Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Τμήμα Πληροφορικής & Τηλεπικοινωνιών

Εργασία σχεδιασμού, ανάπτυξης και αξιολόγησης παράλληλων προγραμμάτων σε MPI, Υβριδικό MPI+OpenMP, CUDA που υλοποιούν την jacobi method with successive over-relaxation

Παράλληλα Συστήματα - Μ127

Σεπτέμβριος 2022

Πέτρος Παθούλας (cs3190005), petros.pathoulas@di.uoa.gr
Ανδρέας Πατεράκης (ic1200015), ic1200015@di.uoa.gr

Εισαγωγή

Η εργασία έχει σαν στόχο την εξοικείωση των φοιτητών στην παραλληλοποίηση προγραμμάτων σε γλώσσα C με τις προγραμματιστικές διεπαφές MPI, OpenMP και CUDA. Το πρόγραμμα που επεξεργαζόμαστε και μελετάμε μας δίνεται έτοιμο και αφορά τον υπολογισμό της εξίσωσης poisson με τη μέθοδο Jacobi. Η μελέτη έγινε στο cluster ΑΡΓΩ, ο οποίος έχει σύνολο 11 κόμβους σε rack, παραδίδοντας στη διάθεση μας το σύνολο των 80 πυρήνων και των 2 GPU.

Σε πρώτο στάδιο, ζητείται μία βελτιωμένη εκδοχή του ακολουθιακού προγράμματος που δίνεται και η προετοιμασία του για παραλληλοποίηση τις διεπαφές που αναφέρθηκαν. Στη συνέχεια, παραθέτουμε την υλοποίηση του προγράμματος μαζί με τις μελέτες κλιμάκωσης και σχεδιασμού με τη διεπαφή MPI. Λόγω κάποιων τεχνικών προβλημάτων οι μόνες ολοκληρωμένες μελέτες που παραδίδονται είναι αυτή των βελτιστοποιήσεων στο σειριακό πρόγραμμα και του παραλληλοποιημένου προγράμματος του MPI.
Επιπροσθέτως, στο παραδοτέο αρχείο συμπεριλαμβάνουμε την υλοποίηση του υβριδικού MPI+OpenMP προγράμματος καθώς και αυτή της CUDA.

Τέλος, στο φάκελο reports και common/scripts βρίσκονται τα παρακάτω αρχεία:

- Ένας αυτοματοποιημένος μηχανισμός για να τρέχουμε εκτελέσεις των προγραμμάτων παράλληλα.
- Τα αποτελέσματα αυτών συγκεντρωμένα σε csv αρχεία.
- Ένα πρόγραμμα σε γλώσσα python για τις γραφικές αναπαραστάσεις.

Οδηγίες για την εκτέλεση των προγραμμάτων

Σε όλες τις ενότητες υπάρχουν αρχεία *Makefile* για την εύκολη εκτέλεση των προγραμμάτων. Οι εντολές των *Makefile* παίρνουν ως προκαθορισμένες παραμέτρους για το μέγεθος του προβλήματος και τον αριθμό των διεργασιών το 840 και 4 αντιστοιχα για το πρόγραμμα MPI και 2 για τον αριθμό των νημάτων για το πρόγραμμα hybrid MPI+OpenMP. Για παράδειγμα, εκτελούμε το MPI πρόγραμμα ως εξής:

- Μεταγλώττιση:
  `make compile-release`

- Εκτέλεση:
  `make qsub-release np=16 size=1680`

  όπου `np` ο αριθμός των διεργασιών και `size` το μέγεθος του προβλήματος.

Με την εντολή `make show-latest-qsub-info` βλέπουμε το αποτέλεσμα της πιο πρόσφατης εκτέλεσης μας.

Στην ενότητα με το ακολουθιακό πρόγραμμα υπάρχουν διάφορες παραλλαγές βελτιστοποίησης που μπορούν να τρέξουν ξεχωριστά με την επιλογή της κατάλληλης παραμέτρου. Οπότε για παράδειγμα με την εντολή `make qsub-release -O1` τρέχουμε την βελτιστοποιημένη εκδοχή 1 του ακολουθιακού προγράμματος.

Ακολουθιακό Πρόγραμμα

Ως πρώτο βήμα της εργασίας, δοκιμάσαμε να βελτιστοποιήσουμε τον ακολουθιακό κώδικα. Δοκιμάσαμε διάφορες βελτιστοποιήσεις οι οποίες μπορούν να χωριστούν σε δύο κατηγορίες:

- **Βελτιστοποιήσεις προϋπολογισμών πριν την έναρξη των jacobi iterations.**

  Η ιδέα πίσω από αυτές τις βελτιστοποιήσεις είναι η εξής. Αρχικά βρίσκουμε τις τιμές που υπολογίζονται εντός του jacobi iteration double loop αλλά είναι ανεξάρτητες από τα περιεχόμενα του πίνακα u. Έπειτα, αφαιρούμε τον υπολογισμό από το double loop και τον μεταφέρουμε σε σημείο πριν την έναρξη των jacobi iterations. Οι βελτιστοποιήσεις αυτές αντικαθιστούν ένα σημαντικό μέρος από τους υπολογισμούς του double loop με memory reads σε precalculated τιμές.

  Με μια πρώτη ματιά, οι βελτιστοποιήσεις προϋπολογισμών έδειξαν σημαντική βελτίωση στην απόδοση του προγράμματος.

- **Βελτιστοποιήσεις σε επίπεδο εντολών.**

  Η ιδέα πίσω από αυτές τις βελτιστοποιήσεις είναι να προσπαθήσουμε να βελτιώσουμε τον κώδικα όσο περισσότερο γίνεται σε επίπεδο εντολών.

  Ως παράδειγμα μιας τέτοιας βελτιστοποίησης είναι η αλλαγή της εξίσωσης υπολογισμού του “f” ώστε να περιέχει λιγότερες πράξεις πρόσθεσης και πολλαπλασιασμού. Από (8 πολλαπλασιασμούς και 7 προσθαφαιρέσεις):
  ```
  f = -alpha*(1.0-fX*fX)*(1.0-fY*fY)-2.0*(1.0-fX*fX)-2.0*(1.0-fY*fY)
  ```
  Σε (7 πολλαπλασιασμούς και 3 προσθαφαιρέσεις):
  ```
  f = alpha_plus_2*(fX*fX + fY*fY)-alpha*fX*fX*fY*fY-alpha_plus_4
  ```

  Στην πράξη τις περισσότερες από αυτές τις βελτιστοποιήσεις τις εφαρμόζει ο compiler όταν μεταγλωτίζει με -O2 ή -O3. Και για όσες δεν εφαρμόζει ο compiler, όπως για παράδειγμα την παραπάνω μετατροπή εξίσωσης, δεν προκύπτει μεγάλο όφελος γλιτώνοντας μερικές πράξεις πολλαπλασιασμού, πόσο μάλλον πρόσθεσης. Οπότε δεν προσέφεραν παρά μια πολύ μικρή βελτίωση στην απόδοση του προγράμματος.

Όλες οι βελτιστοποιήσεις που δοκιμάσαμε βρίσκονται στο παραδοτέο μας και μπορούν να ελεγχθούν παραμετροποιώντας το πρόγραμμα με σχετικό command line argument κατά την εκτέλεσή του, για παράδειγμα:

```
make compile-release opt=3x
```

Οι βελτιστοποιήσεις του σειριακού προγράμματος βρίσκονται στο `./sequential/src/jacobi_iteration_*.c`

Θα δείτε πως στο σειριακό κώδικά μας χρησιμοποιούμε MPI και κάποια επικοινωνία μεταξύ των διεργασιών. Έχουμε συμπεριλάβει MPI λογική στο sequential πρόγραμμά μας απλά για να τεστάρουμε τις βελτιστοποιήσεις κάνοντας spawn πολλαπλά identical processes στο ίδιο node (δηλαδή δεν υπάρχει παραλληλοποίηση για την επίλυση του προβλήματος). Για παραπάνω πληροφορίες δείτε το section "Βελτιστοποιήσεις και cache" στην επόμενη σελίδα.

Οι βελτιστοποιήσεις μας αναλυτικά:

1) Προϋπολογισμός των fX και fY:
    - Βελτίωση απόδοσης: `~41%`
    - Memory overhead: `O(n+m)`
    - Make command line flag: `opt=1`
2) Προϋπολογισμός των fX και fY και βελτιστοποιήσεις σε επίπεδο εντολών:
    - Βελτίωση απόδοσης: `~42%`
    - Memory overhead: `O(n+m)`
    - Make command line flag: `opt=1x`
3) Προϋπολογισμός των fX*fX και fY*fY:
    - Βελτίωση απόδοσης: `~40%`
    - Memory overhead: `O(n+m)`
    - Make command line flag: `opt=2`
4) Προϋπολογισμός των fX*fX και fY*fY και βελτιστοποιήσεις σε επίπεδο εντολών:
    - Βελτίωση απόδοσης: `~42%`
    - Memory overhead: `O(n+m)`
    - Make command line flag: `opt=2x`
5) Προϋπολογισμός του f:
    - Βελτίωση απόδοσης: `~42%`
    - Memory overhead: `O(n*m)`
    - Make command line flag: `opt=3`
6) Προϋπολογισμός του f και βελτιστοποιήσεις σε επίπεδο εντολών:
    - Βελτίωση απόδοσης: `~43%`
    - Memory overhead: `O(n*m)`
    - Make command line flag: `opt=3x`

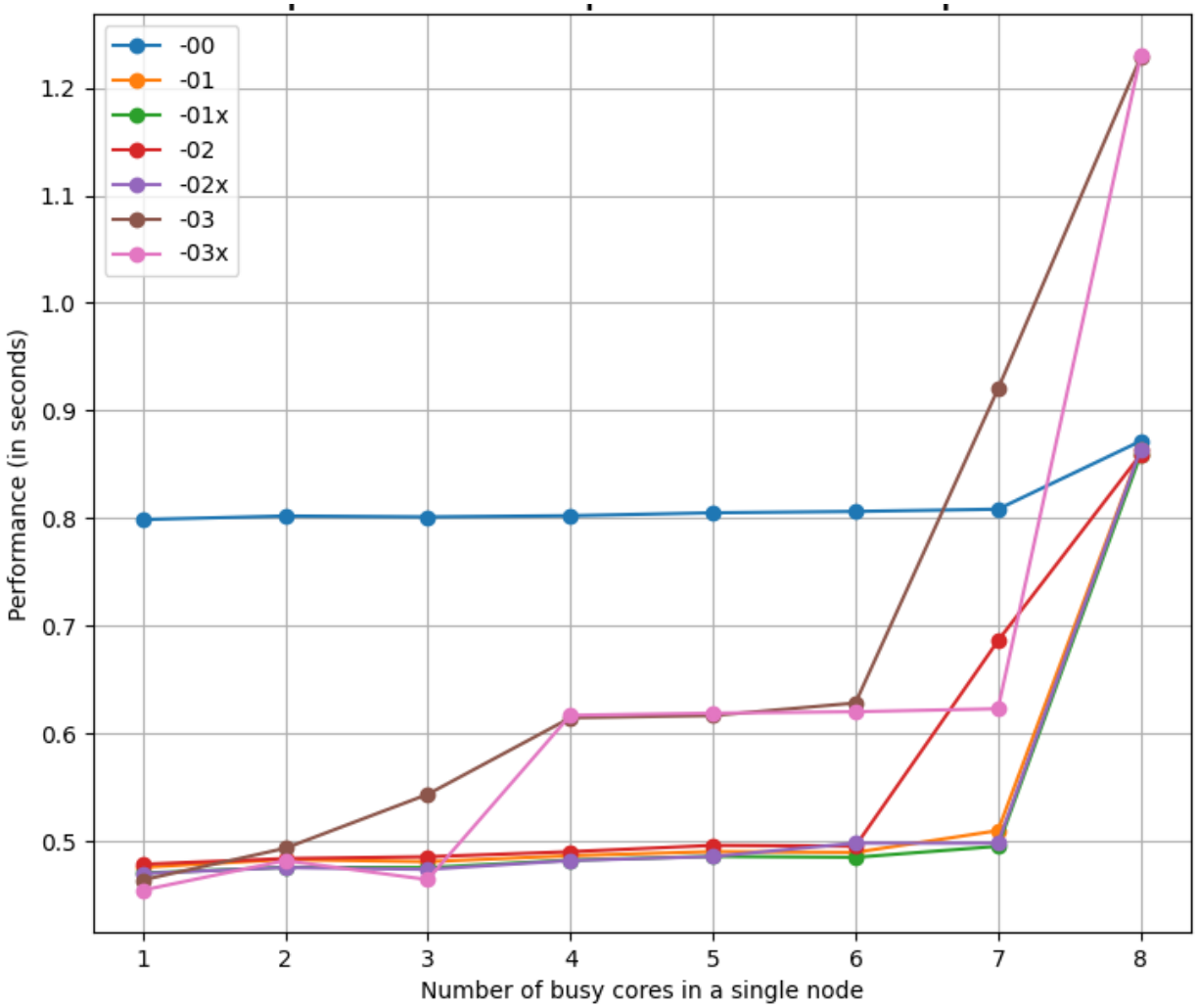
Από τα παραπάνω συμπεραίνουμε πως η σημαντική βελτίωση προέρχεται από τον προϋπολογισμό των fX και fY. Οι υπόλοιπες βελτιστοποίησεις βοηθάνε κάπως, αλλά όχι σημαντικά.

Βελτιστοποιήσεις και cache

Μεταφέροντας την καλύτερη βελτιστοποίηση, την `3x`, στην MPI υλοποίησή μας, διαπιστώσαμε ότι σε διάφορες περιπτώσεις έδινε άσχημα αποτελέσματα. Μετά από μερικές δοκιμές είδαμε πως όταν το πρόγραμμα χρησιμοποιεί σε πολλούς από τους πυρήνες ενός κόμβου, τότε η απόδοση πέφτει.
Καταλήξαμε στο ότι το πρόβλημα κρυβόταν στην υπερβολική χρήση της cache της CPU. Συγκεκριμένα, η βελτιστοποίηση `3x` χρησιμοποιεί σημαντικά περισσότερη μνήμη (i.e. `O(n*m)` αντί `O(n+m)`) στην οποία ανατρέχει συνεχώς για να φορτώσει διαφορετικές precalculated τιμές. Όταν κάθε πυρήνας έχει αναλάβει μέρος της εκτέλεσης και προσπαθεί να φορτώσει διαφορετικά κομμάτια των precalculated τιμών, προκαλούνται πολλά conflicts στην cache, οπότε και προκύπτουν πολύ περισσότερα cache misses.

Τελικώς διαπιστώσαμε πως η φαινομενικά καλύτερη βελτιστοποιημένη εκδοχή του προγράμματος, χειροτέρευε κατά πολύ τους χρόνους εκτέλεσης όταν και οι 8 πυρήνες της cpu ενός node συμμετείχαν στη λύση του προβλήματος. Οπότε επιλέξαμε να συνεχίσουμε με τη βελτιστοποίηση `1` η οποία είναι ένας συνδυασμός σταθερά καλής απόδοσης και ευανάγνωστου κώδικα.

Η συμπεριφορά που περιγράφεται παραπάνω φαίνεται στο παρακάτω διάγραμμα:



MPI

Για την υλοποίηση της δομής του προγράμματος ακολουθήσαμε τον οδηγό που μας δόθηκε στην εκφώνηση της εργασίας.

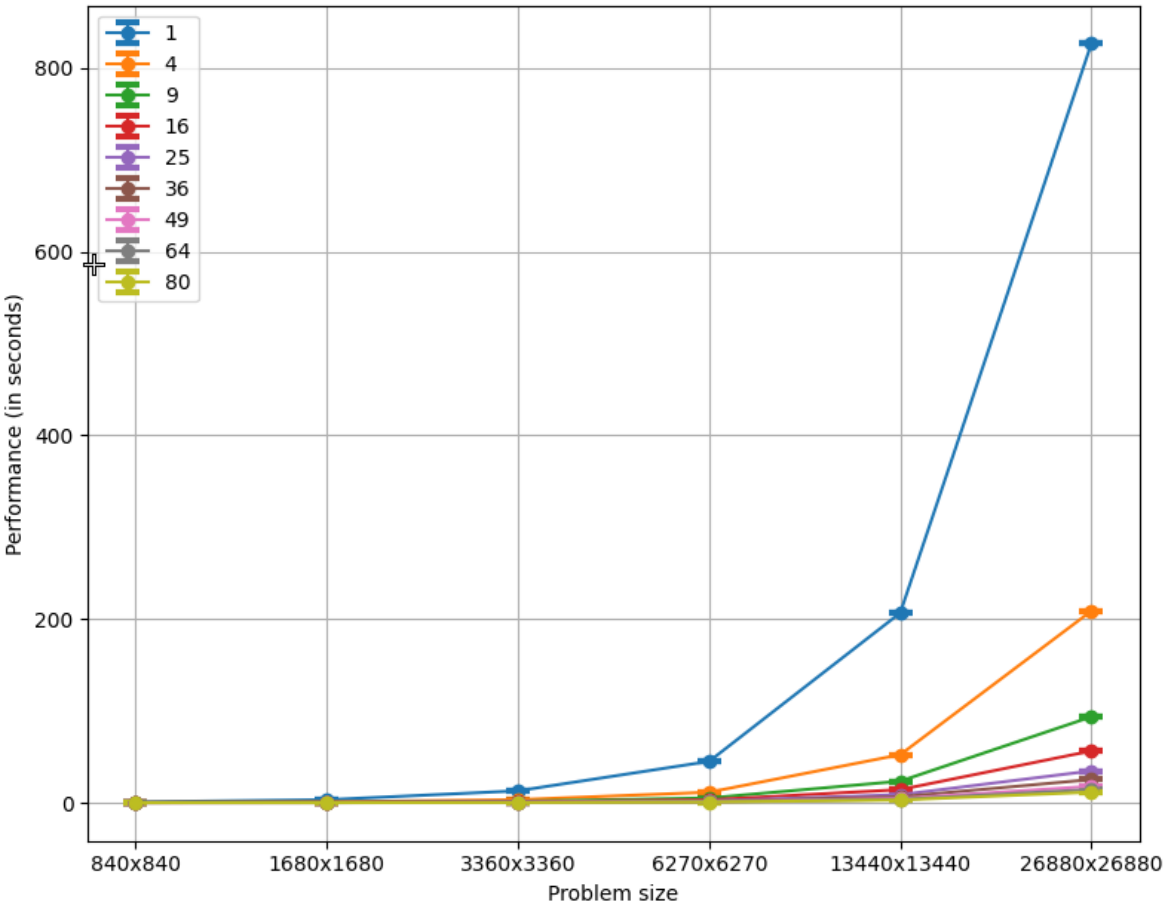
Αρχικά, η μέθοδος `MPI_Dims_create` ορίζει τις διαστάσεις της τοπολογίας μας (εδώ 2) και αναθέτει τον αριθμό των των διεργασιών που θα αναλάβει κάθε μία. Έπειτα, το πρόγραμμα δημιουργεί μία καρτεσιανή τοπολογία με τη μέθοδο `MPI_Cart_Create` με την οποία κάθε διεργασία συνδέεται με τους γείτονες της σε ενα εικονικό grid. Έπειτα γίνεται η μεταφορά των διεργασιών στη τοπολογία με τη μέθοδο `MPI_Cart_coords` και ορίζουμε την κατεύθυνση των hops με την μέθοδο `MPI_Cart_shift`.
Στο σημείο αυτό, ορίζονται 2 δομές δεδομένων MPI, η *row* τύπου MPI contiguous και η *column* τύπου MPI vector που θα χρησιμοποιηθούν για τη λήψη και αποστολή των halo points από και προς τις γειτονικές διεργασίες ως ομάδες στοιχείων και όχι μεμονωμένα. Στη συνέχεια δόθηκε προσοχή στο να γίνει ο κατάλληλος διαχωρισμός των sub-grids από το αρχικό grid μεταξύ των διεργασιών.

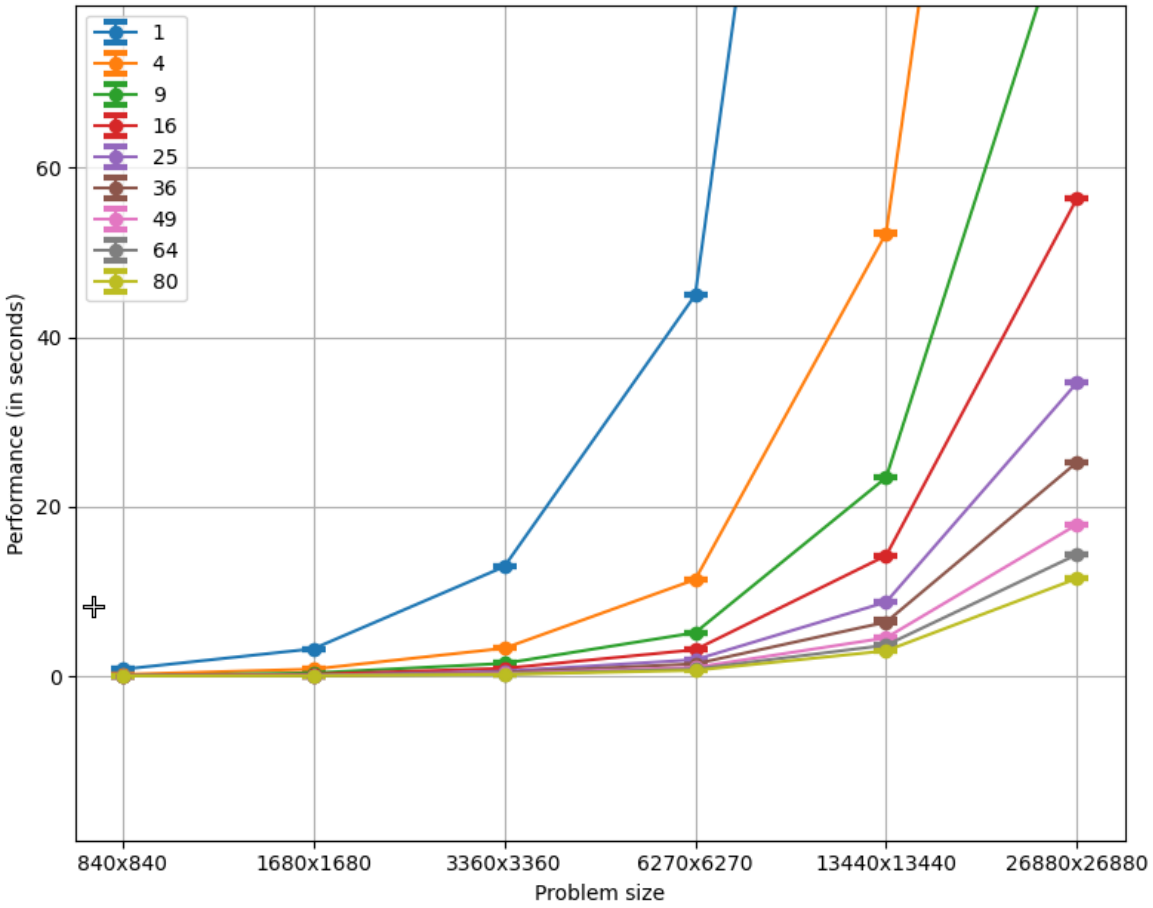
Αφού γίνουν όλα τα παραπάνω, κάθε διεργασία εισέρχεται στο κομμάτι του κώδικα με την κεντρική επανάληψη *while*. Όπως προαναφέρθηκε, η δομή του κώδικα ακολουθεί τη λογική των οδηγιών της εκφώνησης όσον αφορά τη σειρά των πράξεων των οποίων την απόδοση μελετάμε. Τέλος, η μέθοδος *check_solution* έχει παραλληλοποιηθεί και αυτή, ώστε να συγκεντρώνονται τα αποτελέσματα της από κάθε διεργασία στην κεντρική με αριθμό 0.

Βελτιστοποίησεις MPI

1) Όπως αναφέρθηκε και παραπάνω έγινε χρήση των datatypes με σκοπό την αποφυγή αντιγραφής τιμών σε buffers, αλλά και την αποστολή ομάδων τιμών και όχι κάθε μιας μεμονωμένα, μειώνοντας έτσι το πρόβλημα του latency.
2) Οι εντολές MPI receive βρίσκονται πριν τις MPI send ώστε να οι διεργασίες να είναι έτοιμες να δεχτούν μηνύματα.
3) Ο υπολογισμός των ranks των 4 σταθερών γειτόνων γίνεται μια φορά έξω από την κεντρική επανάληψη.
4) Τοποθέτηση διεργασιών που επικοινωνούν στον ίδιο κόμβο με τη χρήση του rank reorder για την μείωση κόστους επικοινωνίας.
5) Ο υπολογισμός των εσωτερικών τιμών της διεργασίας υπολογίζεται χωρίς να μπλοκάρεται από την αποστολή και λήψη των halo points.
6) Μετέπειτα του υπολογισμού των εσωτερικών στοιχείων κάθε διεργασίας δεν χρησιμοποιήθηκε κάποια από τις μεθόδους MPI Wait/Waitall, αλλά η `MPI_Waitany`, ώστε το πρόγραμμα να ξεκινάει τους υπολογισμούς για την συγκεκριμένη διεργασία που έλαβε χωρίς να περιμένει τις υπόλοιπες αναγκαστικά.

Διάγραμμα MPI (ολόκληρο και zoomed)





Στο διάγραμμα απεικονίζεται η απόδοση του MPI προγράμματος. Στους άξονες βρίσκονται η απόδοση σε δευτερόλεπτα και το μέγεθος του προβλήματος, ενώ ο πίνακας πάνω και δεξιά ορίζει τον αριθμό των διεργασιών με το αντίστοιχο χρώμα. Το αποτέλεσμα όπως φαίνεται είναι πολύ θετικό! Η αύξηση του αριθμού των διεργασιών οδηγεί σε όλο και μεγαλύτερη μείωση του χρόνου εκτέλεσης του προγράμματος. Συγκεκριμένα, κάθε αύξηση του αριθμού των διεργασιών φανερώνει και ενα υποτετραπλασιασμό του χρόνου εκτέλεσης σε σχέση με τον αμέσως προηγούμενο αριθμό.

Hybrid MPI & OpenMP

Στην υλοποίηση αυτή προστέθηκε κώδικας OpenMP πάνω στο MPI πρόγραμμα. Αρχικά, ακολουθώντας τις υποδείξεις της εκφώνησης, δοκιμάστηκαν διάφοροι συνδυασμοί διεργασιών-νημάτων πάνω σε ένα κόμβο με στόχο την εύρεση του πιο αποδοτικού, καταλήγοντας στον 4δ-2ν, δηλαδή 4 διεργασίες και 2 νήματα. Έπειτα, πραγματοποιήθηκαν περισσότερες δοκιμές με σταθερό αριθμό διεργασιών (4) και όλα τα πιθανά μεγέθη του προβλήματος και αριθμό νημάτων (2,4,8), οι οποίες επιβεβαίωσαν το αρχικό συμπέρασμα μειώνοντας το χρόνο εκτέλεσης στο μισό συγκριτικά με το αντίστοιχο MPI πρόγραμμα. Η παραλληλοποίηση έγινε σε 3 σημεία:

- Στο διπλό for loop με τον υπολογισμό των εσωτερικών στοιχείων.
- Στα δύο for loop για τον υπολογισμό των εξωτερικών στοιχείων.

Παρόλα αυτά, λόγω των προαναφερθέντων τεχνικών προβλημάτων δεν μπορέσαμε να κάνουμε μετρήσεις σε όλο τους το εύρος που ζητείται.

CUDA

Single GPU

Η CUDA υλοποίηση έγινε μόνο με τη δυνατότητα compilation αλλά όχι εκτέλεσης και τεσταρίσματος (καθώς η argo δεν εκτελεί επιτυχώς CUDA προγράμματα), οπότε ενδέχεται το πρόγραμμα να περιλαμβάνει διάφορα λογικά bugs.

Όπως θα δείτε στο αρχείο `./cuda/src/jacobi_gpu.cu`, υπάρχουν 3 TODOs βελτιστοποιησης και 1 TODO που έχει να κάνει με βέβαιη λανθασμένη λογική του κώδικά μας γύρω από το sum-reduction του residual error μεταξύ των blocks.

Παρεμπιπτόντως, το πρώτο iteration του sum-reduce γίνεται μέσα στον `jacobi_iteration_gpu` kernel καθώς έχουμε ήδη έτοιμα τα δεδομένα στη thread-shared memory του κάθε block.

2 GPUs

Δεν έχουμε υλοποιήσει λύση του προβλήματος χρησιμοποιώντας 2 GPUs (πάλι εδώ δε μας διευκόλυνε η αδυναμία δοκιμαστικών εκτελέσεων ορθής συμπεριφοράς του προγράμματος). Η λογική μας πάντως θα ήταν η εξής:

1. Θα αναθέσουμε το πάνω μισό του grid μας στη μία GPU και το δεύτερο μισό στην άλλη.
2. Έστω `n*m` το μέγεθος του grid δεδομένων μας (μαζί με τα τριγύρω μηδενικά κελιά). Δεσμεύουμε δύο device/gpu buffers μεγέθους `n*(m/2+1)`, ένα για κάθε GPU. Το `+1` αφορά τα ghost rows του κάθε buffer.
3. Αντιγράφουμε τις γραμμές `[0,m/2+1)` του grid στο πρώτο buffer και τις γραμμές `[m/2-1,m)` του grid στο δεύτερο buffer.
4. Για κάθε GPU κάνουμε instantiate τον ίδιο kernel `jacobi_iteration_gpu` περνώντας στην καθεμία διαφορετικό buffer.
5. Αφού υπολογιστούν οι νέες τιμές του πίνακα, κάνουμε copy τις shared γραμμές του grid στα ghost rows των buffers, με `cudaMemcpy` και last argument `cudaMemcpyHostToDevice`.
6. Αφού υπολογιστούν τα sum-reduced residual errors απο την κάθε GPU, ο CPU κώδικας της `jacobi_gpu` απλά εκτελεί:

```
sqrt(error_from_gpu1 + error_from_gpu2) / (n*m)
```