ΠΑΡΑΛΛΗΛΗ ΕΠΕΞΕΡΓΑΣΙΑ

ΠΑΡΑΛΛΗΛΗ ΟΛΙΚΗ ΒΕΛΤΙΣΤΟΠΟΙΗΣΗ

Όνομα/Επώνυμο: Κωνσταντίνος Καραΐσκος Α.Μ.: 1072636 Έτος: 5ο

24/6/2024

ΕΙΣΑΓΩΓΗ:

Τα προγράμματα γράφτηκαν και εκτελέστηκαν στον server της σχολής:
falcon.ceid.upatras.gr.

Πληροφορίες επεξεργαστή (lspci | grep –i vga):

AMD EPYC family 23 model 1
#CPUs: 2
#Threads/core: 1
Clock: 2195.874 MHz
L1d cache: 32K
L1i cache: 64K
L2 cache: 512K
L3 cache: 8192K

RAM (free –h):

	total	used	free	shared	buff/cache	available
Mem:	7.5Gi	672Mi	1.8Gi	384Mi	5.0Gi	6.2Gi
Swap:	4.0Gi	0.0Ki	4.0Gi			

Άρα, περίπου 8.05 Gb διαθέσιμη RAM συνολικά.

Το Makefile περιέχει τις εντολές για το compile των προγραμμάτων και την δημιουργία των εκτελέσιμων αρχείων, καθώς και τη δυνατότητα διαγραφής των δημιουργηθέντων εκτελέσιμων.

ΑΝΑΛΥΣΗ ΥΛΟΠΟΙΗΣΕΩΝ:

OMP:

Καθώς διαμοιράζουμε τον αριθμό των επαναλήψεων σε threads χρειάστηκε να ορίσουμε κάποιες από τις υπάρχουσες ή και να δημιουργήσουμε καινούργιες private μεταβλητές, στις οποίες θα αποθηκεύονται τα εκάστοτε αποτελέσματα κάθε thread και στο τέλος να συγκεντρώσουμε τα καλύτερα εξ αυτών σε shared μεταβλητές. Προκειμένου να αποφύγουμε διάφορα race-conditions, η ανανέωση των shared μεταβλητών έγινε με τη χρήση μίας critical region.

Αρχικά, αντικαταστήσαμε την srand48() με την thread safe function erand48(), η οποία παίρνει ως όρισμα την τιμή του randBuffer[] η τιμή του οποίου διαφέρει από thread σε thread.

Καθώς η μεταβλητή funevals θα αυξάνονταν από κάθε thread, ορίσαμε μία νέα τοπική μεταβλητή για κάθε thread, την loc_funevals*, η οποία θα συγκεντρώνει τον συνολικό αριθμό κλήσεων της συνάρτησης f() του εκάστοτε thread και στο τέλος συγκεντρώνουμε την τιμή όλων των loc_funevals στην funevals, που θα περιέχει το συνολικό αριθμό κλήσεων της f(). Θα μπορούσαμε να περικλείσουμε την “funevals++;” σε μία “ #pragma omp atomic “, αλλά αυτό λόγω του ότι χρησιμοποιεί ένα mutex για αμοιβαίο αποκλεισμό, θα αύξανε τον χρόνο εκτέλεσης.

Αρχικοποιήσαμε την παράλληλη περιοχή πριν την for() που μας ενδιαφέρει προκειμένου να ορίσουμε κάποιες νέες τοπικές μεταβλητές για κάθε thread. Συγκεκριμένα ορίσαμε τις τοπικές για κάθε thread μεταβλητές, local_fx, local_nt, local_nf και loc_funevals την οποία αναφέραμε πριν. Οι τρείς πρώτες θα χρησιμοποιούνται για την αποθήκευση τιμών σε κάθε trial. Ακόμα οι τοπικές μεταβλητές “loc_best_pt, loc_best_fx, loc_best_trial, loc_best_nt και loc_best_nf” θα αποθηκεύουν τις αντίσοιχες τιμές για το καλύτερο αποτέλεσμα της f(), για κάθε thread.

Ο βρόγχος των trials παραλληλοποιήθηκε με static scheduling, καθώς οι επαναλήψεις μεταξύ τους έχουν μικρή διαφορά ως προς το χρόνο τρεξίματος και ορίσαμε nowait καθώς δεν υπάρχει λόγος για το implicit barrier στο τέλος του. Μέσα στη for() τρέχει ο mds και γίνεται η ανανέωση των “loc_*” μεταβλητών κάθε φορά που υπάρχει κάποιο καλύτερο αποτέλεσμα.

Τέλος, γίνεται η ενημέρωση των best_fx, best_trial, best_nt, best_nf, best_pt και funevals μέσα στην κρίσιμη περιοχή. Κάθε thread ελέγχει αν το τοπικό του ελάχιστο είναι καλύτερο από το τωρινό ολικό ελάχιστο και αν ναι ανανεώνει τις παραπάνω τιμές και έπειτα προσθέτει τον δικό του αριθμό εκτελέσεων την f() στον ολικό αριθμό εκτελέσεών της. Έπειτα, αποθηκεύουμε τα αποτελέσματα σε ένα αρχείο για να μπορούμε να τα κάνουμε plot στη συνέχεια.

* Επίσης εισάγαμε και το όρισμα “unsigned long *loc_funevals” στις συναρτήσεις mds() και f(), ώστε να μπορούμε να δώσουμε ως όρισμα την τοπική loc_funevals κάθε thread, για να ανανεωθεί.

OMP TASKS:

Καθώς έχουμε παραλληλοποιήσει ήδη τη main() στο multistart_mds_omp.c, προκειμένου να αξιοποιήσουμε και τον παραλληλισμό εντός της μεθόδου mds, χρησιμοποιούμε tasks. Συγκεκριμένα, μετά την αρχικοποίηση του simplex (initialize_simplex(u,n,point,delta)), ορίζουμε την παράλληλη περιοχή, μέσα στην οποία ορίζουμε δυο single περιοχές οι οποίες θα εκτελεστούν από ένα μόνο thread, το οποίο θα παράγει, όπου χρειάζεται, tasks τα οποία θα εκτελούν τα υπόλοιπα threads.

Η πρώτη single περιοχή είναι η πρώτη for() η οποία εκτελείται από ένα μόνο thread και παράγει ένα task για κάθε επανάληψή της, μέσα στο οποίο γίνεται μία κλήση της f() και ανανεώνεται η τιμή της "nf" η οποία βρίσκεται σε περιοχή atomic προκειμένου να αποφύγουμε race-conditions. Ανάμεσα από την πρώτη και τη δεύτερη single περιοχή ορίζουμε ένα "#pragma omp taskwait" και "#pragma omp barrier" προκειμένου να είμαστε σίγουροι ότι πριν προχωρήσουμε θα έχουν τελειώσει όλα τα tasks γιατί χρειαζόμαστε την "fu" παρακάτω και όλα τα threads θα βρίσκονται στο ίδιο σημείο.

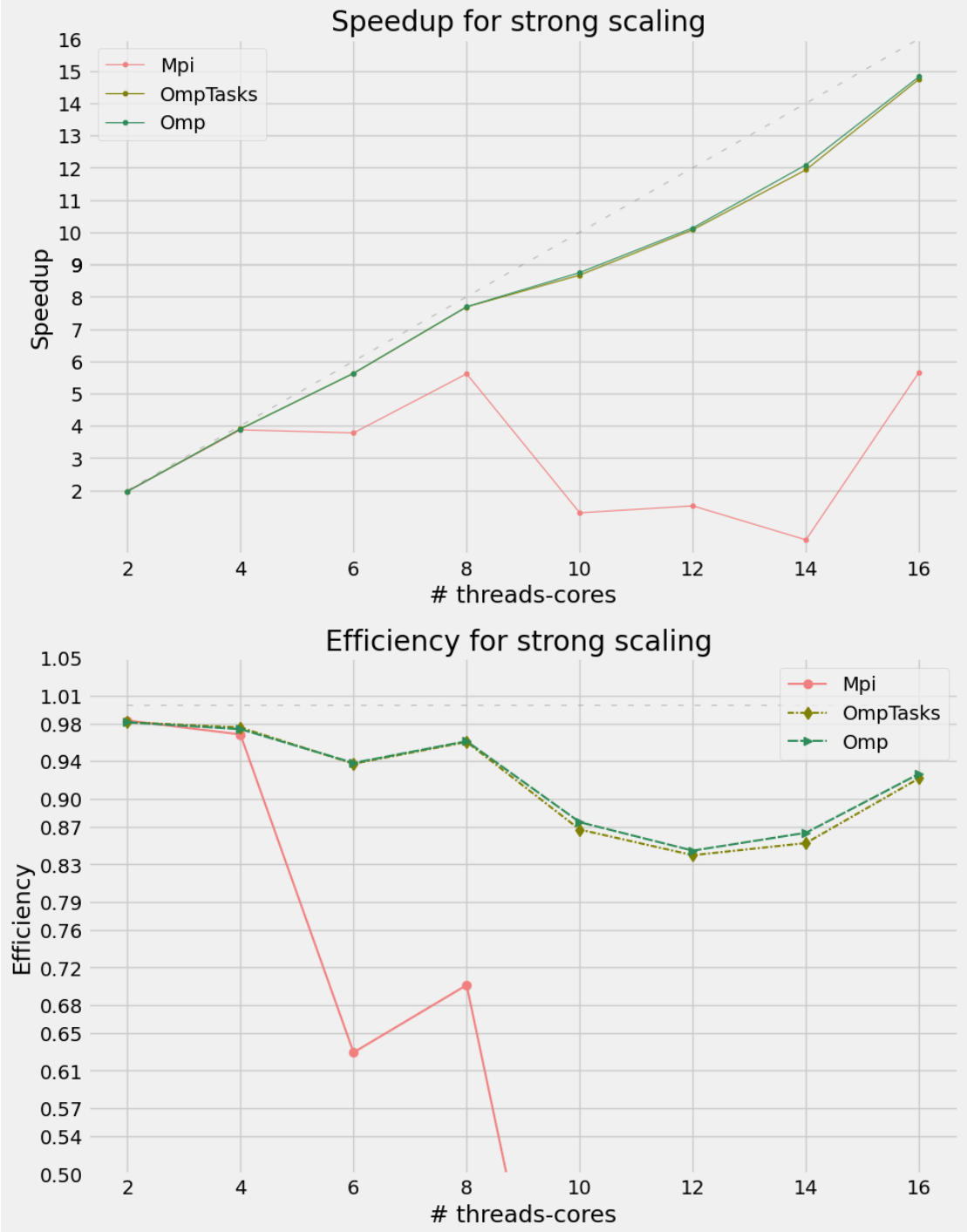
Η δεύτερη single περιοχή είναι ο βρόγχος while() μέσα στον οποίο γίνονται οι διάφοροι υπολογισμοί της mds προκειμένου να βρεθεί η ζητούμενη τιμή της f(). Και πάλι, το thread που εκτελεί τη single περιοχή, παράγει tasks στα οποία γίνεται κλήση της f και γίνεται χρήση της "#pragma omp atomic" για την ανανέωση μεταβλητών οι οποίες μπορεί να προκαλέσουν race-conditions. Και πάλι μετά από κάθε for() στο οποίο είχαν παραχθεί tasks γίνεται χρήση της "#pragma omp taskwait" προκειμένου να είμαστε σίγουροι ότι πριν προχωρήσουμε, θα έχουν ολοκληρωθεί όλα τα tasks που έχουν παραχθεί. Τέλος, έξω από το βρόγχο while(), γίνεται χρήση για ακόμα μία φορά των "#pragma omp taskwait" και "#pragma omp barrier" για τους ίδιους λόγους με πριν.

MPI:

Αρχικοποιούμε το περιβάλλον του MPI και τις μεταβλητές κάθε διεργασίας. Χρησιμοποιούμε την "MPI_Barrier(MPI_COMM_WORLD)" ώστε να συγχρονιστούν όλες οι διεργασίες και να αρχίσουμε τον timer. Κάθε διεργασία εκτελεί έναν αριθμό από trials για να βρει το τοπικό ελάχιστό της το οποίο αποθηκεύει στις τοπικές μεταβλητές της. Μετά την εκτέλεση όλων των trials εκτελούμε δύο "MPI_Allreduce", μία για να βρούμε το ολικό ελάχιστο μεταξύ όλων των διεργασιών το οποίο αποθηκεύουμε στην μεταβλητή "global_best" και μία για να βρούμε τον συνολικό αριθμό κλήσεων της f() που αποθηκεύουμε στη μεταβλητή "funevals". Τέλος, η διεργασία η οποία είχε βρει το ολικό μέγιστο αποθηκεύει τα αποτελέσματα σε ένα αρχείο.

ΑΠΟΤΕΛΕΣΜΑΤΑ:

Τρόπος Παραλληλοποίησης	Threads/ Processes	Time	Best_fx	Funevals	Speedup
Serial	1	36.719s	3.3360207e-03	630552	-
Omp	2	18.708s	3.3360207e-03	630556	1.9627
	4	9.424s	3.3360207e-03	630560	3.8963
	6	6.524s	3.3360207e-03	630556	5.6282
	8	4.775s	3.3360207e-03	630556	7.6898
	10	4.197s	3.0304832e-03	630556	8.7488
	12	3.623s	3.0304832e-03	630552	10.1349
	14	3.037s	3.0304832e-03	630548	12.0905
	16	2.477s	3.0304832e-03	630548	14.8239
Omp Tasks	2	18.705s	3.3360207e-03	630556	1.9630
	4	9.407s	3.3360207e-03	630560	3.9033
	6	6.530s	3.3360207e-03	630552	5.6231
	8	4.779s	3.3360207e-03	630556	7.6834
	10	4.235s	3.0304832e-03	630556	8.6703
	12	3.644s	3.0304832e-03	630552	10.0765
	14	3.076s	3.0304832e-03	630548	11.9372
	16	2.490s	3.0304832e-03	630548	14.7465
Mpi	2	18.672s	3.3360207e-03	630556	1.9665
	4	9.479s	3.3360207e-03	630556	3.8737
	6	9.723s	3.3360207e-03	630560	3.7765
	8	6.545s	3.3360207e-03	630556	5.6102
	10	28.254s	3.0304832e-03	630556	1.2996
	12	24.245s	3.0304832e-03	630552	1.5144
	14	58.568s	3.0304832e-03	630548	0.6269
	16	6.488s	3.0304832e-03	630548	5.6595


Speedup for strong scaling
Mpi
OmpTasks
Omp
Speedup
threads-cores
Efficiency for strong scaling
Mpi
OmpTasks
Omp
Efficiency
threads-cores

ΣΥΜΠΕΡΑΣΜΑΤΑ:

Μέχρι και για 8 threads η απόδοση της παραλληλοποίησης με OMP και OMP+Tasks αυξάνεται όπως και στο ιδανικό. Μετά, η απόδοση μειώνεται λόγω χρήσης λογικών πυρήνων. Τόσο το OMP όσο και το OMP+Tasks κινούνται περίπου στην ίδια απόδοση, γεγονός το οποίο μπορεί να οφείλεται στη συγκεκριμένη περίπτωση επειδή ο αριθμός των επαναλήψεων είναι σχετικά μικρός και έτσι δεν μπορούμε να δούμε με μεγαλύτερη ακρίβεια την αναμενόμενη καλύτερη απόδοση των tasks, καθώς αυτά είναι πιο lightweight σε σχέση με τα threads.

Στη συγκεκριμένη περίπτωση, η υλοποίηση με MPI εμφανίζει αρκετά κακή απόδοση για 8 processes και πάνω, το οποίο μπορεί να οφείλεται στην έλλειψη επεξεργαστικής ισχύος που διαθέτουμε ή και σε κάποιο λάθος μας, γενικά όμως το MPI χρησιμοποιείται σε κατανεμημένα συστήματα, οπότε μπορεί να συμβάλει και το συγκεκριμένο στα αποτελέσματά μας.