ΕΡΓΑΣΙΑ

ΣΤΟ ΜΑΘΗΜΑ ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ

Σκοπός της έκθεσης μου είναι η αναλυτική παρουσίαση της εργασίας μου. Παραδίδω το jacobi_parallelMPI.c το οποίο αποτελεί την παραληλλοποιημένη έκδοση του jacobi_serial.c που μας δώσατε. Παραδίδω και ένα ακόμη pdf με τις μετρήσεις μου πάνω στο πρόγραμμα.

Αρχικά διαβάζονται οι παράμετροι της εκτέλεσης από το αρχείο εισόδου από την root διεργασία η οποία και τις κάνει broadcast σε όλες τις υπόλοιπες. Έπειτα δημιουργώ ένα δισδιάστατο καρτεσιανό communicator για να ορίσω τη θέση κάθε διεργασίας στο χάρτη και ποιο μέρος του πίνακα θα πάρει. Η κάθε διεργασία μαθαίνει ποιοι είναι οι γείτονες της, όσους έχει. Έπειτα η κάθε διεργασία δημιουργεί δυναμικά τους πίνακες u_old και u, με μέγεθος (n / sqrt(procs) +2)*(m / sqrt(procs) +2), γιατί ο αρχικός τετραγωνικός πίνακας θα χρειαστεί να μοιραστεί ισομερώς σε n=*procs* πλήθος πινάκων οι οποίοι έχουν και γύρω γύρω άλω. Έπειτα δημιουργώ τα datatypes που χρειάζονται για την επικοινωνία μεταξύ των διεργασιών (row & column για την άλω αλλά και δύο ακόμα τύπους οι οποίοι είναι αναγκαίοι για τον σωστό διαμοιρασμό του αρχικού πίνακα u_root στις διεργασίες). Έπειτα γίνεται Scatter του u_root πίνακα (καλόντας την MPI_Scatterv). Ύστερα, αφού μας είναι αναγκαίο το persistent communication λόγω της φύσης του προβλήματος, έξω απο την κεντρική while επανάληψη κάνω τις κατάλληλες κλήσεις MPI_Recv_init και MPI_Send_init.

Μέσα στην κεντρική επανάληψη,
Όταν δεν γίνεται έλεγχος σύγκλισης,
γίνεται προφανώς το exchange των halos με τον τρόπο που μας έχετε υποδείξει, και έπειτα υπολόγιζεται το residual με την one_jacobi_iteration (η οποία έχει παραμείνει ίδια και απαράλλαχτη, εκτός της τιμής που επιστρέφει , πλεόν επιστρέφει σκέτο το error και η συνάρτηση που είχε υπολογίζεται εκτός, καθώς υπήρχε απώλεια δεδομένων λόγω της μεγάλης ακρίβειας που απαιτούν τα δεδομένα, άρα γίνεται μια και καλή στο τέλος). Γίνεται έλεγχος σύγκλισης αλλά μόνο τοπικά.
(αν η sqrt(local_error)/(n/sqrt(comm_sz)*m/sqrt(comm_sz)) > maxAcceptableError , θα κοπεί το while πριν τις 50 επαναλήψεις, κάτι που προφανώς δεν θα γίνει)

Όταν γίνεται έλεγχος σύγκλισης,
αυτό που αλλάζει είναι ότι καλείται η MPI_Allreduce και υπολογίζεται το global error σε κάθε επανάληψη . (αν η sqrt(global_error)/(n*m) > maxAcceptableError , θα κοπεί το while πριν τις 50 επαναλήψεις, κάτι που δεν γίνεται ποτέ για τα μεγέθη δεδομένων που μπορούμε να τρέξουμε).
Τέλος, υπολογίζουμε το error με την checksolution η οποία έχει μείνει ίδια και αυτή εκτός απο την μικρή αλλαγή στην τιμή που επιστρέφει όπως ακριβώς έγινε και για την one_jacobi_iteration, για τους ίδιους λόγους. Έχω φτιάξει κι άλλον ένα τρόπο για να υπολογίζεται το error, ο οποίος είναι να γίνει gather των u_old της κάθε διεργασίας σε έναν μέγαλο πίνακα και να υλοποιηθεί η check solution σε αυτόν, αλλά απεδείχθη πιο χρονοβόρο οπότε το *σχολίασα*. (//)

Τέλος, γίνονται free τα datatypes που δημιούργησα και οι πίνακες που δεσμεύθηκαν δυναμικά.
Το πρόγραμμα που δημιούργησα δεν μπορεί να δουλέψει με 80 διεργασίες (προσπάθησα να το κάνω να λειτουργήσει αλλά χωρίς αποτέλεσμα) καθώς δεν μπορεί να διαχωρίσει ισομερώς τον τετραγωνικό πίνακα σε 80 κομμάτια, αφού όπως έδειξα πιο πάνω χρησιμοποιεί την τετραγωνική ρίζα του πλήθους των διεργασιών για να διαχωρίσει ισομερώς τον πίνακα. Ακέραιη τετραγωνική ρίζα του 80 δεν υπάρχει, αλλά υπάρχει ακέραιη τετραγωνική ρίζα του 81, οπότε αντι για 80 διεργασίες χρησιμοποιώ 81. Αυτό προφανώς δεν λειτουργεί και τόσο καλά, ιδιαίτερα για τα μικρά μεγέθη δεδομένων. Ωστόσο όσο αυξάνεται το μέγεθος των δεδομένων τόσο παρατηρούμε ότι αυξάνεται και το efficiency του προγράμματος μου για 81 διεργασίες και συγκλίνει κάπου, περισσότερα για αυτό αργότερα.

Για αυτό τον λόγο έχω μετρήσει και το *challenge* και για 81 διεργασίες, για να υπάρχει κάποια ομοιομορφία στην σύγκριση. Όπως είναι λογικό ούτε το challenge έχει καλό efficiency για 81 διεργασίες, μάλιστα τα πάει αρκετά χειρότερα από το δικό μου πρόγραμμα, το jacobi_parallelMPI.c
Για τους παραπάνω λόγους δεν θα λάβω υπόψιν μου και τόσο αυτές τις στήλες (για 81 διεργασίες) στην μελέτη που θα κάνω για την κλιμάκωση.

Θα ήθελα επίσης να σημειώσω ότι για μέγεθος αρχικού πίνακα 53760x53760 είχα error από την calloc, κάτι που προσπάθησα να επιλύσω με διαφορετική υλοποίηση, αλλά δεν λειτούργησε. Έτσι, το πρόγραμμα μου πάνω στην ΑΡΓΩ τρέχει για μέγεθος πίνακα μέχρι και 26880x26880.

Πριν ξεκινήσω θα ήθελα να αναφέρω ότι, μετά απο πειραματικές εκτελέσεις, τις μετρήσεις μου τις έχω κάνει με τις εξής παρακάτω παραμέτρους στο PBS.
Για (N = 4) #PBS -l select=1:ncpus=4:mpiprocs=4:mem=16400000kb
Για ($9 \leq N \leq 64$) #PBS -l select=$\sqrt{N}$:ncpus=$\sqrt{N}$:mpiprocs=$\sqrt{N}$:mem=16400000kb
Για (N = 9) #PBS -l select=9:ncpus=1:mpiprocs=9:mem=16400000kb

Μελετώντας τις μετρήσεις μου για το εκτελέσιμο challenge που μας δώσατε, βγάζω το πόρισμα ότι το πρόγραμμα δεν κλιμακώνει καθόλου καλά, δεν είναι καν weakly scalable, αφού όσο κι αν αυξήσω το μέγεθος δεδομένων το efficiency παραμένει σταθερό με πολύ μικρές αποκλίσεις (< 0.1) για συσγκεκριμένο πλήθος διεργασιών. Επίσης όσο αυξάνεται το πλήθος των διεργασιών το efficiency τείνει προς το 0 (χωρίς να το αγγίζει ποτέ για το maximum των 80 διεργασιών που μελετάμε). Παράτυπα έχω κάνει μετρήσεις και για 81 διεργασίες, όπως είπα παραπάνω αφού το δικό μου πρόγραμμα τρέχει για 81 και όχι για 80 για λόγους που εξηγώ παραπάνω. Οι μετρήσεις του challenge για 81 διεργασίες έχουν γίνει καθαρά για λόγους ομοιομορφίας και δεν θα τους σχολιάσω κάθως όπως είναι λογικό το efficiency του προγράμματος εκεί πατώνει ακόμα περισσότερο. Το challenge δεν κλιμακώνει καλά επειδή χρησιμοποιήθηκε η μεθοδολογία του Foster για την υλοποίηση του.
Η μεθοδολογία του Foster αποτελείται από τα εξής βήματα.
1. Partitioning. Διαχωρίζουμε τις διεργασίες και τα δεδομένα σε μικρότερες διεργασίες με στόχο να βρούμε τις διεργασίες που μπορούν να εκτελεστούν παράλληλα.
2. Communication. Ορίζουμε το σύνολο της επικοινωνίας που θα υπάρξει μεταξύ των Cpus. 3. Aggregation. Συνδυασμός των 2 παραπάνω βημάτων σε μεγαλύτερες διεργασίες.
4. Mapping. Ορίζουμε τις διεργασίες από τα προηγούμενα σε κάθε Cpu με στόχο την ελαχιστοποίηση της επικοινωνίας και το βέλτιστο Load Balancing.
Η αποτυχία του challenge στο να κλιμακώσει καλά έγκειται στο ότι δεν έχει γίνει σωστός διαμερισμός των δεδομένων ανα τις διεργασίες για τη φύση του συγκεκριμένου προβλήματος. Είναι προφανές διαισθητικά ότι για την διαδικασία που εκτελεί ο αλγόριθμος - υπολογισμός του stencil - ο πιο αποδοτικός διαμερισμός του πίνακα είναι σε κύβους και όχι σε λωρίδες-σειρές. Με τον τρόπο που χρησιμοποιεί το challenge ξοδεύει πολύ χρόνο σε επικοινωνία αφού όσο κι αν μεγαλώσει ο αριθμός των διεργασιών, η επικοινωνία παραμένει ακριβώς ίδια. Κάθε διεργασία έχει να στείλει και να παραλάβει 2*N floats (2 rows)(όπου N*N το μέγεθος του πίνακα), ανεξαρτήτως του αριθμού των διεργασιών, κάτι που τελικά καταλήγει ασφυκτικό όσο μεγαλώνει το πλήθος των διεργασιών αφού πλέον πολύ περισσότερος χρόνος ξοδεύεται σε επικοινωνία, σε κάθε διεργασία, από ότι θα πρεπε και οι πράξεις και οι υπολογισμοί είναι αναλογικά λίγότεροι.

Ο διαμερισμός του άρχικου πίνακα σε τετράγωνα και όχι σε λωρίδες, διαισθητικά μοιάζει πολύ πιο ταιριαστός στη φύση του προβλήματος, κάτι που αποδεικνύεται και πειραματικά με τις μετρήσεις που έχω κάνει στο πρόγραμμα που ανέπτυξα με τη συγκεκριμένη λογική όσον αφορά τον διαμερισμό των δεδομένων. Στο αρχικό τμήμα της έκθεσης εξηγώ πως έχω υλοποιήσει το δικό μου παράλληλο πρόγραμμα (παραλληλοποίηση του ακολουθιακού jacobi_serial.c), το jacobi_parallelMPI.c . Έχω

μετρήσεις για δύο παραλαγές όπως ζητείται στην εκφώνηση, μια *χωρίς έλεγχο σύγκλισης* και μια *με έλεγχο σύγκλισης σε κάθε βήμα της επανάληψης*.

Ξεκινώντας την μελέτη μου για το jacobi_parallelMPI.c , θα σχολιάσω την υλοποίηση *χωρίς έλεγχο σύγκλισης*. Παρατηρώ ότι, όπως ήταν αναμενόμενο, έχει πολύ καλύτερη συμπεριφορά από το challenge και το efficiency του είναι απρόσμενα ικανοποιητικό, αφού βρίσκεται σχεδόν μόνιμα, ανεξαρτήτως πλήθους διεργασιών και μεγέθους δεδομένων, πάνω από 0.9 (με μικρές εξαιρέσεις) . Πιο συγκεκριμένα, για να μιλήσω με όρους του βιβλίου, αφού ενώ αυξάνεται το πλήθος των διεργασιών, διατηρείται το efficiency σταθερό με πολυ μικρές αποκλίσεις (< 0.05) χωρίς να αυξηθεί το μέγεθος των δεδομένων μπορώ να χαρακτηρίσω το πρόγραμμα ως ισχυρά κλιμακούμενο (strongly scalable). Αυτό βεβαιώνω ότι ισχύει μέχρι και το μέγεθος δεδομένων που έχω μετρήσει και καταγράψει, και είμαι πολύ βέβαιος ότι ισχύει και για 80 διεργασίες (παρόλο που δεν μπόρεσα να το υλοποιήσω με αυτό το πλήθος λόγω τεχνικών θεμάτων). Ωστόσο, αυτό δεν ισχύει για πλήθος διεργασιων = 81, για τον απλό λόγο ότι η ρίζα του 81 δεν είναι ακριβές πολλαπλάσιο του 840 και συνεπώς όλων των υπολοίπων μεγεθών δεδομένων. Αλλά το χρησιμοποιώ επειδή έχει ακέραια ρίζα για λόγους που εξηγώ παραπάνω, αφού δεν χώρισα τον πίνακα σε 80 διεργασίες (που δεν έχει ακέραια ρίζα). Παρόλα αυτά θα ήθελα να τονίσω ότι ακόμη και για πλήθος διεργασιών = 81, παρατηρούμε ότι όσο αυξάνεται το μέγεθος δεδομένων τόσο πιο κοντά τείνει στο average efficiency του προγράμματος το οποίο κυμαίνεται περίπου στο διάστημα **[** *0.9* **,** *1* **]** . Επίσης, θα ήθελα να προσθέσω, ότι στον πίνακα των μετρήσεων μου, παρατηρώ ότι για μικρά μεγέθη δεδομένων (840x840 & 1680x1680) το efficiency μου σε μεγάλο πλήθος διεργασιών πέφτει ελάχιστα σταδιακά, αλλά με μη ανησυχητικό ρυθμό. Αυτό θα το χαρακτηριζά λογικό, και βασίζω αυτή τη σκέψη στον νόμο του Amdahl που ουσιαστικά μας λέει ότι υπάρχει ένα όριο στο βαθμό που μπορεί να παραλληλοποιηθεί ένα πρόγραμμα (για συγκεκριμένο μέγεθος δεδομένων κάθε φορά). Κάτι αναπάντεχο που παρατήρησα στις μετρήσεις μου για μέγεθος δεδομένων 840x840, είναι ότι ενώ έχω καθοδική πορεία όσο αυξάνεται το πλήθος διεργασιών (με χαμηλό ρυθμό), βλέπω ότι για πλήθος διεργασιών = 49 έχω superlinear speedup με efficiency = **1.19!** (δεν είναι λάθος η μέτρηση, την έκανα πολλές φορές) και έπειτα για 64 διεργασίες το efficiency πέφτει στο 0.98 που και πάλι είναι πάρα πολύ ψηλά. Αυτό με οδηγεί στο συμπέρασμα ότι όσο αυξάνονται οι οι πυρήνες στους οποίους τρέχει το πρόγραμμα έχουμε μεγαλύτερο μέγεθος μνήμης cache διαθέσιμο, άρα γίνονται λιγότερα reads και writes από την RAM, και από ότι φαίνεται για το μέγεθος 840x840 η χρυσή τομή είναι οι 49 διεργασίες (που τρέχουν σε 7 κόμβους και 7 επεξεργαστές) όπου μάλλον έκει χωράει ολόκληρο το πρόγραμμα στην cache και γινεται και ο σωστότερος διαμοιρασμός όσον αφορά την μεγιστοποίηση της αποδοτικότητας. Προσπερνάω τελείως τις μετρήσεις για 81 διεργασίες.

Τώρα όσον αφορά την υλοποίηση του jacobi_parallelMPI.c με έλεγχο σύγκλισης, όλα παραμένουν ίδια εκτός της μικρής διαφοράς, όπου υλοποιούμε τον συνολικό έλεγχο σύγκλισης στο κεντρικό while loop. Για τα μεγέθη δεδομένων που μπορεί το πρόγραμμα μου να χειριστεί, δηλαδή μέχρι και 26880x26880, πριν να εμφανίσει η calloc error, ο έλεγχος σύγκλισης δεν θα σταματήσει ποτέ την κεντρική επανάληψη πριν να ολοκληρωθούν οι 50 επαναλήψεις. Άρα, η υλοποίηση του jacobi_parallelMPI.c *με έλεγχο σύγκλισης σε κάθε βήμα της επανάληψης*, για το δικό μου πρόγραμμα, δεν κάνει τίποτα παραπάνω από το να προσθέτει ένα μηδαμινό χρονικό πρόστιμο. Έχω καταγράψει τις μετρήσεις μου πάνω σε αυτό αλλά δεν έχω υπολογίσει speedup και efficiency, μιας και είναι πολύ παρόμοια με αυτά του jacobi_parallelMPI.c χωρίς έλεγχο σύγκλισης, αλλά και επειδή δεν έχω λόγο να επεκταθώ περισσότερο στη μελέτη αυτής της υλοποίησης αφού απεδείχθη άκαρπη.