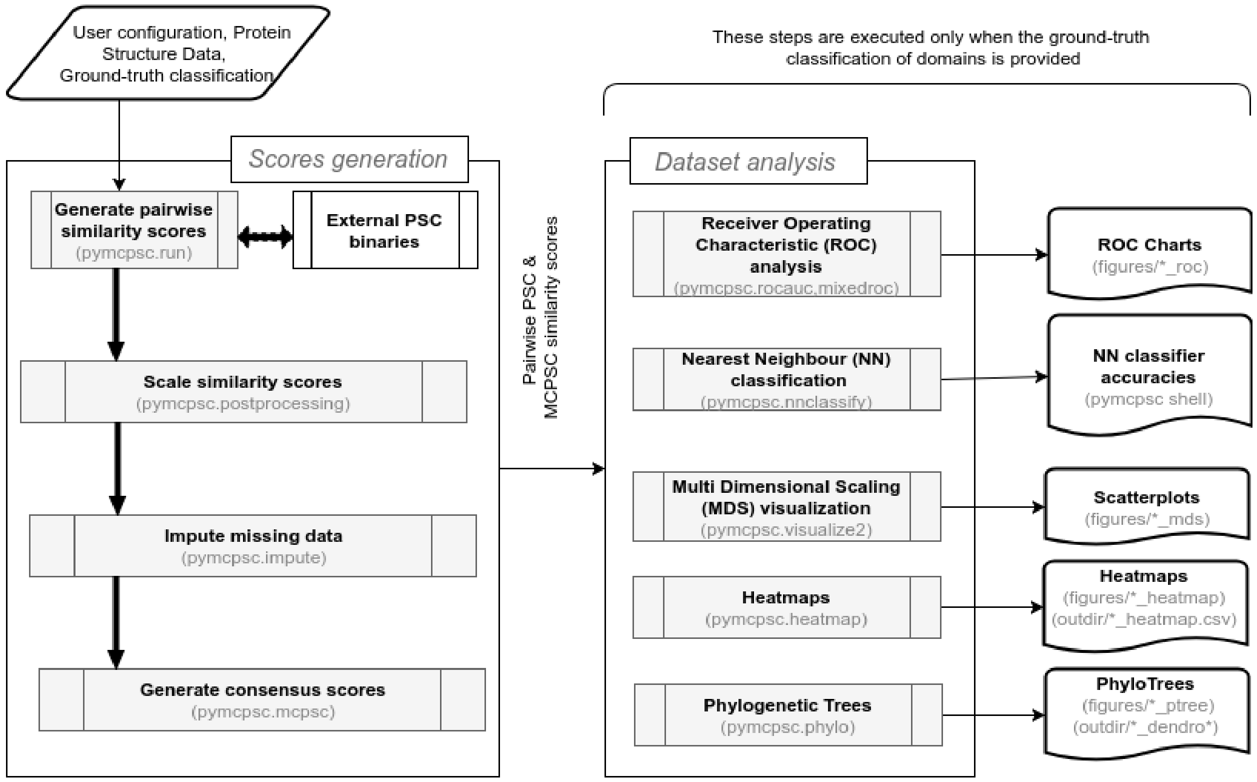

User configuration, Protein Structure Data, Ground-truth classification
These steps are executed only when the ground-truth classification of domains is provided
Scores generation
Generate pairwise similarity scores (pymcpsc.run)
External PSC binaries
Scale similarity scores (pymcpsc.postprocessing)
Impute missing data (pymcpsc.impute)
Generate consensus scores (pymcpsc.mcpsc)
Pairwise PSC & MCPSC similarity scores
Dataset analysis
Receiver Operating Characteristic (ROC) analysis (pymcpsc.rocauc,mixedroc)
Nearest Neighbour (NN) classification (pymcpsc.nnclassify)
Multi Dimensional Scaling (MDS) visualization (pymcpsc.visualize2)
Heatmaps (pymcpsc.heatmap)
Phylogenetic Trees (pymcpsc.phylo)
ROC Charts (figures/*_roc)
NN classifier accuracies (pymcpsc shell)
Scatterplots (figures/*_mds)
Heatmaps (figures/*_heatmap) (outdir/*_heatmap.csv)
PhyloTrees (figures/*_ptree) (outdir/*_dendro*)

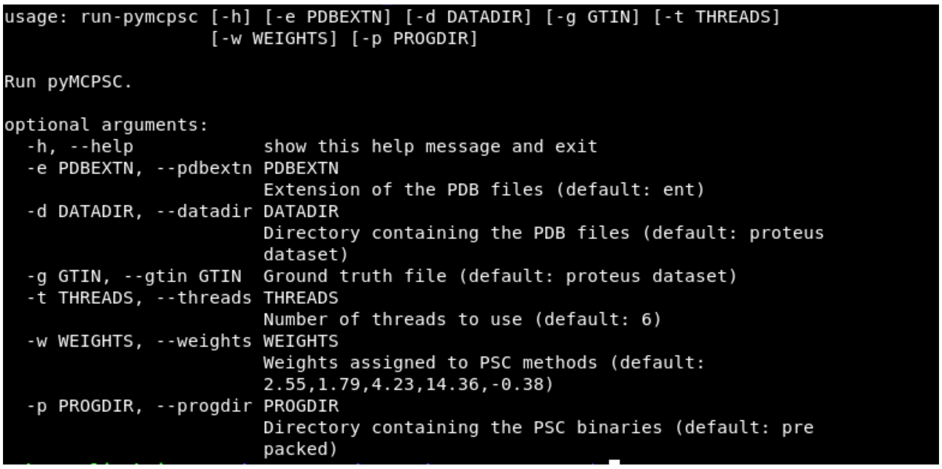
Figure 8.1: Schematic overview of the architecture of *pyMCPSC*.

load, the contact map generation binary of GRALIGN is reused and the standard Python compression libraries are used for generating the similarity scores.

Table 8.1: Download links for PSC methods used in *pyMCPSC*.

PSC Method	Download URL
CE	http://source.rcsb.org/jfatcatserver/ceHome.jsp
TM-align	http://zhanglab.ccmb.med.umich.edu/TM-align/
GRALIGN	http://www0.cs.ucl.ac.uk/staff/natasa/GR-Align/index.html
FAST	https://biowulf.bu.edu/FAST/download.htm

The user may use the *scripts/psc_get.sh* script packaged with *pyMCPSC* to download the source and binaries of the default PSC methods. Where sources are available, the script downloads and builds the sources on the user's machine. We note that sources for all the programs are not available - FAST and GRALIGN are available only in Binary formats. The script uses several standard Linux tools and requires various compilers during the process. As as a first step the script checks for availability of the dependencies and exits if a dependency is missing. It is beyond the scope of these instructions to direct the user on how the missing dependency may be resolved. The final binaries of the PSC methods are placed in the *programs* folder created where the script is executed.



```
usage: run-pymcpsc [-h] [-e PDBEXTN] [-d DATADIR] [-g GTIN] [-t THREADS]
                   [-w WEIGHTS] [-p PROGDIR]

Run pyMCPSC.

optional arguments:
  -h, --help            show this help message and exit
  -e PDBEXTN, --pdbextn PDBEXTN
                        Extension of the PDB files (default: ent)
  -d DATADIR, --datadir DATADIR
                        Directory containing the PDB files (default: proteus
                        dataset)
  -g GTIN, --gtin GTIN  Ground truth file (default: proteus dataset)
  -t THREADS, --threads THREADS
                        Number of threads to use (default: 6)
  -w WEIGHTS, --weights WEIGHTS
                        Weights assigned to PSC methods (default:
                        2.55,1.79,4.23,14.36,-0.38)
  -p PROGDIR, --progdir PROGDIR
                        Directory containing the PSC binaries (default: pre
                        packed)
```

Figure 8.2: Usage help message print out by *pyMCPSC* explaining the parameters accepted by the program.

8.3.2 Dependencies and Installation

pyMCPSC relies on extensively used scientific packages such as: Pandas [3], Scikit [90], Numpy [135], Seaborn [137], dendropy [126], Ete3 [50] and Matplotlib [51]. Binaries for the five default PSC methods are pre-packaged in *pyMCPSC*, however currently they are available only on machines running 64-Bit Linux O/S (limiting factor is GRALIGN). However, a docker container of *pyMCPSC* can be built and run on any operating system. *pyMCPSC* has been tested on Python 2 (version 2.7) and Python 3 (version 3.5).

The current implementation of *pyMCPSC* has been built and tested on a machine running 64-bit Linux. We also provide a pre-built docker image (available for download from http://bit.ly/2lRj7xD) which has been tested on multiple operating systems (including Mac OS and Windows). Detailed build and usage instructions can be found in the documentation of *pyMCPSC* (available for download from http://bit.ly/2xdP21j). The documentation can be generated from source on any system with the Make toolchain and Sphinx setup. The program parameters can be specified on the CLI (Figure 8.2). *pyMCPSC* provides a set of sensible default fallback values for the optional arguments. Descriptions and default values for all CLI arguments are provided in *pyMCPSC* documentation. If no values are specified by the user *pyMCPSC* runs the experiment described in this paper. Results (including figures) generated by *pyMCPSC* are placed in directories located in the current working directory (CWD), i.e. the one from where the program is launched.

8.3.3 Extending *pyMCPSC*

The functionality of *pyMCPSC* can be extended by including more PSC methods in the analysis as described below. In order to introduce a new PSC method into the processing pipeline, a new *Class* must be added to the utility. The *Classes* provide functionality to run the external PSC binary and provide a text level interface, between the utility and the PSC binary, to read the output generated by the PSC method. Once this key implementation has been written, minor edits to other sections of the code are sufficient to include the method into the processing. The implementations of the 5 PSC wrapper classes included in *pyMCPSC* by default provide examples of how a new wrapper class may be written. It must be noted that the *pyMCPSC* does not implement PSC methods itself, but rather expects to be provided with executable binaries (one per component PSC method participating in the MCPSC scores calculation). A wrapper class must be written to allow the external binary to be usable in *pyMCPSC*.

Template for adding new PSC method wrappers

Extending *pyMCPSC* to incorporate a new PSC method requires implementing a class that follows the template shown in Listing 1 below. The key aspects of the template are: a) the path to the binary must be passed at instantiation time, and b) the results of pairwise comparison must be returned for each pair so that they can be used in *pyMCPSC*. Examples of implementing this template for a PSC method can be found in the file *run.py* in *pyMCPSC* sources.

The following steps must be followed in order to introduce a new PSC method to *pyMCPSC* and make its processing results for available for the consensus scores calculation:

- Implement a class based on the template to handle the input and output of the PSC method

- Instantiate the class and pass it to the doMulti method defined in *run.py* (see line 476 for an example)

- Add the method entry to the name lists *psc_methods, psc_method_names* defined in *run_mcpsc.py*

- Add the output file generated by the execution of the PSC method to the *infiles* list defined in the file *postprocessing.py* and update the column names for the output file generated by the module

- Add the method weight for consensus scores calculation by updating the value of *__def_WEIGHTS__* in *run_pymcpsc.py*

```
class PSC_HANDLER:
  def __init__(self, path_to_binary):
    # store path in class instance and
    # perform any additional house keeping

  def process_pair(self, domain1, domain2):
    # execute external binary with domains
    ...
    # read execution output
    ...
    # collect results of pairwise processing
    ...
    return results
```

Listing 8.1: Template of *Class* that needs to be added to our command-line utility to introduce processing for a new PSC method. An instance of this class can then be passed to a thread pool for distributed processing of a list of pairwise PSC jobs.

9. PERFORMANCE BENCHMARKING: CONSENSUS MCPSC VS COMPONENT PSC METHODS

In this chapter we report the results of benchmarking the consensus MCPSC methods performance as compared to its component PSC methods. First, we present the performance of *pyMCPSC* on a multi-core processor and then present the results of qualitative analysis of consensus MCPSC on the Proteus 300 dataset. Lastly, we present the results of MCPSC analysis of the very large SCOPCATH dataset.

We will demonstrate the use of *pyMCPSC* using protein pairs obtained from the Proteus dataset [6]. PSC scores were obtained for these pairs and analyzed as discussed in the paper. The number of pairwise PSC jobs processed per PSC method is actually one half of this value because of the symmetry of the PSC scores matrix, however the post processing and performance calculations are performed with the full matrix. The PDB files, the ground-truth SCOP classification and the pairwise domain list as well as the experimental setup are included in the *test* folder of the downloadable sources. *pyMCPSC* generates performance results for three sets of domain pairs, defined as follows:

- Original Dataset: It includes the similarity scores for the domain pairs defined in the original dataset, but with missing values. The number of missing values may vary depending on the PSC method as explained above.
- Common Subset: It consists of the subset of domain pairs taken from the Original dataset for which scores have been generated by all PSC methods. In the case of the Proteus dataset, this corresponds to 27312 domain pairs, which is less than half of the total number of pairs processed.
- Imputed Dataset: It consists of the Original dataset with the missing scores filled using data imputation The total number of domain pairs for the Proteus dataset is $P = 72630$.

9.1 Performing MCPSC on a multi-core processor

Using *pyMCPSC* we generated pairwise similarity scores (all-to-all) based on the 5 PSC methods and the 5 MCPSC schemes (M1 - M5) included in the utility by default, as well as the pairwise median MCPSC scheme. Experiments were carried out using multi-threaded processing on an Intel Core i7- 5960X “Haswel” 8-Core (16 Threads) CPU running at 3.0 GHz with 32 GB of RAM and an SSD running Linux. The Core i7 CPU features highly optimized out-of-order execution and HT (Hyper Threading), Intel’s flavor of Simultaneous Multi-Threading (SMT).

The number of domain pairs for which scores were successfully generated varies among the PSC methods (Table 9.1), with GRALIGN and FAST having the lowest coverage. This is attributed to differences between the build and runtime environments, the thresholds built into the PSC method programs and errors in the structure files downloaded from the PDB. A speedup factor of 9.13 is achieved for end-to-end processing of the Proteus 300 dataset using *pyMCPSC* when $p = 16$ threads are used (Table 2).

Table 9.1 provides the number and percentage of pairs (coverage) successfully processed by each PSC method. Table 9.2 shows the time *pyMCPSC* needs to process the pairwise PSC tasks for the Proteus_300 dataset when using an increasing number of threads (from 1 to 16). GRALIGN is not run in parallel by *pyMCPSC* because its binary is already optimized to use all the available cores of the CPU. The table shows the time taken by the five PSC methods and the consensus scores calculation (Scale similarity scores, Impute missing data, Generate consensus scores). In addition to the end-to-end computation time we also provide in Table 9.2 the total times for the Scores Generation and the Dataset Analysis blocks. We believe that the superlinear speedup observed in parallel pairwise PSC processing is due to PDB structure data caching which allows multi-threaded runs reuse the cached files.

Table 9.1: PSC methods coverage for the Proteus dataset.

PSC Method	# Domain pairs processed	Coverage
CE [119]	64964	89%
FAST [147]	39604	55%
GRALIGN [73]	56406	78%
TM-align [143]	72630	100%
USM [1]	72630	100%

Table 9.2: Time (in seconds) and Speedup (S) for end-to-end all-to-all analysis of the Proteus_300 dataset using *pyMCPSC*on a multi-core PC with Intel i7 CPU having 8 cores (16 threads), 32 GB RAM, running at 3.0 GHz, under Ubuntu 14.04 Linux. GRALIGN already uses all the CPU cores by default.

	1 Thread	**4 Threads**		**8 Threads**		**12 Threads**		**16 Threads**	
	Time	**Time**	**S**	**Time**	**S**	**Time**	**S**	**Time**	**S**
Pairwise scores generation									
GRALIGN	86	86	1.00	86	1.00	86	1.00	86	1.00
USM	139	46	3.02	20	6.95	17	8.18	15	9.27
FAST	4100	1035	3.96	412	9.95	329	12.46	313	13.10
TM-align	3601	1032	3.49	423	8.51	333	10.81	299	12.04
CE	16776	4022	4.17	1858	9.03	1420	11.81	1213	13.83
Consensus scores	28	28	1.00	28	1.00	28	1.00	28	1.00
Block level									
Scores Generation	24730	6249	3.96	2827	8.75	2213	11.17	1954	12.66
Dataset Analysis	843	849	0.99	844	1.00	846	1.00	847	1.00
End to End									
End to End	25573	7098	3.60	3671	6.97	3059	8.36	2801	9.13

In Figure 9.1 we show the speedup factor achieved and the total processing time as the number of threads increases from 1 to 16. Nearly linear speedup is observed till the number of threads reaches the number of available cores of the CPU (8). The speedup continues to grow with the number of cores even beyond that point, albeit at a slower rate. This analysis suggests that the emerging many-core processors with more than 16 cores could also be exploited by *pyMCPSC*to analyze very large datasets.

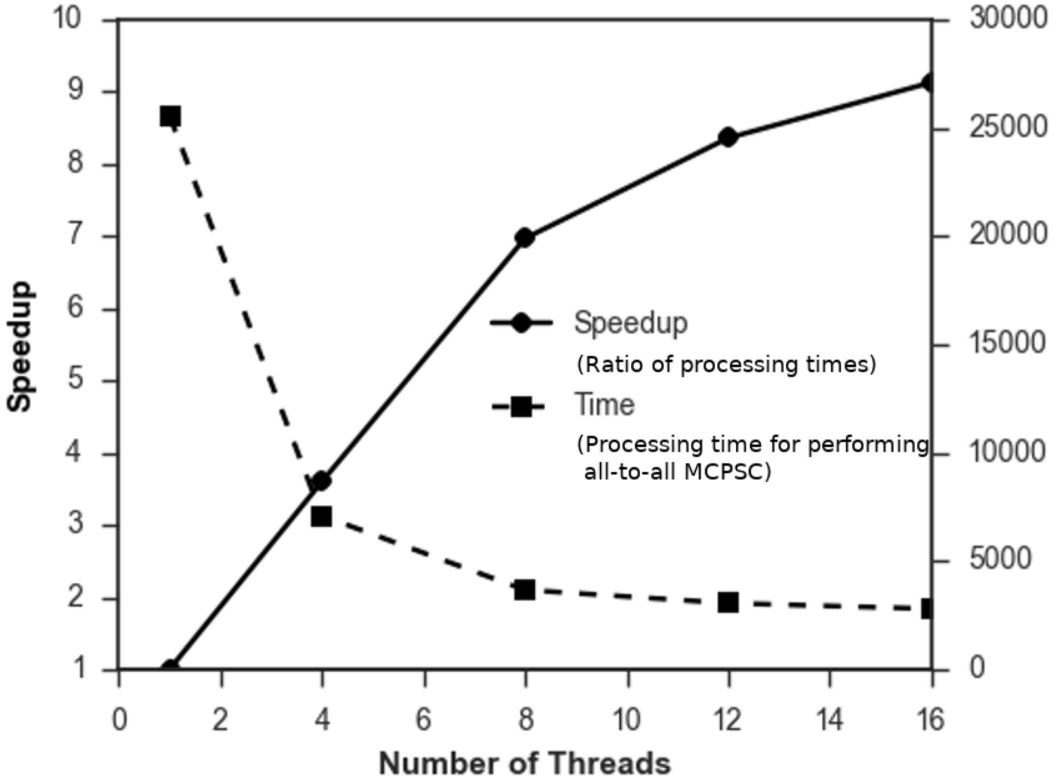


Figure 9.1: Speedup factor and total processing time for performing all-to-all MCPSC with increasing number of threads on a Intel Core i7 multicore CPU using the Proteus 300 dataset.

9.2 MCPSC provides quality consensus scores

Receiver Operating Characteristics (ROC) analysis can be used to compare the classification performance of MCPSC with that of the component PSC methods. *pyMCPSC*uses ROCs and corresponding Area Under the Curve (AUC) values for performance benchmarking if ground truth data is available.

The following procedure is used to create the ROC curves: a) Vary a similarity threshold from 1 down to 0, moving from maximum to minimum similarity; b) For each threshold value record the number of True Positives (TP), False Positives (FP), False Negatives (FN) and True Negatives (TN). In this context, TPs (FPs) are domain pairs with similarity scores greater than the set threshold in which the two domains in the pair have the same (different) classification at the SCOP hierarchy [67] level considered respectively. Similarly, FNs (TNs) are domain pairs with similarity score less than the threshold having same (different) domain classifications respectively. Having calculated the TPs, FPs, FNs and TNs for a threshold value, we can compute the True Positive Rate and False Negative Rate values as shown in [96].

In Figure 9.2 (a), we see that for this dataset TM-align achieves the highest AUC among the five supported PSC methods. Moreover using the median MCPSC score matches or exceeds the AUC performance of the best component method. This actually remains the case even if we remove TM-align from the pool of the PSC methods and repeat the same analysis with the four remaining methods Figure 9.2 (b). In reality, we do not expect to know which PSC method will perform the best for any given dataset. So as the results suggest, combining PSC methods to obtain MCPSC scores and then using their median as the final consensus score to assess similarities is an effective strategy.

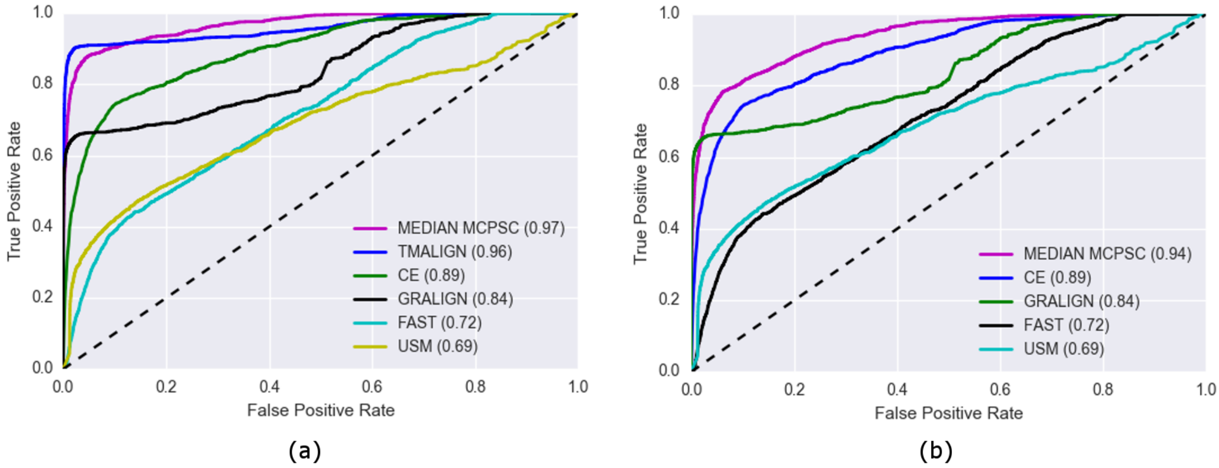
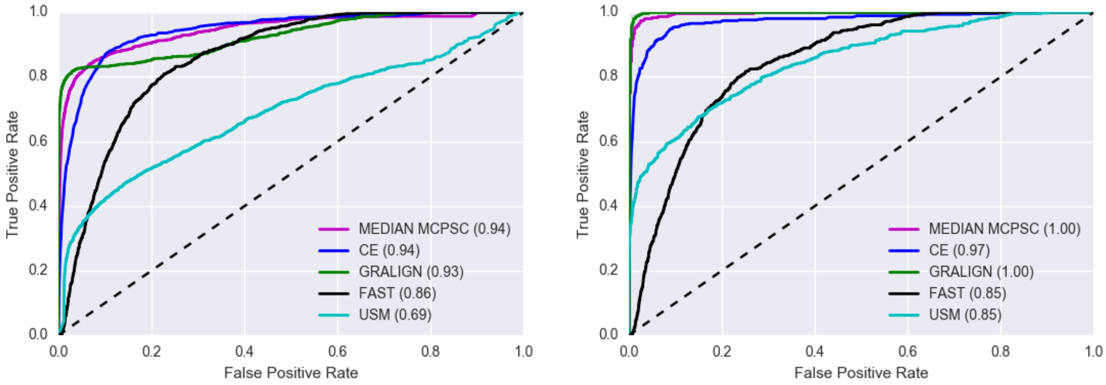


Figure 9.2: ROC curves of all PSC methods and the median MCPSC method using the Imputed dataset of pairwise similarity scores. The ROCs are generated at the SCOP Superfamily level (Level 3). Panel (a) shows the results with all five PSC methods and panel (b) with TM-align excluded.

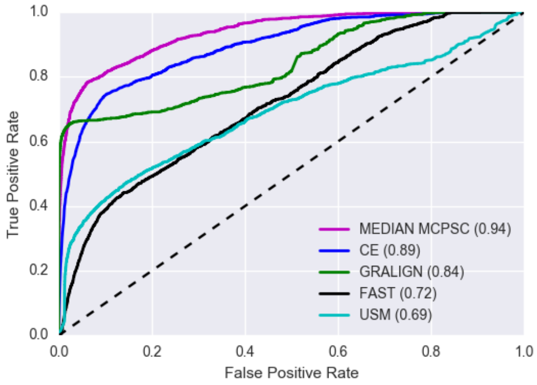
In Figure 9.3 we provide the ROC curves and corresponding AUC values for all PSC and the Median MCPSC method for all three datasets of domain pairs as defined previoysly. In general, the closer a ROC curve follows the left-side border and then the top border of the ROC space, the better the classification performance of the corresponding PSC method, because it indicates that high PSC scores (i.e. high similarity) are assigned to domain pairs where both domains belong to the same SCOP Superfamily (Scop Level 3). The results show that for this dataset TM-align happens to be the best performing PSC method. However, the median MCPSC matches or exceeds the AUC of the best component PSC method in all cases.

9.3 MCPSC consensus scores can be used to accurately classify query domains

Nearest-neighbor (NN) auto-classification [32] can be used to assess how well PSC methods can classify a query domain, given pairwise PSC scores and the structural classification of other domains. When a new protein structure is determined, it is typically compared with the structures of proteins with known SCOP classifications. Therefore, the accuracy of the PSC and MCPSC based NN-classifiers effectively reflects their ability to be used for automatic protein domain classification.



(a) Original Dataset **(b) Common Dataset**



(c) Imputed Dataset

Figure 9.3: Median MCPSC matches or exceeds the best performing method (CE) among the remaining four component PSC methods after removing TM-align from the pool used to derive the MCPSC consensus scores. The ROCs are generated at the SCOP Superfamily level (Level 3).

Distance matrices based on the PSC and MCPSC scores are used by *pyMCPSC*to perform NN domain classification. The following process is repeated for each supported PSC and MCPSC method:

- Each domain is considered as a query and assigned the class label of its Nearest-neighbor using the pairwise scores as distances. This leave-one-out class label assignment is repeated for every domain and the predicted classes are recorded.
- The percentage of domains correctly classified is then calculated.
- A domain is correctly classified if the predicted and actual (ground truth) class labels match.

MCPSC based NN-classification matches or exceeds the performance of the best PSC method at all SCOP hierarchy levels, with and without data imputation Table 9.3. Moreover, whereas the classification performance of the five supported PSC component methods varies considerably for the same SCOP level, the performance of the five different

MCPSC methods is consistent. This suggests that using *pyMCPSC* to implement different MCPSC methods and then using their median score in conjunction with NN classification can provide trustworthy query domain auto-classification results. These results also highlight that in the absence of ground truth information and/or lack of prior knowledge as to the best PSC method for a dataset, MCPSC can be employed to accurately auto-classify new domains. For more details see Appendix 10.2.

Table 9.3: Fraction of domains correctly classified at different SCOP hierarchy levels using a Nearest-Neighbor classifier built with similarity scores produced by different PSC and MCPSC methods. In the SCOP hierarchy: Level 1 = Class, Level 2 = Fold, Level 3 = Superfamily and Level 4 = Family.

	Original dataset				**Common subset**				**Imputed dataset**			
SCOP Level	1	2	3	4	1	2	3	4	1	2	3	4
TM-align	1.00	1.00	0.99	0.99	0.74	0.57	0.57	0.57	1.00	1.00	0.99	0.99
CE	0.78	0.61	0.61	0.60	0.63	0.47	0.47	0.47	0.76	0.60	0.60	0.58
GRALIGN	1.00	1.00	1.00	1.00	0.74	0.57	0.57	0.57	0.89	0.89	0.89	0.88
FAST	0.20	0.08	0.08	0.08	0.19	0.07	0.07	0.07	0.20	0.08	0.08	0.08
USM	0.84	0.72	0.67	0.65	0.65	0.51	0.49	0.49	0.84	0.72	0.67	0.65
M1	0.99	0.98	0.98	0.98	0.73	0.57	0.56	0.56	0.99	0.99	0.98	0.98
M2	0.99	0.98	0.98	0.98	0.75	0.57	0.56	0.56	0.99	0.98	0.97	0.97
M3	1.00	1.00	1.00	1.00	0.74	0.57	0.57	0.57	1.00	1.00	1.00	1.00
M4	0.99	0.99	0.99	0.99	0.72	0.57	0.57	0.57	0.99	0.99	0.99	0.99
M5	1.00	1.00	1.00	1.00	0.74	0.57	0.57	0.57	1.00	1.00	1.00	1.00
Median MCPSC	0.99	0.99	0.99	0.99	0.74	0.57	0.57	0.57	0.99	0.99	0.99	0.99

The results show that the best MCPSC method matches the performance of the best component method and the Median MCPCS based classification is almost always optimal, which makes median MCPSC a good choice for classifying query domains when prior knowledge about the best PSC method is not available. Moreover, the performance differences of the MCPSC methods are minor, suggesting that they are all quite robust to significant variations on the performance of their component PSC methods. The lower performance observed for all methods on the Common subset is probably a result of the small percentage of domain pairs for which similarity scores are available by all methods (less than 50%).

9.4 MCPSC reveals structural relations between domains

pyMCPSC uses PSC/MCPSC based distance matrices in conjunction with Multi-Dimensional Scaling (MDS) [33] to generate insightful scatterplots of protein domain organization in the structural space. An $N \times N$, distance matrix D is constructed, with N being the number of unique domains in the dataset. Matrix element D_{ij} corresponds to 1 - S_{ij}, the pairwise scaled dissimilarity score of domains d_i and d_j, where $i, j \leq N$, are drawn from the imputed data set. Missing values ($N^2 - P$) are set to 1. The value of 1 (max dissimilarity) is selected so that all domains appearing close in the visualization are in fact close to each other based on the selected method's score.

pyMCPSC uses matrix D as the basis for MDS to produce scatterplots of domains. This effectively assigns a 2-Dimensional coordinate to each protein domain constrained by the pairwise domain distances specified in matrix D. The resulting scatterplots can be used to visually explore a domains dataset, revealing existing correlations. Figure 9.4 shows the layout of the domains of the imputed dataset in 2-D space resulting from MDS using the median MCPSC scores. Such a visualization produced by *pyMCPSC* suggests that for the given dataset the SCOP Class C domains (red color) exhibit higher interdomain similarity. This is in stark contrast to the domains of SCOP Class D (cyan color) which are diffused across the scatterplot.

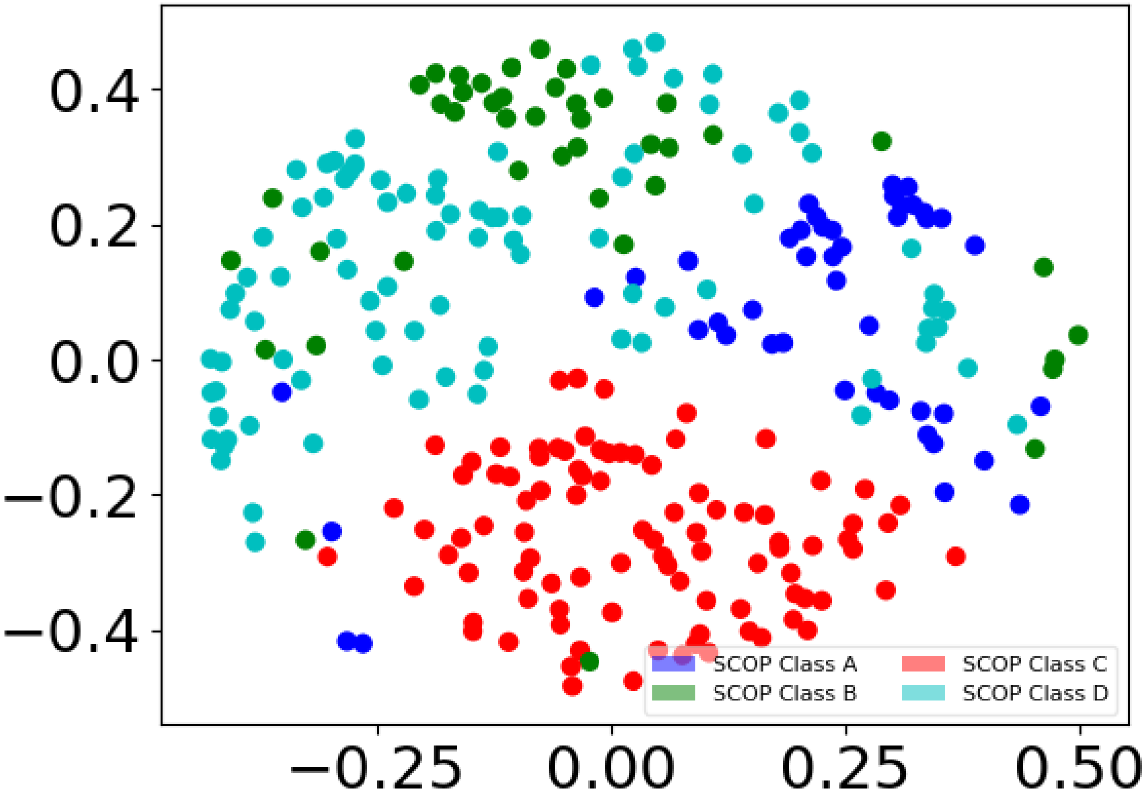
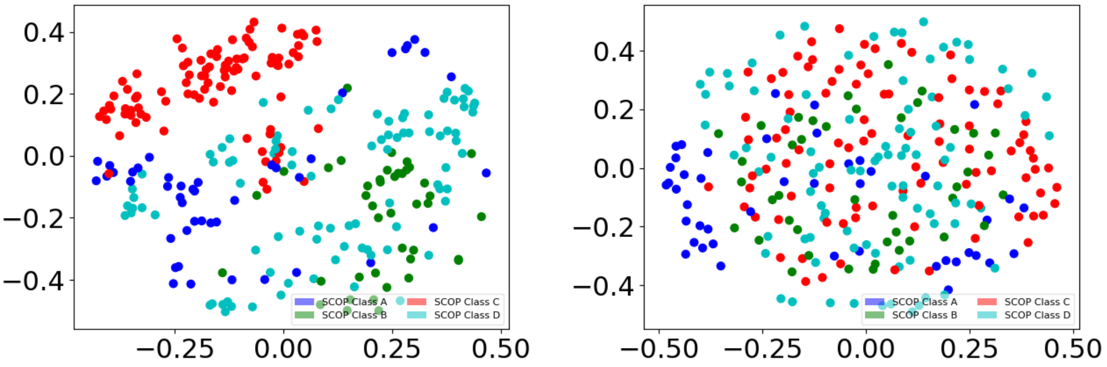


Figure 9.4: MDS scatter plot based on median MCPSC scores. Domains are colored according to their SCOP class (Level 1).

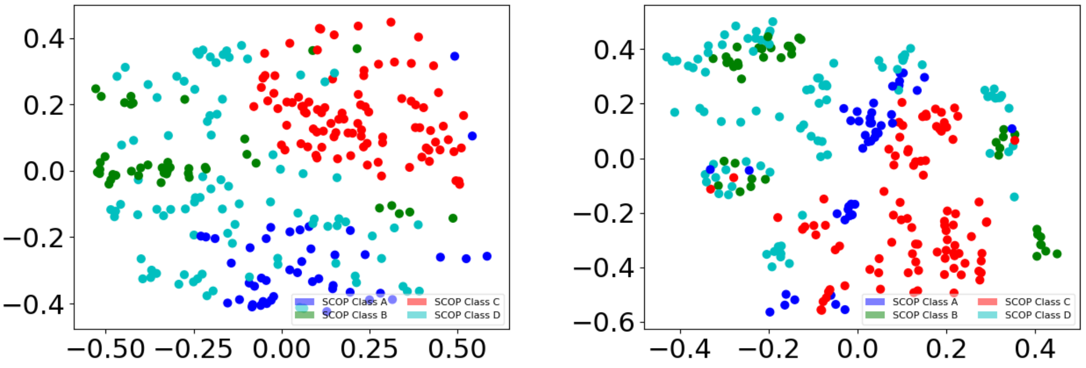
In Figure 9.5 we provide the Multi-dimensional Scaling based scatterplot visualizations generated using distance matrices based on the scaled dissimilarity scores of the different PSC methods. The scatter plots of the PSC methods differ significantly.

Heatmaps provide further evidence for the grouping of Class domains observed in the MDS based scatter plots. *pyMCPSC* uses similarity score based distance matrices in conjunction with Heatmaps. An $N \times N$ matrix, **S**, is generated where $\mathbf{S}_{ij}$ is the pairwise similarity score of domains d_i and d_j, $i, j \leq N$, using the imputed data set. This similarity matrix is used to generate the Domain level heatmaps as shown in Figure 9.6 for median MCPSC scores. Similarly, a $T \times T$ matrix, **F** can be constructed for a selected method,



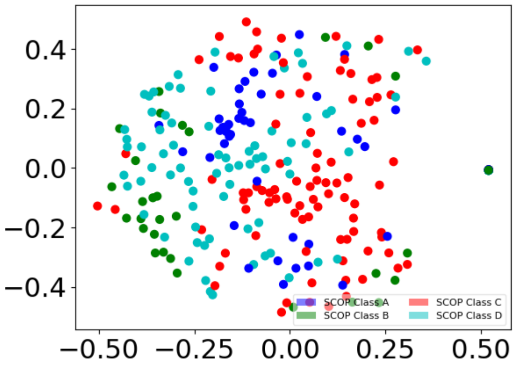
(a) CE

(b) FAST



(c) TM-align

(d) GRALIGN



(e) USM

Figure 9.5: MDS scatterplots for the five PSC methods generated using distance matrices and the imputed dataset. The points are colored by the ground-truth SCOP Level 1 classification of each domain. Blue = SCOP Class A, Green = SCOP Class B, Red = SCOP Class C and Cyan = SCOP Class D.

with T being the number of unique SCOP Folds in the dataset. Element $\mathbf{F}_{ij}$ of this matrix corresponds to the mean pairwise scaled similarity score of domains belonging to folds t_i and t_j, where $i, j \leq T$. Each element of this matrix therefore effectively represents the similarity between folds as an average of the similarity between their domains. This similarity matrix is used to generate the Fold level heatmaps as shown in Figure 9.7 for median MCPSC scores. *pyMCPSC* saves the heatmap matrices to the *outdir* as CSV files allowing the visualization to be generated using other third-party tools if needed. When the size of the matrices **S** or **F** is larger than 300×300, *pyMCPSC* disables image generation to avoid potentially creating huge files.

9.5 MCPSC can reveal functional relations between protein domains

pyMCPSC uses similarity score based distance matrices (D) in “Phylogenetic Trees” [85] to provide functional grouping of domains. *pyMCPSC* uses a Neighbor-joining algorithm from *dendropy* [126] to create dendrograms and uses them to generate unrooted circular layout “Phylogenetic Trees”. The goal is to create trees where the domains are separated into clades based on their function [11].

In Figure 1 we have marked two groups of domains belonging to different clades in the tree. The most common keyword for Group 1 is ‘GTP-Binding’ while for Group 2 it is ‘Phosphoprotein’ (see details in Appendix 10.2). The clades of the Phylogenetic Tree generated by *pyMCPSC* could therefore be used by a researcher to identify groups of domains (within the same SCOP class as in this example) that are functionally different.

9.6 *pyMCPSC* can handle very large datasets

9.6.1 Dataset

The Gold-standard benchmark dataset introduced in [35] was used in this work. The dataset contains protein domains that are consistently defined in both SCOP v1.75 and CATH v3.2.0 (i.e. with domain overlap greater than 80%) and that share less than 50% of sequence identity. Further, the benchmark only considers domain pairs that are consistently classified across the SCOP fold classification and the CATH topology classification. The dataset consists of $N =$ 6759 unique domains and defines $P =$ 3,213,631 domain pairs (similar and non-similar sets combined) [35]. Further, the 6759 domains are classified into 11 (4) Classes, 792 (780) Folds and 1348 (1550) Superfamilies according to the SCOP (CATH) classification databases respectively.

9.6.2 ROC Analysis

Figure 9.9 shows the Receiver Operating Characteristic (ROC) curves [40] for the PSC methods and the median MCPSC taken over the entire dataset of domain pairs (3,213,631). It can be seen that the Median MCPSC performs as well its component PSC methods in the scenario where PSC scores from all methods are available for all domain pairs. In gen-

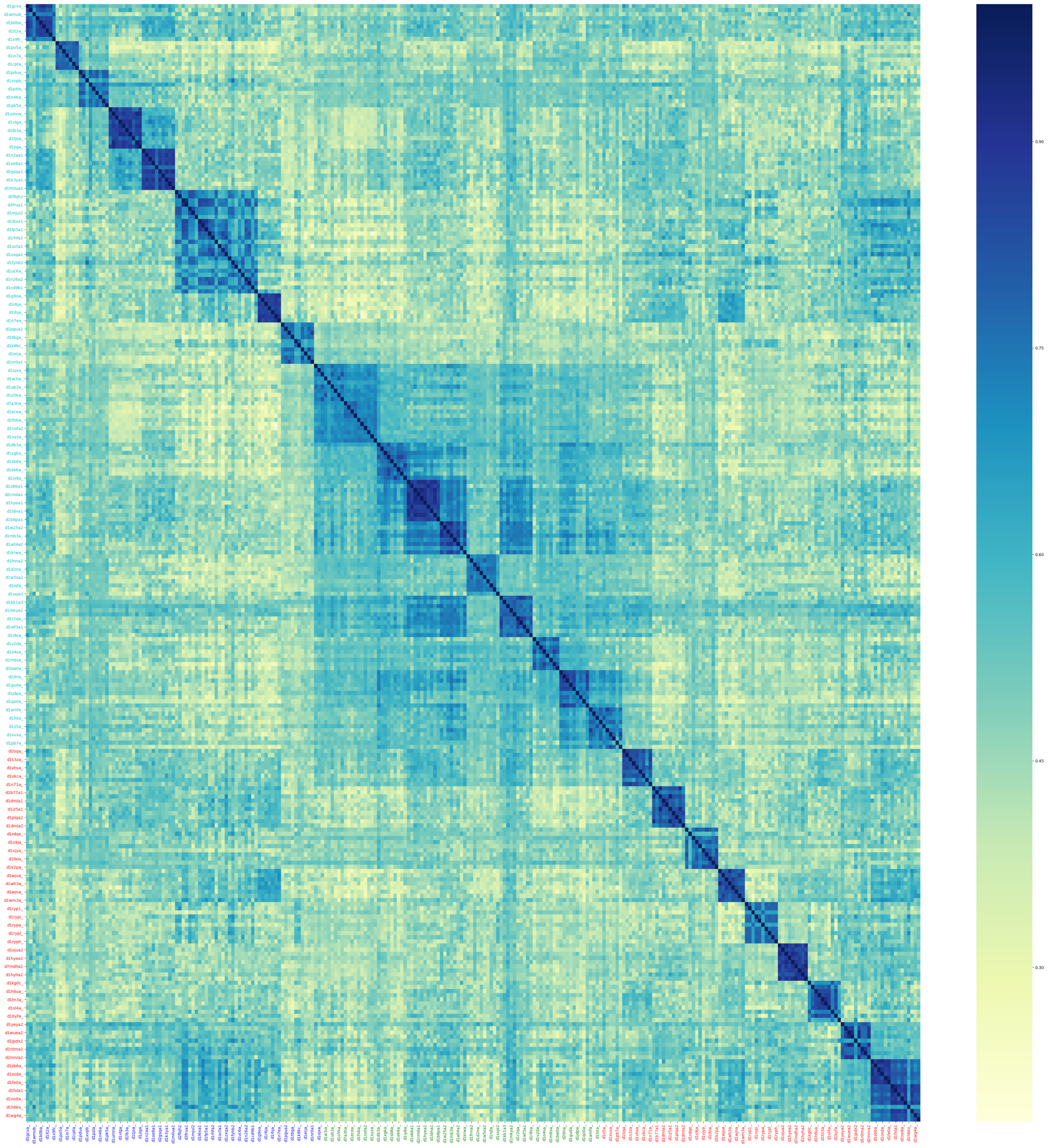


Figure 9.6: Heatmaps generated for the median MCPSC method using similarity matrices at the Domain level for the imputed dataset. The domains are colored according to their ground-truth SCOP classification. Blue = SCOP Class A, Green = SCOP Class B, Red = SCOP Class C and Cyan = SCOP Class D. Heatmaps reveal presence of sub-clusters of domains within each SCOP class evidenced by darker regions of varying sizes along the main diagonal.

eral, the closer the curve follows the left-hand border and then the top border of the ROC space, the better the performance of the PSC method because it indicates that low PSC scores (i.e. low dissimilarity) are assigned to domain pairs where both domains belong to the same CATH class.

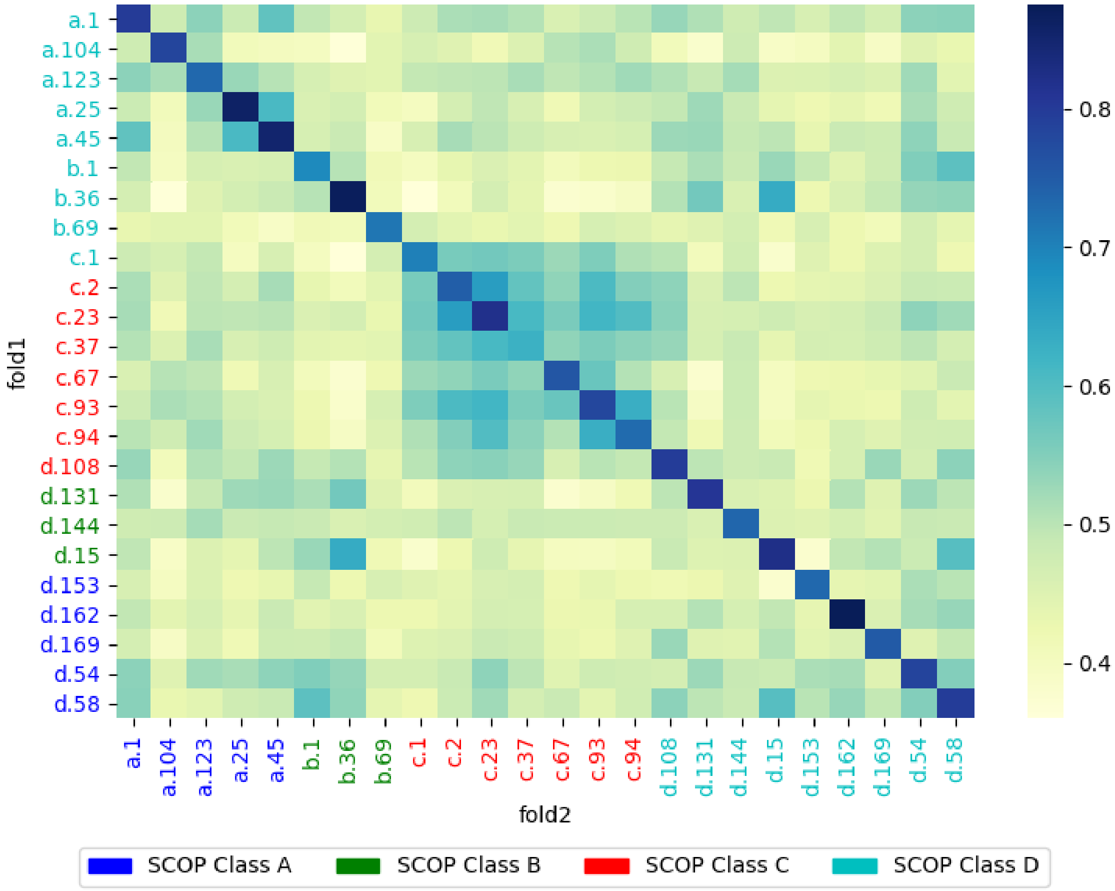


Figure 9.7: Heatmaps generated for the median MCPSC method using similarity matrices at the Fold level for the imputed dataset. The folds are colored according to the ground-truth SCOP classification. Blue = SCOP Class ‘a’, Green = SCOP Class ‘b’, Red = SCOP Class ‘c’ and Cyan = SCOP Class ‘d’. Heatmaps reveal presence of sub-clusters of folds within each SCOP class especially for SCOP Class ‘c’.

9.6.3 Nearest-Neighbor classification

The performance of Nearest-Neighbor classifiers built using the pairwise similarity scores are summarized in Table 9.4. The table shows the performance of the Nearest-Neighbor (NN) classification with the three datasets. The results show that the best MCPSC method (M5) matches the performance of the best component method. This is likely because in M5 (user defined weights) the individual PSC method weights were assigned by supervised training. As suggested also by the ROC curves, Median MCPSC based classification is performing consistently very well, which makes median MCPSC a good choice for classifying query domains when prior knowledge about the best performing PSC method is not available, as usually the case. It must be noted that the Median MCPSC classification performance is not the median of the five MCPSC methods, but rather the performance of the NN-classifier built using the median of the MCPSC pairwise similarity scores.

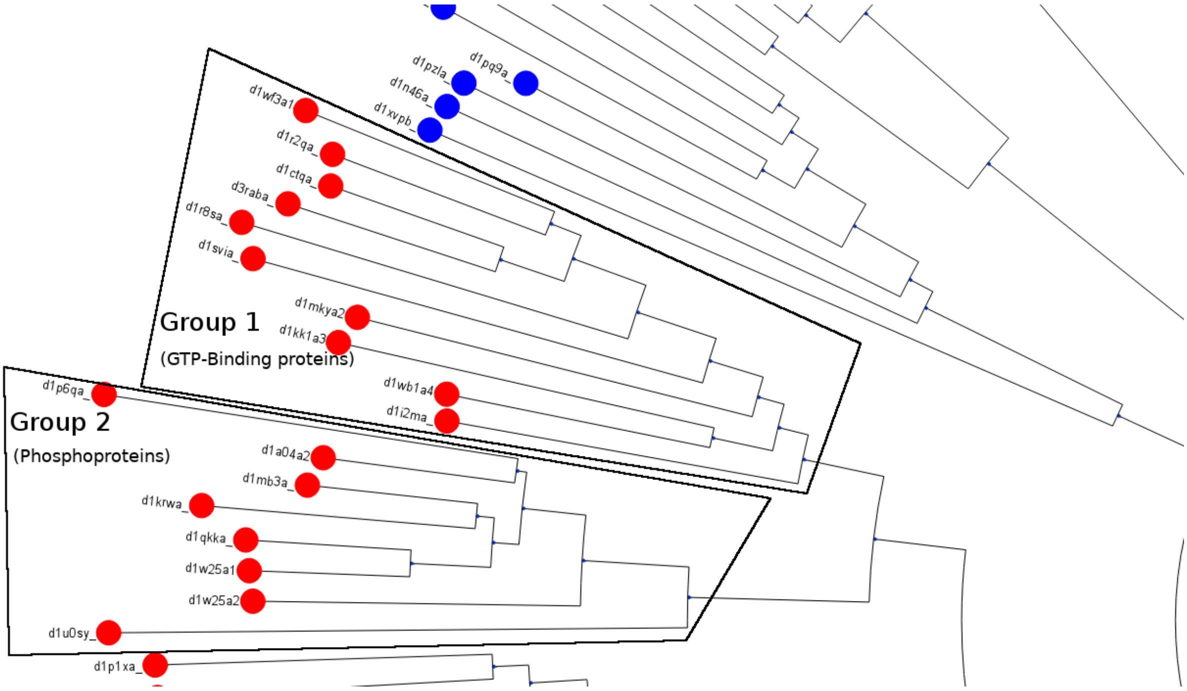
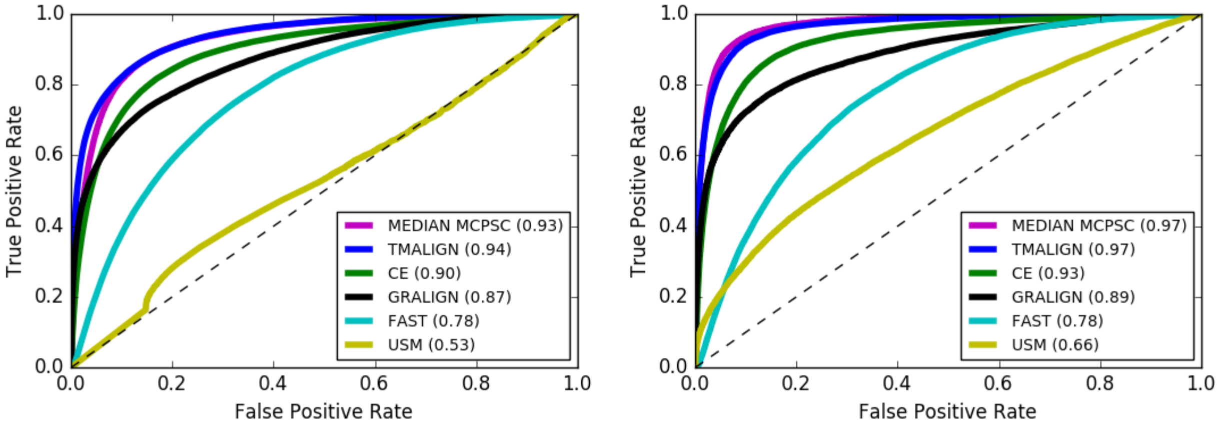
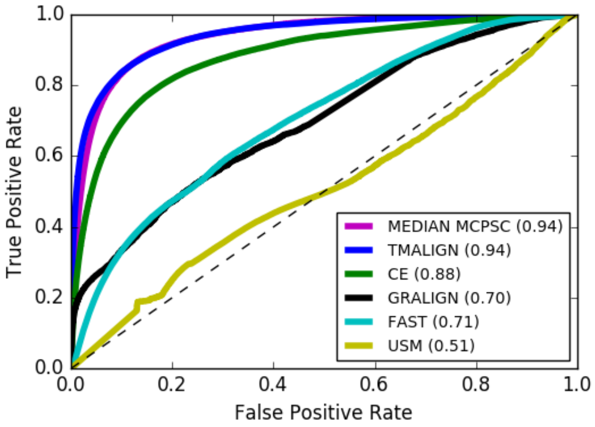


Figure 9.8: The unrooted Phylogenetic Tree based on median MCPSC consensus scores. Domains are colored according to their SCOP class (Level 1). Domains of the two clades that are marked belong to Class C but represent different functional groups.



(a) Original Dataset

(b) Common Dataset



(c) Imputed Dataset

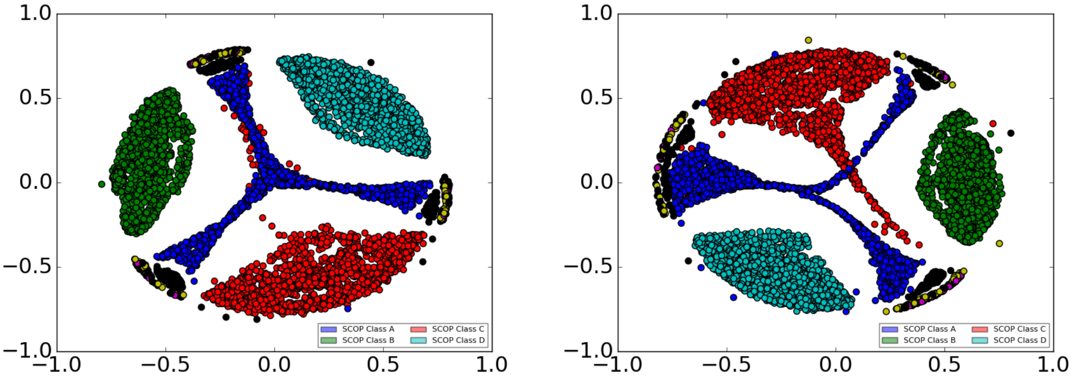
Figure 9.9: ROC curves generated by *pyMCPSC*. The panels are plots for the component PSC methods and median MCPSC method over the three variations of the domain pairs SCOP-CATH dataset. The Area Under the Curve (AUC) is provided in parentheses. The ROCs are generated at the SCOP Superfamily level (Level 3).

Table 9.4: Fraction of domains correctly classified at different SCOP hierarchy levels using a Nearest-Neighbor classifier built with similarity scores produced by different PSC and MCPSC methods. In the SCOP hierarchy: Level 1 = Class, Level 2 = Fold, Level 3 = Superfamily and Level 4 = Family.

	Original dataset				**Common subset**				**Imputed dataset**			
SCOP Level	1	2	3	4	1	2	3	4	1	2	3	4
TM-align	1.00	0.98	0.91	0.77	1.00	0.74	0.66	0.48	1.00	0.90	0.84	0.71
CE	1.00	0.62	0.56	0.46	1.00	0.49	0.44	0.32	1.00	0.57	0.51	0.42
GRALIGN	1.00	0.91	0.84	0.67	1.00	0.70	0.62	0.47	1.00	0.56	0.47	0.37
FAST	1.00	0.11	0.10	0.09	1.00	0.10	0.09	0.07	1.00	0.11	0.10	0.08
USM	1.00	0.39	0.31	0.24	1.00	0.46	0.38	0.28	1.00	0.36	0.29	0.22
M1	1.00	0.94	0.86	0.68	1.00	0.75	0.66	0.49	1.00	0.85	0.77	0.59
M2	1.00	0.94	0.86	0.68	1.00	0.75	0.66	0.48	1.00	0.86	0.78	0.60
M3	1.00	0.96	0.88	0.70	1.00	0.75	0.66	0.49	1.00	0.87	0.79	0.62
M4	1.00	0.94	0.86	0.68	1.00	0.75	0.66	0.49	1.00	0.85	0.77	0.59
M5	1.00	0.98	0.91	0.76	1.00	0.75	0.67	0.49	1.00	0.90	0.83	0.66
Median MCPSC	1.00	0.94	0.86	0.69	1.00	0.75	0.66	0.49	1.00	0.87	0.79	0.61

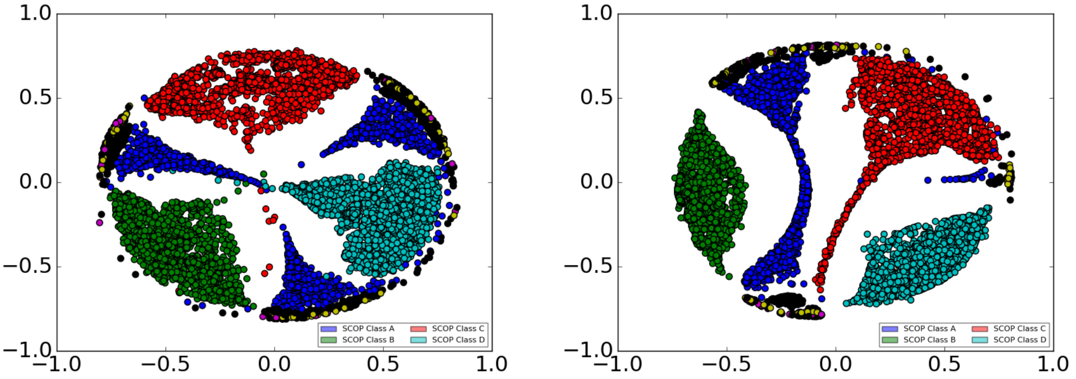
9.6.4 Multidimensional Scaling Scatterplots of protein domains

In Figures 9.10 and 9.11 we provide the Multi-dimensional Scaling [130] based scatterplot visualizations generated using distance matrices based on the scaled dissimilarity scores of the different PSC methods and MCPSC methods. The figure depicts the domains colored by SCOP class to which they belong. The figures highlight the difference between the spatial arrangements of the different classes and the fairly clear separation between them for the bigger SCOP classes. Domains belonging to Class C (α/β) are split into two major subclusters. This points towards an inherent grouping of domains belonging to this class and is potentially a reflection of the presence of two large Fold levels (SCOP level 2) in Class C in this dataset. Analysis of the domains belonging to the two main sub-clusters of Class C revealed that this is in fact the case. One of the sub-clusters has a dominating presence of domains belonging to Architecture level (CATH) *3-Layer (aba) Sandwhich* (64%) while the other is dominated by domains belonging to the Architecture level *2-Layer Sandwhich* (50%). These two Arcitecture levels contribute nearly 1000 domains each to the Class, while the remaining Architectures in the class (13 in all) are significantly smaller. It is also interesting to note from the figure that Class D ($\alpha + \beta$) domains tend to appear at spatial points dominated by points of other Classes. Class D is spread over the entire structural space in general, indicating close structural similarities between members of this Class with those of other SCOP classes.



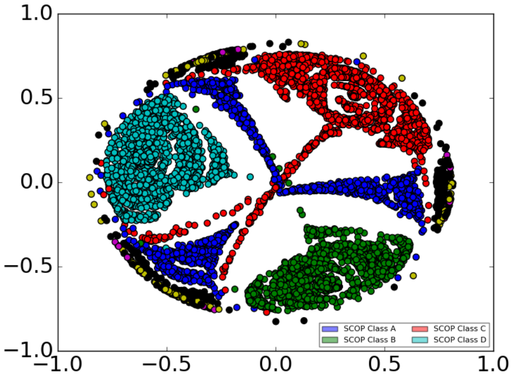
(a) CE

(b) FAST



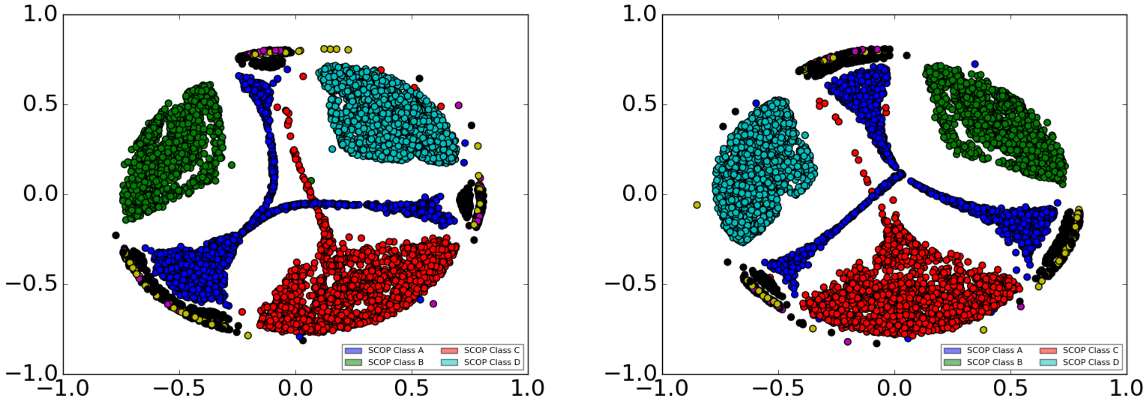
(c) TM-align

(d) GRALIGN



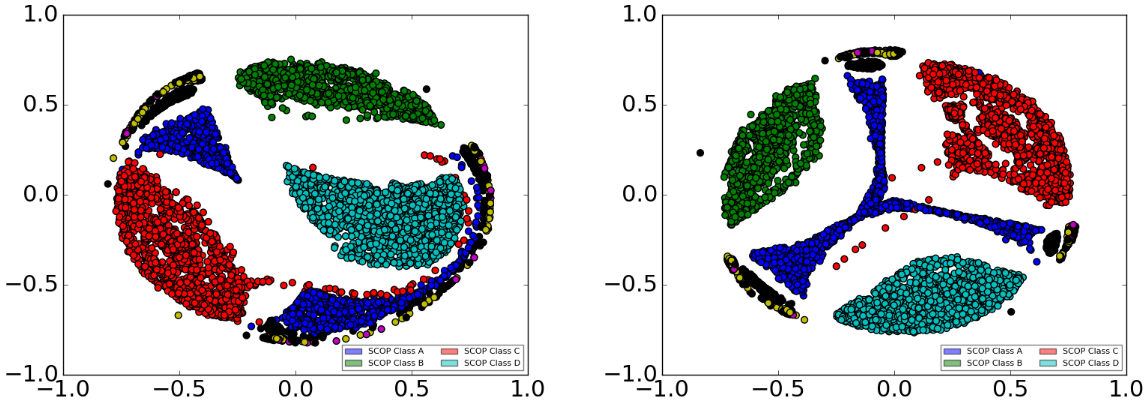
(e) USM

Figure 9.10: MDS scatter-plots for the five PSC methods generated using distance matrices and the imputed dataset. The points are colored by the ground-truth SCOP Level 1 classification of each domain. Blue = SCOP Class A, Green = SCOP Class B, Red = SCOP Class C and Cyan = SCOP Class D. Black points are domains belonging to other classes.



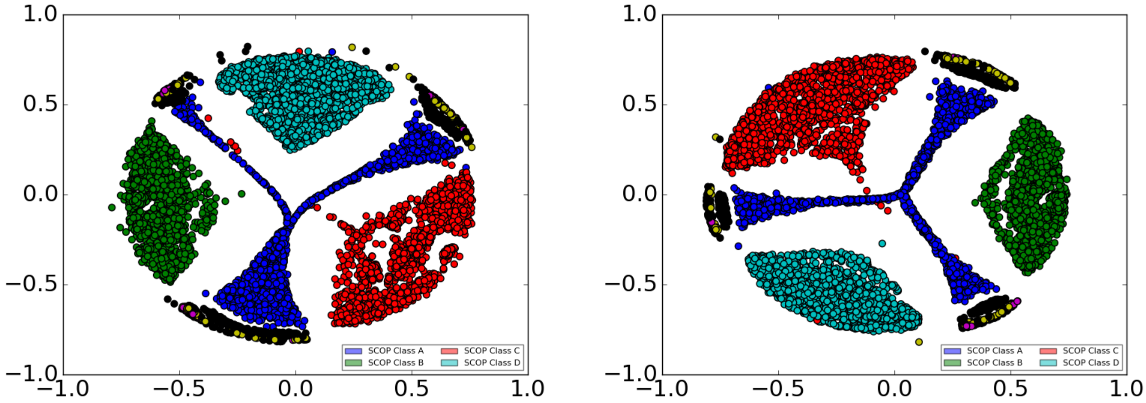
(a) M1

(b) M2



(c) M3

(d) M3



(e) M5

(f) Median

Figure 9.11: MDS scatter-plots for the five MCPSC methods generated using distance matrices and the imputed dataset. The points are colored by the ground-truth SCOP Level 1 classification of each domain. Blue = SCOP Class A, Green = SCOP Class B, Red = SCOP Class C and Cyan = SCOP Class D. Black points are domains belonging to other classes.

10. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

As the number of protein structures grows we are faced with the important but complex task of assigning function to proteins as well as classifying them into biologically pertinent groups. The faster these tasks can be performed, the faster biologists and medical researchers can determine possible applications for the protein. The main focus of this thesis was on developing computational methods which facilitate fast and efficient Multi-criteria Protein Structure Comparison (MCPSC).

In the near future, we will see a continued increase in the amount of cores integrated on a single chip as well as increased availability of commodity many-core processors. This trend is already visible today in that off-the-shelf PCs contain processors with up to 8 cores, server grade machines often contain multiple processors with up to 32 cores, and multi-core processors are even appearing in average grade mobile devices. Further, the success of GPUs, Tilera's architecture and Intel's initiative for integrating more and more cores on a single chip also attest to this trend. The ubiquity and the advanced core architectures employed by multi-core CPUs make it imperative to build software that can efficiently utilize their processing power. Our experiments, presented in Chapter 7, show that a modern Core i7 is able to deliver high throughput for all-to-all MCPSC. Utilizing the performance gain delivered by multi-core CPUs becomes especially important when considered in combination with distributed setups, such as clusters, which may already contain nodes with multiple cores.

Scalability limitations of the bus-based architecture, coupled with the greater potential of Network-on-Chip based processors to deliver efficiency and high performance, implies that NoCs are likely to be used extensively in future many-core processors. It is therefore important to start developing software frameworks and solutions that capitalize on this parallel architecture to meet the increasing computational demands in structural proteomics and bioinformatics in general. Our experimental results, presented in Chapter 6, make it clear that Intel's Single-chip Cloud Computer (SCC) Network-on-Chip (NoC) processor matches the speedup and efficiency achieved by a cluster of faster workstations [113]. However, a market ready NoC CPU will have a much better per watt performance as compared to a cluster while also saving in space and infrastructure management costs. Additionally any savings in watts consumed also reflects in savings in cooling infrastructure required for the hardware. Furthermore, with the per watt performance being in focus for processor manufacturers, such as Intel and Tilera, this gap is set to expand even further. As this study demonstrates, the likely increase in the availability and ubiquitousness of many-core processors, the near linear speedup in tackling the MCPSC scenario and the ease of porting new PSC methods to NoC based processors, make many-core processors of great interest for the high performance structural proteomics and bioinformatics communities in general.

Here we summarize our contributions and directions for future research.

10.1 Fast MCPSC with modern processor architectures

Using many-core CPUs

In Chapter 5, we discussed in detail the solution developed for utilizing the parallelism offered by the Intel SCC for all-to-all MCPSC. We presented the design of an algorithmic skeleton library which we used for porting PSC methods to the SCC. The source code for this work has been made publicly available and can be used by researchers with access to such hardware to improve efficiency of their PSC related tasks.

With the number of many-core CPUs available in the market on the rise, due to efforts from Tilera and Intel e.g. Intel Xeon Phi, it would be of interest to port *rckskel* to these new architectures. For this purpose, the library should be re-factored to isolate the communication sub-system, which currently relies on RCCE, such that porting it to other architectures becomes equivalent to replacing the underlying communications library. Such an effort could help generalize the task of porting PSC methods to different architectures so that new hardware can be quickly experimented with as and when it becomes available.

Further, studying the energy efficiency of many-core CPUs with respect to multi-core is an interesting direction in which work could be carried out especially as the NoCs get bigger. On future larger NoC processors optimal solutions may require the use of clusters of cores, concurrently processing PSC jobs with different methods. Using the *rckskel* library would facilitate the software development, for such more complicated scenarios, hiding low level details from the users. To this end it would be useful to introduce communication controls that will allow true blocking implementations of ’send’ and ’recv’ regardless of the communication library used (RCCE for example provides busy-wait loops for blocking) as well as energy management routines (currently not enabled). Such an implementation would increase the energy efficiency of applications built using the library.

Using multi-core CPUs

The ubiquity of multi-core CPUs makes them a very attractive option for speeding-up applications. In this work we developed two types of applications for the multi-core architectures a) using a master-slaves strategy to study efficiency of a multi-core CPU compared to a many-core CPU and b) a utility for performing large scale MCPSC, in Chapter 8. Both our solutions have been provided publicly so that researchers can take advantage of them.

The large-scale MCPSC utility makes it possible to run experiments with very large datasets on a local workstation through an easy to use GUI. We believe that this utility should prove useful across the domain and help researchers that are not very comfortable with IT processes to leverage MCPSC in their work. Further, because the utility is easy to extend (with additional PSC methods), it should enable more IT savy researchers to quickly carry out PSC experiments, using PSC methods of their choice, in order to compare and contrast them with existing methods.

In the long run, availability of such utilities is essential for the broader Bioinformatics community to leverage PSC data in their day to day tasks. This includes researchers in domains ranging from those involved with classification of proteins to those involved in determining functions of proteins. A formal API may be developed for such utilities allowing a standardization of PSC software to make it more amenable towards building plug-and-play software, allowing researchers to add/remove PSC methods with little to no need for programming expertize.

Both multi- and many-core architectures lend themselves to the popular and powerful MapReduce parallel programming model. If the underlying communication libraries - typically built on top of Message Passing Interface (MPI) - are available, this highly popular approach can be applied to large scale MCPSC in a manner similar to its application in other domains of Bioinformatics [148]. While some implementation of MapReduce are available for multi-core machines, it must be noted that no MapReduce framework implementations are available for the SCC. One possible approach for using MapReduce in MCPSC would be to a) determine pairwise PSC to be performed on each PE (pre-Map), b) run all PSC methods on all pairs on each PE in parallel (Map), c) collect PSC scores from all PEs to a designated PE (shuffle) and d) combine and generate final MCPSC scores (Reduce). Movement of data for task distribution (pre-Map), intermediate data transfer performed by the ’shuffle’ mechanism and collection of results, if required by the Reduce step, would benefit from low cost of inter-PE communication. Load balancing methods similar to those discussed in this work could be useful in determining effective strategies for the distribution of PSC tasks (pre-Map). Analysis of MCPSC performance on multi- and many-core processors with MapReduce would be required to understand the design trade-offs that would lead to efficient designs.

10.2 Structured analysis of MCPSC based classification of proteins

Large-scale MCPSC is the norm with the huge database sizes and the need for multi-criteria, it is therefore important to carry out structured analysis of large-scale MCPSC results. The rate of deposition of complex and non-globular structures in the PDB is on the rise and is likely to continue to increase. Thus both the number and variety of recorded structures will increase making characterizing the function of known structures a major challenge [19]. This challenge not only calls for the use of fast techniques for dealing with such large data but also for structured analysis of the structural space. In this work we used the solutions developed for efficient all-to-all MCPSC to carry out a systematic analysis of classifying and clustering protein domains from a very large dataset, using consensus based MCPSC scores.

In Chapter 8, we present a detailed analysis of results obtained from large-scale MCPSC, showing that MCPSC delivers near optimal performance in terms of classification and clustering of proteins. This is largely because MCPSC scores tend to trace the best performing component PSC method. We believe that this is an interesting result because in the absence of the ground-truth data it is not evident which PSC methods is performing the best.

Consensus based MCPSC scores can therefore be used to analyze existence of clusters within datasets of biological importance as well as to auto-classify protein (domains) into categories similar to SCOP/CATH.

Real-world data problems, such as missing data, analyzed in this work are likely to plague any large-scale MCPSC analysis. Due to a variety of factors, such as a) runtime environmental, b) PSC method limitations and c) errors in PDB files, there is likely to be a lot of unprocessed protein pairs in the PSC scores datasets. We analyzed multiple imputation techniques and identify methods that works well. This result is of direct use to leveraging PSC results in applications such as classification/clustering because of the high likelihood of missing data issues. Such systematic analysis of dealing with a problem that is likely to get worse as the sparsity of protein-protein PSC score matrices gets bigger (due to increasing number of known protein structures) is crucial and our work can form the basis for research in this direction.

Finally, more effort is needed to study further the complete structure space by carrying out even larger-scale experiments (with all the PDB for instance). Understanding the nature of the complete structure space can be catalytic in defining and identifying unique structural units that are recurrent between protein structures. Such a finding could help design better structure alignment metrics, as well as improve the ability of existing methods to categorize proteins in a biologically relevant manner. Currently, domains are considered evolutionary units because they show similar activity even when extracted from a protein chain all together. Thus proteins with similar domains or with similar arrangements of domains are considered similar. However, the sequence diversity with the domains as recurring structural units is still huge and certainly worth exploring both for causes and consequences.

APPENDIX I - INSTALLING *PYMCPSC*

This appendix describes the process for installing the *pyMCPSC*utility on a users machine. The pyMCPSC Project is a utility for performing large-scale Multi-criteria Protein Structure Comparison (MCPSC) on a multi-core CPU with an easy to extend API for adding and removing PSC methods. It makes use of several relevant python tools to perform typical associated tasks.

Installation

From Docker Image

First, make sure you have installed Docker on your machine following instructions provided at https://docs.docker.com/install/. Once Docker is installed check it by running 'docker –version' to ensure it is in your path. Note that Windows users with older versions (before 10) will need to follow the install path for Docker-toolbox.

Locate the docker image of pyMCPSC that you have built or downloaded and load it (the daemon should be running) as follows::

```
docker load -i [path_to_docker_image.tar]
```

Assuming that the name of the docker image is pymcpsc you should now be able to see the image in the docker list by issuing the following command::

```
docker images
```

Test if pyMCPSC can be executed in the image by issuing the following command::

```
docker run pymcpsc /usr/src/app/scripts/docker-launch-pymcpsc.sh -h
```

Executing this command should print the usage help message of pyMCPSC with command line arguments it accepts.

Building a Docker Image

Assuming the requisite Docker tools are installed it is possible to build the docker image for pyMCPSC. Navigate to the location where you downloaded pyMCPSC (or cloned it) and issue the following command::

```
docker build -t pymcpsc .
```

This process will take a little time, depending on your local machine and network connections, but once successfully completed the pyMCPSC image will appear in the docker images list.

Native installation

This installation mode is relevant to developers who wish to extend pyMCPSC and **works only on Linux based systems**. Before pyMCPSC is installed the dependencies of pyMCPSC must be made available. The suggested route is to use the Anaconda/Miniconda platform which has a higher success rate across Linux systems, however, it is certainly possible to install the dependencies independently of the platform. We currently recommend using Python 2.7, Python 3.5 or Python 3.6 as we have only tested with these versions of Python.

With Miniconda

First, Download and install Miniconda 4.0.5 (https://www.continuum.io/archives).

Python 2 users on Linux may use::

```
wget https://repo.continuum.io/miniconda/Miniconda2-4.0.5-Linux-x86_64.sh
```

Python 3 users on Linux may use::

```
wget https://repo.continuum.io/miniconda/Miniconda3-4.0.5-Linux-x86_64.sh
```

Once Miniconda has been installed and is in the user path install the dependencies as follows::

```
conda install pyqt=4.11 numpy matplotlib pandas scikit-learn scipy seaborn
```

Without Anaconda

This requires installation of all the dependencies individually. You will need admin / sudo rights on your system depending on how you choose to install the dependencies. On a Ubuntu / Debian based system the users should run the following commands to install the required packages::

```
sudo apt install python-pyqt4 python-lxml python-numpy \
                 python-scipy python-matplotlib python-seaborn
```

Finally, navigate to the location where you have downloaded pyMCPSC source and execute the following commands::

```
python setup.py build
python setup.py install
```

Test if pyMCPSC has installed correctly by issuing the following command::

```
run-pymcpsc -h
```

The output should be the help message showing the arguments accepted by the utility.

APPENDIX II - EXPERIMENTAL SETUP WITH *PYMCPSC*

pyMCPSC supports multiple usage modes. For those that are only interested in running the utility a Docker image may be the best place to start. For those interested in 'natively' deploying the utility it is also possible to do so, however, this route will work only on Linux based systems.

Configuring an experiment

Preparing the dataset

In order to run pyMCPSC the user must have protein (domain) structure files available locally on the machine where pyMCPSC will be run. A useful script to download structure files from the PDB has been included scripts/download.sh. The script creates a directory pdb files in the CWD and downloads the structure files into this directory. pyMCPSC includes structure files for the Chew Kedem dataset in tests/chew kedem dataset/pdb files.

Preparing the ground truth file

The next step is to prepare the ground truth file which provides the list of protein domain pairs to be included in the PSC and their SCOP/CATH classification. The ground truth file is a tab separated value file with 4 columns: protein 1, protein 2, classification of protein 1 and classification of protein 2. A sample ground truth file included in pyMCPSC is tests/chew kedem dataset/ground truth ck. A useful script to prepare the ground truth file from a listing of domains and their classifications is included scripts/prepare ground truth.py. The script writes out the ground truth file in the CWD.

Setting up the experiment

pyMCPSC supports several parameters to control the experiment these are as follows

- *DATADIR*: The value should be the full path to the directory with the PDB structure files.
- *GTIN*: The value should be the full path to the ground-truth data file
- *PDBEXTN*: The value should be the extension of the PDB structure files

- *THREADS*: The value should be the number of threads pyMCPSC is allowed to launch.
- *WEIGHTS*: The value should be a comma separated string with weights for the PSC methods (in order - ce,fast,gralign,TM-align,usm).
- *PROGDIR*: The value should be the full path to the directory with the PSC binaries.

Typically the user will not supply this value unless the PSC binaries have been compiled to some custom location. The program parameters can be specified on the CLI. pyMCPSC provides a set of sensible default fallback values for the optional arguments. If no value is specified for the Datadir, the pre-packaged proteus dataset is automatically selected. If no value is specified for Progdir, the default set of five PSC methods is used. If no value is specified for the Gtin and the proteus dataset is used, the default ground truth data is automatically selected. Note that if the proteus dataset is not used then then the user must provide a ground truth file for the performance benchmarking to be performed. If no values are specified by the user pyMCPSC runs the experiment described in the original paper.

Running an experiment

A key aspect of pyMCPSC that is relevant to the experimental setup is that pyMCPSC generates several output Figures and data files (that the user may wish to analyze with other tools). The outputs are written to subfolders in the CWD it is therefore important to ensure this is a writeable location especially when running in Docker mode.

From Docker

The simplest way to run an experiment with pyMCPSC (recreating the results of the original paper on Proteus_300 dataset) is to issue the following command::

```
docker run -v [absolute_path]:/usr/shared pymcpsc:latest \
              /usr/src/app/scripts/docker-launch-pymcpsc.sh
```

Note that we are mounting a path on the local filesystem to a specfic location in the docker which is where pyMCPSC expects to write its output. This path must be absolute and not relative.

In order to use your own dataset of domains and ground truth (see previous sections on how to generate this data) the user must place these in the same directory that will be mounted to /usr/shared. For instance, create a subdirectory data in the 'absolute_path' and place the PDB files (with pdb extension) in the directory and place the corresponding ground truth data in a file 'ground_truth'. Then issue the following command to run pyMCPSC with the custom data::

```
docker run -v [absolute_path]:/usr/shared pymcpsc:latest \
              /usr/src/app/scripts/docker-launch-pymcpsc.sh \
              -e pdb -d /usr/shared/data \
              -g /usr/shared/ground_truth
```

Note that both, directory data and file ground_truth, reside at [absolute_path] on the local filesystem but are passed to pyMCPSC as arguments in the location where they are expected to be found in the Docker.

We have included a small dataset (Chew-Kedem) dataset in pyMCPSC sources (tests/chew_kedem_dataset).

From native installation

To run a MCPSC experiment with pyMCPSC create a directory, TEST_DIR, where the experiment outputs will be written. We refer to the location where pyMCPSC was cloned (unpacked from zip) as CLONE_DIR. Running pyMCPSC is as easy as invoking the run-pymcpsc command with the appropriate parameters::

```
cd $TEST_DIR$
run-pymcpsc [-h ] [-e PDBEXTN] [-d DATADIR] [-g GTIN ] [-t THREADS]
            [-w WEIGHTS] [-p PROGDIR]
```

Results generated by pyMCPSC are placed in the work and outdir directories located in the current working directory (CWD), i.e. the one from where the program is launched. Moreover, figures generated by pyMCPSC are placed in the figures directory in the CWD. To run pyMCPSC with the Proteus_300 dataset (prepacked) and generate the results reported in the original paper use the command::

```
cd $TEST_DIR$
run-pymcpsc
```

Sample dataset for experiment

We include in pyMCPSC the Chew-Kedem dataset and associated ground truth file as a test dataset. To configure pyMCPSC to run with this dataset point pyMCPSC to run with with the tests/chew kedem dataset/pdb files folder (as DATADIR) and tests/chew kedem dataset/ground truth ck (as GTIN). The parameter PDBEXTN should be set as PDB. A minimal commnand for using this dataset is listed below (INSTALL_DIR is the location where pyMCPSC was extracted)::

```
cd $TEST DIR$
```

```
run-pymcpsc -e pdb \
            -d $INSTALL_DIR$/pymcpsc/tests/chew_kedem_dataset/pdb_files \
            -g $INSTALL_DIR$/pymcpsc/tests/chew_kedem_dataset/groundtruth_ck
```

APPENDIX III - EXTENDED NEAREST-NEIGHBOR CLASSIFICATION ANALYSIS

Using the data generated by *pyMCPSC* (stored in file processed.imputed.mcpsc.csv) a break down of the performance of the different PSC and Median MCPSC method was generated, as shown in Table B below. In addition to the data generated by *pyMCPSC*functional annotation of the SCOP families were obtained from (http://scop.berkeley.edu) in order to assess if there is a correlation between the classification performance and the domain SCOP families. It is interesting to observe that there is no component method that is uniformly best on all families. Moreover, the consensus Median MCPSC classifier performs best for most families, with TM-align exceeding its performance on very few cases, however there is no pattern to these families. Interestingly, purely RMSD based methods (CE and Fast) are the least effective for most Folds. Further, we compared the domains misclassified by the different PCS/MCPSC method however there appeared to be no correlation between the misclassified domains.

Table 1: Fraction of domains correctly classified at SCOP family level using a Nearest-Neighbor classifier built with similarity scores produced by different PSC and MCPSC methods.

Median MCPSC	TM-align	CE	FAST	GRALIGN	USM	# Domains	SCOP Family	Functional Annotation
1.00	1.00	0.13	0.00	1.00	0.63	8	c.67.1.1	AAT-like
1.00	0.89	0.22	0.00	1.00	0.11	9	c.1.10.1	Class I aldolase
1.00	1.00	0.50	0.10	0.90	0.80	10	d.131.1.2	DNA polymerase processivity factor
1.00	1.00	0.40	0.00	1.00	1.00	10	a.45.1.1	Glutathione S-transferase (GST), C-terminal domain
1.00	1.00	0.50	0.00	0.90	0.50	10	c.94.1.1	Phosphate binding protein-like
1.00	1.00	1.00	0.10	0.90	0.90	10	a.25.1.1	Ferritin
1.00	1.00	1.00	0.00	1.00	0.86	7	b.36.1.1	PDZ domain
0.89	1.00	0.33	0.00	0.89	0.78	9	d.54.1.1	Enolase N-terminal domain-like
1.00	1.00	0.56	0.00	0.89	0.78	9	d.108.1.1	N-acetyl transferase, NAT
1.00	1.00	0.88	0.00	1.00	0.88	8	d.15.1.1	Ubiquitin-related
0.90	1.00	0.30	0.00	0.90	0.80	10	d.144.1.7	Protein kinases, catalytic subunit
1.00	1.00	0.14	0.00	1.00	0.71	7	a.104.1.1	Cytochrome P450
1.00	1.00	0.00	0.00	1.00	0.30	10	c.1.8.3	beta-glycanases
1.00	1.00	0.11	0.00	1.00	0.78	9	c.93.1.1	L-arabinose binding protein-like
1.00	1.00	1.00	0.11	0.89	1.00	9	d.162.1.1	Lactate and malate dehydrogenases, C-terminal domain
1.00	1.00	0.60	0.00	0.90	0.60	10	c.37.1.20	Extended AAA-ATPase
1.00	1.00	0.60	0.80	1.00	1.00	10	c.2.1.5	LDH N-terminal domain-like
1.00	1.00	1.00	0.13	0.88	0.75	8	c.23.1.1	CheY-related
1.00	1.00	0.80	0.00	0.20	0.00	10	d.153.1.4	Proteasome subunits
1.00	1.00	0.56	0.00	1.00	0.44	9	c.2.1.2	Tyrosine-dependent oxidoreductases
0.89	1.00	0.89	0.22	0.89	0.78	9	d.58.7.1	Canonical RBD
1.00	1.00	0.80	0.30	0.70	0.70	10	d.169.1.1	C-type lectin domain
1.00	1.00	0.78	0.00	0.78	0.78	9	a.1.1.2	Globins
1.00	1.00	0.83	0.50	1.00	1.00	6	d.58.17.1	HMA, heavy metal-associated
1.00	1.00	0.63	0.00	0.50	0.50	8	b.1.2.1	Fibronectin type III
1.00	1.00	0.90	0.00	0.70	0.40	10	b.69.4.1	WD40-repeat
1.00	1.00	0.50	0.40	1.00	0.60	10	c.37.1.8	G proteins
1.00	1.00	0.90	0.00	0.40	0.50	10	b.1.1.2	C1 set domains (antibody constant domain-like)
0.89	1.00	0.00	0.00	0.89	0.44	9	a.123.1.1	Nuclear receptor ligand-binding domain
0.86	0.86	0.71	0.00	0.71	0.29	7	b.1.1.4	I set domains

APPENDIX IV - SIMILARITY BASED UNROOTED PHYLOGENETIC TREES

We can use *pyMCPSC* to generate an unrooted ‘Phylogenetic Tree’ visualization of the dataset domains. A small part of an unrooted Phylogenetic tree constructed is shown in Fig 5 of the manuscript (the complete Tree shown in Fig 1 below). Two Clades were selected for comparison that are at the same level in the tree and contain domains from the same SCOP Class. The goal was to assess the potential difference between the domains in the two clades in terms of their biological function. In order to obtain information about the function of the domains the following process was repeated:

- Download pdb-to-swissprot data from: http://www.uniprot.org/docs/pdbtosp
- Download uniprot descriptions from: http://uniprot.org
- Find keywords applicable to each domain (pdb - swissprot number - keyword)

At the end of this process we have obtained all keywords associated with each domain of the dataset. The most frequent keyword occurring in the domains of each clade was then determined. Clade 1 consists of 9 domains: d1wf3a1, d1r2qa_, d3raba_, d1ctqa_-, d1r8sa_, d1svia_, d1mkya2, d1kk1a3, d1i2ma_ and Clade 2 consists of 4 domains: d1a04a2, d1w25a1, d1qkka_, d1w25a2 (domains for which no keywords were found were excluded). Table 2 lists the frequencies of the keywords for both clades. As can be seen domains of Clade 1 are G-protein regulators while Clade 2 domains are Phosphoproteins.

Clade 1		Clade 2	
Function	**Count**	**Function**	**Count**
GTP-binding	9	Phosphoprotein	4
Nucleotide-binding	9	Nucleotide-binding	4
Complete proteome	9	Complete proteome	4
3D-structure	9	Two-component regulatory system.	4
Reference proteome	8	3D-structure	4
Membrane	5	Cytoplasm	3
Cytoplasm	5	Transcription	2
Cell membrane	5	Transferase	2
Lipoprotein	4	Reference proteome	2
Protein transport	4	Activator	2
Transport.	3	GTP-binding	2
Direct protein sequencing	3	Repeat	2
Prenylation	3	Transcription regulation	2
Acetylation	3	DNA-binding	2
Golgi apparatus	2	Cell cycle	2
Cell division	2	Metal-binding	2
Alternative splicing	2	ATP-binding	2
Phosphoprotein	2	Magnesium	2
Nucleus	2	Differentiation	2
Cell cycle	2	Transducer	2
Methylation	2	Nitrate assimilation	1
Polymorphism	1	Plasmid	1
Cytoplasmic vesicle	1	Repressor	1

ER-Golgi transport	1		
Disease mutation	1		
Septation.	1		
Cell inner membrane	1		
Ribosome biogenesis	2		
Metal-binding	1		
rRNA-binding.	1		
Proto-oncogene	1		
Ubl conjugation.	1		
Exocytosis	1		
Host-virus interaction	1		
Synaptosome	1		
Synapse	1		
Initiation factor	1		
Endocytosis	1		
S-nitrosylation.	1		
Magnesium	1		
Protein biosynthesis.	1		
Postsynaptic cell membrane	1		
RNA-binding	1		
Endosome	1		
Phagocytosis	1		
Myristate	1		
Palmitate	1		
Isopeptide bond	1		
Mitosis	1		
Repeat	1		
Cell projection	1		
Cell junction	1		
Transport	1		

Table 2: Frequency of functional keywords obtained for the domains of the two Clades identified on the Phylogenetic Tree. Keywords are obtained from Uniprot by matching the domain names with corresponding Uniprot Ids.

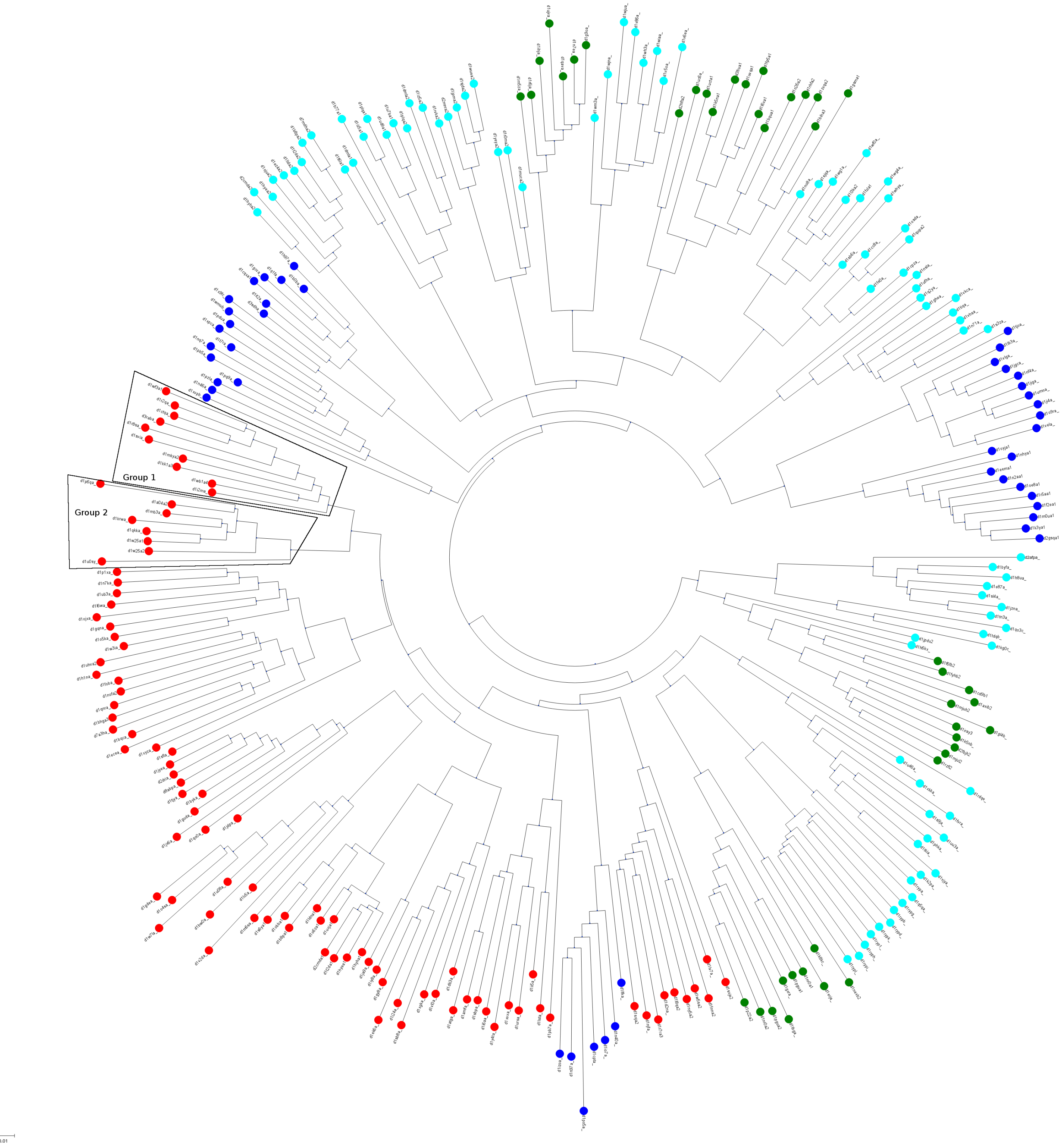


Figure 1: The unrooted 'Phylogenetic Tree' based on median MCPSC consensus scores. Domains are colored according to their SCOP class (Level 1). Domains of both clades circled belong to Class C but represent different functional groups.

REFERENCES

[1] Measuring the Similarity of Protein Structures by Means of the Universal Similarity Metric (Auxiliary Programs and Scripts). URL:http://www.cs.nott.ac.uk/~nxk/USM/protocol.html.

[2] RCCE: a small library for Many-Core communication. URL:http://techresearch.intel.com/spaw2/uploads/files//RCCE_Specification.pdf.

[3] pandas: Python Data Analysis Library. Online, 2012.

[4] N. N. Alexandrov. SARFing the PDB. *Protein Engineering*, 9:727–732, 1996.

[5] Rumen Andonov, Noël Malod-Dognin, and Nicola Yanev. Maximum contact map overlap revisited. *Journal of Computational Biology*, 18(1):27–41, January 2011.

[6] Rumen Andonov, Nicola Yanev, and Noël Malod-Dognin. *An Efficient Lagrangian Relaxation for the Contact Map Overlap Problem*, pages 162–173. Springer, 2008.

[7] Rumen Andonov, Nicola Yanev, and Noël Malod-Dognin. An Efficient Lagrangian Relaxation for the Contact Map Overlap Problem. In Keith A. Crandall and Jens Lagergren, editors, *WABI*, volume 5251 of *Lecture Notes in Computer Science*, pages 162–173. Springer, 2008.

[8] P. J. Artymiuk, A. R. Poirrette, H. M. Grindley, D. W. Rice, and P. Willett. A graph-theoretic approach to the identification of three-dimensional patterns of amino acid side-chains in protein structures. *Journal of molecular biology*, 243(2):327–344, October 1994.

[9] David Atienza, Federico Angiolini, Srinivasan Murali, Antonio Pullini, Luca Benini, and Giovanni De Micheli. Network-On-Chip Design and Synthesis Outlook. *INTEGRATION*, vol. 41, n. 2:1 – 35, 2008. [ARTICOLO].

[10] M. Azimi, N. Cherukuri, D. Jayashima, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. Vaidya. Integration Challenges and Tradeoffs for Tera-scale Architectures. *Intel Technology Journal*, 11:173–184, 2007.

[11] S. Balaji and N. Srinivasan. Comparison of sequence-based and structure-based phylogenetic trees of homologous proteins: Inferences on protein evolution. *Journal of Biosciences*, 32(1):83–96, 2007.

[12] Aydin O. Balkan, Gang Qu, and Uzi Vishkin. A Mesh-of-Trees Interconnection Network for Single-Chip Parallel Processing. In *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors*, ASAP '06, pages 73–80, Washington, DC, USA, 2006. IEEE Computer Society.

[13] D. Barthel, J.D. Hirst, J. Blacewicz, and N.Krasnogor. ProCKSi: a Metaserver for Protein Comparison Using Kolmogorov and Other Similarity Measures. *BMC Bioinformatics*, 8:416, 2007.

[14] Daniel Barthel, Jonathan D Hirst, Jacek Bewicz, Edmund K Burke, and Natalio Krasnogor. ProCKSI: a decision support system for Protein (Structure) Comparison, Knowledge, Similarity and Information. *BMC Bioinformatics*, 8:416, 2007.

[15] Daniel Barthel, Jonathan D. Hirst, Jacek Blazewicz, Edmund K. Burke, and Natalio Krasnogor. ProCKSI: a decision support system for protein (structure) comparison, knowledge, similarity and information. *BMC Bioinformatics*, 8:416, October 2007.

[16] F. Birzele, J. E. Gewehr, G. Csaba, and R. Zimmer. Vorolign–fast structural alignment using Voronoi contacts. *Bioinformatics*, 23(2), January 2007.

[17] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of Network-on-chip. *ACM Comput. Surv.*, 38(1), June 2006.

[18] G. Blake, R. G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, November 2009.

[19] Philip E. Bourne and Ilya N. Shindyalov. *Structure Comparison and Alignment*, pages 321–337. John Wiley and Sons, Inc., 2005.

[20] Duncan K. G. Campbell. Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276, University of York, 1996.

[21] Alberto Caprara, Robert D. Carr, Sorin Istrail, Giuseppe Lancia, and Brian Walenz. 1001 Optimal PDB Structure Alignments: Integer Programming Methods for Finding the Maximum Contact Map Overlap. *Journal of Computational Biology*, 11(1):27–52, 2004.

[22] Lei Chai, Qi Gao, and Dhabaleswar K. Panda. Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID ’07, pages 471–478, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Shuai Che, Jie Li, J W Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with GPUs and FPGAs. *2008 Symposium on Application Specific Processors*, pages 101–107, June 2008.

[24] Luonan Chen, LingYun Wu, Yong Wang, Shihua Zhang, and Xiang-Sun Zhang. Revealing divergent evolution, identifying circular permutations and detecting active-sites by protein structure comparison. *BMC Structural Biology*, 6:18, September 2006.

[25] Luonan Chen, Tianshou Zhou, and Yun Tang. Protein structure alignment by deterministic annealing. *Bioinformatics*, 21(1):51–62, 2005.

[26] L. P. Chew, D. D. Huttenlocher, K. Kedem, and Jon M. Kleinberg. Fast detection of common geometric substructure in proteins. In *Proc. 3rd Annual Conference on Research in Computational Molecular Biology (RECOMB)*, pages 104–114, 1999.

[27] L. Paul Chew and Klara Kedem. Finding the consensus shape for a protein family (extended abstract). In *18th ACM Symposium on Computational Geometry*, pages 64–73, 2002.

[28] PinHao Chi. *Efficient protein tertiary structure retrievals and classifications using content based comparison algorithms*. PhD thesis, University of Missouri at Columbia, Columbia, MO, USA, 2007.

[29] Peter J. A. Cock, Tiago Antao, Jeffrey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, and Michiel J. L. de Hoon. Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422–1423, 2009.

[30] Maxime Colmant, Romain Rouvoy, and Lionel Seinturier. Improving the energy efficiency of software systems for multi-core architectures. In *Proceedings of the 11th Middleware Doctoral Symposium*, MDS ’14, pages 1:1–1:4, New York, NY, USA, 2014. ACM.

[31] Matteo Comin, Concettina Guerra, and Giuseppe Zanotti. PROuST: A comparison method of three-dimensional structures of proteins using indexing techniques. *Journal of Computational Biology*, 11(6):1061–1072, 2004.

[32] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theor.*, 13(1):21–27, September 2006.

[33] Trevor F. Cox and M.A.A. Cox. *Multidimensional Scaling, Second Edition*. Chapman and Hall/CRC, 2 edition, 2000.

[34] G. Csaba, F. Birzele, and R. Zimmer. Protein structure alignment considering phenotypic plasticity. *Bioinformatics*, 24(16):98–104, August 2008.

[35] Gergely Csaba, Fabian Birzele, and Ralf Zimmer. Systematic Comparison of SCOP and CATH: A new Gold Standard for Protein Structure Analysis. *BMC Structural Biology*, 9(23), 2009.

[36] Aniket Dalal, Sandeep Deshmukh, and Pramod P Wangika. Protein structure classification using geometric invariants and dynamic programming. *Protein and Peptide Letters*, 14(7):658–664, 2007.

[37] Natalie L. Dawson, Tony E. Lewis, Sayoni Das, Jonathan G. Lees, David Lee, Paul Ashford, Christine A. Orengo, and Ian Sillitoe. Cath: an expanded resource to predict protein function through structure and sequence. *Nucleic Acids Research*, 45(D1):D289–D295, 2017.

[38] D F Ding, J Qian, and Z K Feng. A differential geometric treatment of protein structure comparison. *Bulletin of Mathematical Biology*, 56(5):923–943, 1994.

[39] Miquel Duran-Frigola, Lydia Siragusa, Eytan Ruppin, Xavier Barril, Gabriele Cruciani, and Patrick Aloy. Detecting similar binding pockets to enable systems polypharmacology. *PLOS Computational Biology*, 13(6):1–18, 06 2017.

[40] Tom Fawcett. An introduction to roc analysis. *Pattern Recogn. Lett.*, 27(8):861–874, June 2006.

[41] D. Fischer, A. Elofsson, D. Rice, and D. Eisenberg. Assessing the performance of fold recognition methods by means of a comprehensive benchmark. *Pacific Symposium on Biocomputing*, pages 300–318, 1996.

[42] Peter Glaskowsky. Tilera's balancing act: 100 cores vs. market realities. http://news.cnet.com/8301-13512_3-10388025-23.html, 2009.

[43] Cristian Grozea, Zorana Bankovic, and Pavel Laskov. FPGA vs. Multi-core CPUs vs. GPUs: Hands-On Experience with a Sorting Application. *Computing*, pages 1–12, 2011.

[44] Julia Handl, Joshua D. Knowles, and Douglas B. Kell. Computational cluster validation in post-genomic data analysis. *Bioinformatics*, 21(15):3201–3212, 2005.

[45] V. Joachim Haupt, Simone Daminelli, and Michael Schroeder. Drug promiscuity in pdb: Protein binding site similarity is key. *PLOS ONE*, 8(6):1–15, 06 2013.

[46] L. Holm and J. Park. DaliLite workbench for protein structure comparison. *Bioinformatics*, 16(6):566–567, June 2000.

[47] L. Holm and C. Sander. Protein structure comparison by alignment of distance matrices. *Journal of Molecular Biology*, 233(1):123–138, September 1993.

[48] L Holm and C Sander. Protein structure comparison by alignment of distance matrices. *Journal of Molecular Biology*, 233(1):123–138, 1993.

[49] Jingcao Hu and Radu Marculescu. DyAD: smart routing for networks-on-chip. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, pages 260–263. ACM, 2004.

[50] Jaime Huerta-Cepas, François Serra, and Peer Bork. Ete 3: Reconstruction, analysis, and visualization of phylogenomic data. *Molecular Biology and Evolution*, 33(6):1635, 2016.

[51] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[52] I. Jonassen, I. Eidhammer, and W. R. Taylor. Discovery of local packing motifs in protein structures. *Proteins*, 34:206–219, 1999.

[53] A. R. Kinjo, K. Horimoto, and K. Nishikawa. Predicting absolute contact numbers of native protein structure from amino acid sequence. *Proteins*, 58(1):158–165, January 2005.

[54] Patrice Koehl. *Protein Structure Classification*, pages 1–55. John Wiley & Sons, Inc., 2006.

[55] Andrey N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems in Information Transmission*, 1:1–7, 1965.

[56] R. Kolodny, D. Petrey, and B. Honig. Protein structure comparison: implications for the nature of 'fold space', and structure and function prediction. *Current Opinion in Structural Biology*, 16(3):393–398, June 2006.

[57] Rachel Kolodny and Nathan Linial. Approximate protein structural alignment in polynomial time. *Proceedings of the National Academy of Sciences of the United States of America*, 101(33):12201–12206, 2004.

[58] Elias Kouskoumvekakis, Dimitrios Soudris, and E. S. Manolakos. Many-core CPUs can deliver scalable performance to stochastic simulations of large-scale biochemical reaction networks. *International Conference on High Performance Computing & Simulation*, Accepted for publication, 2015.

[59] N. Krasnogor and D. A. Pelta. Measuring the similarity of protein structures by means of the universal similarity metric. *Bioinformatics*, 20(7):1015–1021, May 2004.

[60] E. Krissinel and K. Henrick. Secondary-structure matching (SSM), a new tool for fast protein structure alignment in three dimensions. *Acta crystallographica. Section D, Biological crystallography*, 60(Pt 12 Pt 1):2256–2268, December 2004.

[61] Ludmila I. Kuncheva and Christopher J. Whitaker. Measures of diversity in classifier ensembles. *Machine Learning*, 51(2), 2000.

[62] P Lackner, W A Koppensteiner, M J Sippl, and F S Domingues. ProSup: a refined tool for protein structure alignment. *Protein Engineering*, 13(11):745–752, 2000.

[63] R. A. Laskowski, J. D. Watson, and J. M. Thornton. ProFunc: a server for predicting protein function from 3D structure. *Nucleic Acids Research*, 33:89–93, 2005.

[64] Arthur M. Lesk. *Introduction to Bioinformatics*, pages 40–42. Oxford University Press, 2005.

[65] GongHua Li and Jing-Fei Huang. CMASA: an accurate algorithm for detecting local protein structural similarity and its application to enzyme catalytic site annotation. *BMC Bioinformatics*, 11(1):439, 2010.

[66] Wei Liu, Anuj Srivastava, and Jinfeng Zhang. A mathematical framework for protein structure comparison. *PLoS Computational Biology*, 7(2):189, February 2011.

[67] Loredana Lo Conte, Bart Ailey, Tim J. P. Hubbard, Steven E. Brenner, Alexey G. Murzin, and Cyrus Chothia. SCOP: a Structural Classification of Proteins database. *Nucleic Acids Research*, 28(1):257–259, 2000.

[68] Zaixin Lu, Zhiyu Zhao, and Bin Fu. Efficient protein alignment algorithm for protein search. *BMC Bioinformatics*, 11(Suppl 1):34, 2010.

[69] Y Luo, L Lai, X Xu, and Y Tang. Defining topological equivalences in protein structures by means of a dynamic programming algorithm. *Protein Engineering*, 6(4):373–376, 1993.

[70] Giuseppe Maccari, Giulia L B Spampinato, and Valentina Tozzini. Secstant: Secondary structure analysis tool for data selection, statistics and models building. *Bioinformatics*, 2013.

[71] M. S. Madhusudhan, Benjamin M. Webb, Marc A. Marti-Renom, Narayanan Eswar, and Andrej Sali. Alignment of multiple protein structures based on sequence and structure features. *Protein Engineering, Design and Selection*, 22(9):540–574, July 2009.

[72] Noël Malod-Dognin and Nataša Pržulj. GR-Align: fast and flexible alignment of protein 3D structures using graphlet degree similarity. *Bioinformatics*, 30(9):1259–1265, May 2014.

[73] Noel Malod-Dognin and Natasa Przulj. Gr-align: fast and flexible alignment of protein 3d structures using graphlet degree similarity. *Bioinformatics*, 30(9):1259–1265, 2014.

[74] Bryan Marker, Ernie Chan, Jack Poulson, Robert Geijn, Rob Wijngaart, Timothy Mattson, and Theodore Kubaska5. Programming many-core architectures - a case study: dense matrix computations on the Intel SCC processor. *Concurrency And Computation: Practice And Experience*, pages 1–18, 2011.

[75] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core SCC Processor: the Programmer's View. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[76] Lazaros Mavridis and David Ritchie, W. 3D-blast: 3D protein structure alignment, comparison, and classification using spherical polar Fourier correlations. In *Pacific Symposium on Biocomputing 2010*, pages 281–292, Hawaii, United States, Jan 2010. World Scientific Publishing.

[77] Matthew Menke, Bonnie Berger, and Lenore Cowen. Matt: Local flexibility aids protein multiple structure alignment. *PLoS Comput Biol*, 4(1):10, January 2008.

[78] Stephan Mertens. The easiest hard problem: Number partitioning. In A.G. Percus, G. Istrate, and C. Moore, editors, *Computational Complexity and Statistical Physics*, pages 125–139, New York, 2006. Oxford University Press.

[79] Tom Milledge, Gaolin Zheng, Tim Mullins, and Giri Narasimhan. SBLAST: Structural basic local alignment searching tools using geometric hashing. In *Bioinformatics and Bioengineering*, pages 1343–1347, 2007.

[80] Caitlyn L. Mills, Penny J. Beuning, and Mary Jo Ondrechen. Biochemical functional predictions for protein structures of unknown or uncertain function. *Computational and Structural Biotechnology Journal*, 13:182 – 191, 2015.

[81] Dariusz Mrozek, Miłosz Brożek, and Bożena Małysiak-Mrozek. Parallel implementation of 3d protein structure similarity searches using a gpu and the cuda. *Journal of Molecular Modeling*, 20(2):1–17, 2014. This work was supported by the European Union through the European Social Fund (grant agreement number: UDA-POKL.04.01.01-00-106/09).

[82] Dariusz Mrozek, Bozena Malysiak-Mrozek, and Artur Klapcinski. Cloud4Psi: cloud computing for 3D protein structure similarity searching. *Bioinformatics*, 30(19):2822–2825, 2014.

[83] H. Nakamura, Berman H. M., and Henrick K. Announcing the worldwide protein data bank. *Nature Structural Biology*, 10:98, 2003.

[84] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[85] Roderic D.M. Page. Space, time, form: viewing the tree of life. *Trends in Ecology & Evolution*, 27:113 – 120, Jan-02-2012 2012.

[86] Shashi B. Pandit and Jeffrey Skolnick. Fr-TM-align: A new protein structural alignment method based on fragment alignments and the TM-score. *BMC Bioinformatics*, 9(1):531, December 2008.

[87] Bin Pang, Nan Zhao, Michela Becchi, Dimitry Korkin, and Chi-Ren Shyu. Accelerating large-scale protein structure alignments with graphics processing units. *BMC Research Notes*, 5, 2011.

[88] J Park, S and M Yamamura. FROG (fitted rotation and orientation of protein structure by means of real-coded genetic algorithm): Asynchronous parallelizing for protein structure-based comparison on the basis of geometrical similarity. *Genome Informatics*, 13:344–45, 2002.

[89] Bryson R. Payne, G. Scott Owen, Irene Weber, Ying Zhu, and Ping Liu. A portable, reusable framework for scientific computing on gpus. In *ACM SIGGRAPH 2004 Posters*, SIGGRAPH '04, page 93, 2004.

[90] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[91] D Pekurovsky, I N Shindyalov, and P E Bourne. A case study of high-throughput biological data processing on parallel platforms. *Bioinformatics*, 20(12):1940–1947, 2004.

[92] David A. Pelta, Juan R. Gonzalez, and Marcos M. Vega. A simple and fast heuristic for protein structure comparison. *BMC Bioinformatics*, 9:161, March 2008.

[93] Robert Clark Penner, Michael Knudsen, Carsten Wiuf, and Jørgen Ellegaard Andersen. An algebro-topological description of protein domain structure. *PLoS ONE*, 6(5):196, 2011.

[94] Matt Pharr and Randima Fernando. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, first edition, 2005.

[95] Aleksandar Poleksic. Algorithms for optimal protein structure alignment. *Bioinformatics*, 25(21):2751–2756, November 2009.

[96] D. M. W. Powers. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[97] Ganesan Pugalenthi, Ke Tang, Pn Suganthan, and Saikat Chakrabarti. Identification of structurally conserved residues of proteins in absence of structural homologs using neural network ensemble. *Bioinformatics*, page 618, November 2008.

[98] V. Pulim, B. Berger, and J. Bienkowska. Optimal contact map alignment of protein-protein interfaces. *Bioinformatics*, 24(20):2324–2328, October 2008.

[99] Stéphanie Pérot, Olivier Sperandio, Maria A. Miteva, Anne-Claude Camproux, and Bruno O. Villoutreix. Druggable pockets and binding site centric chemical space: a paradigm shift in drug discovery. *Drug Discovery Today*, 15(15):656 – 667, 2010.

[100] Thomas Rauber and Gudula Runger. *Parallel Programming - for Multicore and Cluster Systems.* Springer, 2010.

[101] Oliver C. Redfern, Benoît H. Dessailly, Timothy J. Dallman, Ian Sillitoe, and Christine A. Orengo. Flora: A novel method to predict protein function from structure in diverse superfamilies. *PLOS Computational Biology*, 5(8):1–12, 08 2009.

[102] B Rost and Sander C. Prediction of protein secondary structure at better than 70% accuracy. *Journal of Molecular Biology*, 253:584–599, 1993.

[103] S Rufino and T Blundell. Structure-based identification and clustering of protein families and superfamilies. *Computer Aided Mol. Design*, 8:5–27, 1994.

[104] Subhash Saini, Haoqiang Jin, Robert Hood, David Barker, Piyush Mehrotra, and Rupak Biswas. The impact of hyper-threading on processor resource utilization in production applications. In *18th International Conference on High Performance Computing, HiPC 2011, Bengaluru, India, December 18-21, 2011*, pages 1–10, 2011.

[105] Saeed Salem, Mohammed J. Zaki, and Chris Bystroff. FlexSnap: flexible non-sequential protein structure alignment. *Algorithms for molecular biology : AMB*, 5:12, January 2010.

[106] S. Sarkar, T. Majumder, A. Kalyanaraman, and P.P. Pande. Hardware accelerators for biocomputing: A survey. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3789–3792. IEEE, August 2010.

[107] Souradip Sarkar, Gaurav Ramesh Kulkarni, Partha Pratim Pande, and Ananth Kalyanaraman. Network-on-Chip Hardware Accelerators for Biological Sequence Alignment. *IEEE Transaction on Compututation*, 59(1):29–41, January 2010.

[108] John E. Savage and Mohammad Zubair. A unified model for multicore architectures. In *Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*, IFMT '08, pages 9:1–9:12, New York, NY, USA, 2008. ACM.

[109] Bryan Schauer. Multicore processors - A necessity. URL:www.csa.com/discoveryguides/multicore/review.pdf, 2008.

[110] A. A. Shah, G. Folino, and N. Krasnogor. Toward high-throughput, multicriteria protein-structure comparison and analysis. *IEEE Transactions on NanoBioscience*, 9(2):144–155, June 2010.

[111] Azhar Ali Shah. Studies on distributed approaches for large scale multi-criteria protein structure comparison and analysis. Doctoral Thesis, March 2011.

[112] Azhar Ali Shah, Daniel Barthel, and Natalio Krasnogor. Grid and distributed public computing schemes for structural proteomics : A short overview. *Frontiers of High Performance Computing and Networking ISPA 2007 Workshops*, 4743:424–434, 2007.

[113] Azhar Ali Shah, Gianluigi Folino, and Natalio Krasnogor. Toward high-throughput, multicriteria protein-structure comparison and analysis. *IEEE Transactions on NanoBioscience*, 9(2):144–155, 2010.

[114] A. Sharma, A. Papanikolaou, and E.S. Manolakos. Accelerating all-to-all protein structures comparison with tmalign using a noc many-cores processor architecture. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2013 IEEE 27th International*, pages 510–519, May 2013.

[115] Anuj Sharma and Elias S. Manolakos. Efficient multi-criteria protein structure comparison on modern processor architectures. *BioMed Research International*, 2015:13, 2015. Article ID 563674.

[116] Anuj Sharma and Elias S. Manolakos. Multi-criteria protein structure comparison and structural similarities analysis using pymcpsc. *PLOS ONE*, 13(10):1–15, 10 2018.

[117] Maxim Shatsky, Ruth Nussinov, Haim J. Wolfson, and Sackler Insti. FlexProt: alignment of flexible protein structures without a predefinition of hinge regions. *Journal of Computational Biology*, 11:83–106, 2004.

[118] I. N Shindyalov and P. E Bourne. Protein data representation and query using optimized data decomposition. *CABIOS*, 13:487–496, 1998.

[119] I N Shindyalov and P E Bourne. Protein structure alignment by incremental combinatorial extension (CE) of the optimal path. *Protein Engineering*, 11(9):739–747, 1998.

[120] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, John Wiley & Sons, 8 edition, 2008.

[121] Ian Sillitoe, Tony E. Lewis, Alison Cuff, Sayoni Das, Paul Ashford, Natalie L. Dawson, Nicholas Furnham, Roman A. Laskowski, David Lee, Jonathan G. Lees, Sonja Lehtinen, Romain A. Studer, Janet Thornton, and Christine A. Orengo. CATH: comprehensive structural and functional annotations for genome sequences. *Nucleic Acids Research*, 43(D1):D376–D381, January 2015.

[122] W Simon. Transistor Count and Moore's Law. URL:http://en.wikipedia.org/wiki/File%3ATransistor_Count_and_Moore'27s_Law_-_2011.svg, 2011.

[123] Manfred J. Sippl and Markus Wiederstein. A note on difficult structure alignment problems. *Bioinformatics*, 24(3):426–427, February 2008.

[124] Alex D Stivala, Peter J Stuckey, and Anthony I Wirth. Fast and accurate protein substructure searching with simulated annealing and GPUs. *BMC Bioinformatics*, 11(1):446, 2010.

[125] Michael L Stokes. A brief look at FPGAs, GPUs and Cell Processors. *International Test and Evaluation Association (ITEA) Journal*, (7):9–11, 2007.

[126] Jeet Sukumaran and Mark T. Holder. Dendropy: a python library for phylogenetic computing. *Bioinformatics*, 26(12):1569, 2010.

[127] W R Taylor. A flexible method to align large numbers of biological sequences. *Journal of Molecular Evolution*, 28(1-2):161–169, 1998.

[128] Ella Teplitsky, Karan Joshi, Daniel L. Ericson, Alexander Scalia, Jeffrey D. Mullen, Robert M. Sweet, and Alexei S. Soares. High throughput screening using acoustic droplet ejection to combine protein crystals and chemical libraries on crystallization plates at high density. *Journal of Structural Biology*, (0):–, 2015.

[129] Douglas L. Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. *Acta Crystallographica Section A*, 61(4):478–480, July 2005.

[130] W. S. Torgerson. Multidimensional scaling: I. theory and method. *Psychometrika*, 17:401–419, 1952.

[131] Ehsan Totoni, Babak Behzad, Swapnil Ghike, and Josep Torrellas. Comparing the power and performance of intel's scc to state-of-the-art cpus and gpus. In Rajeev Balasubramonian and Vijayalakshmi Srinivasan, editors, *ISPASS*, pages 78–87. IEEE, 2012.

[132] Guido Van Rossum. Python tutorial, Technical Report CS-R9526. Technical report, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.

[133] Mallika Veeramalai and David Gilbert. A novel method for comparing topological models of protein structures enhanced with ligand information. *Bioinformatics*, 24:2698–2705, December 2008.

[134] Mallika Veeramalai, David Gilbert, and Gabriel Valiente. An optimized TOPS+ comparison method for enhanced TOPS models. *BMC Bioinformatics*, 11(1):138, 2010.

[135] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, 13(2):22–30, March 2011.

[136] Yuhang Wang, Fillia Makedon, James Ford, and Heng Huang. A bipartite graph matching framework for finding correspondences between structural elements in two proteins. *Proceedings of the International Conference of IEEE Engineering in Medicine and Biology Society*, 4:2972–2975, 2004.

[137] Michael Waskom, Olga Botvinnik, Paul Hobson, John B. Cole, Yaroslav Halchenko, Stephan Hoyer, Alistair Miles, Tom Augspurger, Tal Yarkoni, Tobias Megies, Luis Pedro Coelho, Daniel Wehner, cynddl, Erik Ziegler, diego0020, Yury V. Zaytsev, Travis Hoppe, Skipper Seabold, Phillip Cloud, Miikka Koskinen, Kyle Meyer, Adel Qalieh, and Dan Allan. seaborn: v0.5.0 (november 2014), November 2014.

[138] A Williams, D Gilbert, and D R Westhead. Multiple structural alignment for distantly related all beta structures using TOPS pattern discovery and simulated annealing. *Protein Engineering*, 16(12):913–923, 2009.

[139] W. Xie and N. V. Sahinidis. A reduction-based exact algorithm for the contact map overlap problem. *Journal of computational biology : a journal of computational molecular cell biology*, 14(5):637–654, June 2007.

[140] Yuzhen Ye and Adam Godzik. Multiple flexible structure alignment using partial order graphs. *Bioinformatics*, 21(10):2362–2369, 2005.

[141] Golan Yona and Klara Kedem. The URMS-RMS hybrid algorithm for fast and sensitive local protein structure alignment. *Journal of Computational Biology*, 12(1):12–32, 2005.

[142] Lei Zhang, James Bailey, Arun S. Konagurthu, and Kotagiri Ramamohanarao. A fast indexing approach for protein structure comparison. *BMC Bioinformatics*, 11 Suppl 1:46, 2010.

[143] Yang Zhang and Jeffrey Skolnick. TM-align: a protein structure alignment algorithm based on the TM-score. *Nucleic Acids Research*, 33(7):2302–2309, 2005.

[144] Zhiyu Zhao, Bin Fu, Francisco J Alanis, and Christopher M Summa. Feedback algorithm and webserver for protein structure alignment. *Computational Systems Bioinformatics Life Sciences Society Computational Systems Bioinformatics Conference*, 7:109–120, 2008.

[145] Degui Zhi, S Sri Krishna, Haibo Cao, Pavel Pevzner, and Adam Godzik. Representing and comparing protein structures as paths in three-dimensional space. *BMC Bioinformatics*, 7:460, 2006.

[146] Xiaobo Zhou, James Chou, and Stephen Tc Wong. Protein structure similarity from principle component correlation analysis. *BMC Bioinformatics*, 7(1):40, 2006.

[147] Jianhua Zhu and Zhiping Weng. FAST: a novel protein structure alignment algorithm. *Proteins*, 58(3):618–627, 2005.

[148] Quan Zou, Xu-Bin Li, Wen-Rui Jiang, Zi-Yu Lin, Gui-Lin Li, and Ke Chen. Survey of MapReduce frame operation in bioinformatics. *Briefings in Bioinformatics*, 2013.