HPC – 2D Heat Diffusion

Εκπόνηση πρώτης εργαστηριακής άσκησης:

Η Εργασία εκπονήθηκε από τους:

- Σφήκας Θεόδωρος, 1072550
- Τσούλος Βασίλειος, 1072605

❖ Εισαγωγή:

System Specifications:

Το πρόγραμμα για την παράδοση των τελικών αποτελεσμάτων εκτελέστηκε σε επεξεργαστή *Ryzen 5600x* και σε σύστημα με *16GB RAM 3200Mhz*. Ακολουθούν περαιτέρω διευκρινίσεις πάνω στα χαρακτηριστικά του επεξεργαστή:

» AMD Ryzen 5 Desktop Processor:

- # of CPU Cores → 6
- # of Threads → 12
- Max. Boost Clock→ Up to 4.6GHz *(ενεργό)*
- Base Clock→ 3.7GHz
- L1 Cache → 64 KB *(per core)*
- L2 Cache → 512 KB *(per core)*
- L3 Cache→ 32 MB *(shared)*

Οδηγίες εκτέλεσης:

Στον φάκελο που παραδώθηκε βρίσκονται αρχικά δύο .c προγράμματα, το πρώτο αποτελεί την δοσμένη υλοποίηση για το diffusion2d με non blocking μεταφορά δεδομένων, ενώ το δεύτερο αφορά την δική μας υλοποίηση. Ακόμα βρίσκονται το κατάλληλα τροποποιημένα Makefile και δύο shell scripts για την εκτέλεση των ζητούμενων, είτε για μία εκτέλεση, είτε για όλα τα ζητούμενα μαζί αντιστοίχως. Τέλος υπάρχει ένα τυπικό python script για την παραγωγή των γραφικών παραστάσεων με βάση τα αποτελέσματα της εκτέλεσης.

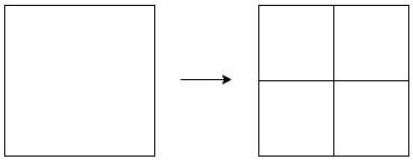
Παραδοχές:

Η υλοποίηση μας βασίστηκε στην προσπάθεια εκμετάλλευσης όσων περισσότερων επιπέδων παραλληλοποίησης θεωρούσαμε ουσιώδεις. Ως εκ-τούτου με την θεώρηση πώς προγραμματίζουμε για έναν multi-node server υλοποιήσαμε hybrid παραλληλισμό με την χρήση mpi+openmp καθώς και έγινε χρήση instrinsics με στόχο το simd parallelization.

❖ Συνοπτικοί σχολιασμοί υλοποίησης:

Η Υλοποίηση αφορά αρχικά την μετατροπή του διαχωρισμού του workload του αρχικού heat-map (two dimensional array) από sub-arrays στηλών σε sub-arrays blocks (τετραγώνων) και την αναφορά καθένα από αυτά σε κάθε ένα process. Ακόμα θεωρούμε πως οι διαστάσεις του αρχικού μητρώου είναι προκαθορισμένες και διαιρούμενες ακέραια σε blocks των 4 ή των 16. Κρίνουμε πως ο σχολιασμός του κώδικα είναι υπερβολικά εκτενής για την εξήγηση της διαδικασίας και για τον λόγο αυτό θα σχολιάσουμε ιδιαίτερα σχεδιαστικές επιλογές και προβληματισμούς που προέκυψαν κατά την υλοποίηση.

1. Αρχικά εφόσον η εκτέλεση προκαθορισμένα γίνεται είτε με 4 ή 16 processes δεν γίνεται έλεγχος για δυναμικό διαχωρισμό του αρχικού μας μητρώου, ούτε γίνεται έλεγχος μήπως ο διαχωρισμός που θα γίνει δεν διαιρεί τέλεια το N_ (matrix dimension). Με στόχο ωστόσο να βρεθούν τα υπομητρώα και η θέσης (global index) κάθε κελιού πρέπει να βρεθεί ο αριθμός των divisions της αρχικής διάστασης. Αυτός δίνεται με την χρήση της τετραγωνικής ρίζας του αριθμού των processes με τα οποία γίνεται η εκτέλεση. Για παράδειγμα για εκτέλεση με 4 διεργασίες θα υπάρχουν 2 blocks σε κάθε dimension, ενώ με 16 διεργασίες 4 blocks. Ύστερα οι συντεταγμένες κάθε κουτιού = κάθε διεργασίας συγκριτικά με τις υπόλοιπες μπορεί να βρεθεί ως : (x,y) = (rank % divisions, rank / divisions) και κατά συνέπεια εύκολα βρίσκουμε το global index κάθε cell κατά την αρχικοποίηση.



2. Σχολιάζοντας τον κώδικα με βάση την σειρά της εκτέλεσης του, ξεκινάμε από την main. Αρχικά κάνουμε establish το mpi communication (MPI_Init_thread ή MPI_Init ανάλογα. Χρησιμοποιούμε το thread funneled καθώς τα threads δεν υλοποιούν mpi calls) ώστε να επιτρέψουμε την επικοινωνία μεταξύ διεργασιών. Αρχικοποιούμε τις παραμέτρους εκτέλεσης βάση των call arguments και δημιουργούμε εξωτερικά μία και μόνο parallel περιοχή (hybrid κώδικα). Η δημιουργία των threads γίνεται αποκλειστικά μόνο μία φορά για να αποφευχθεί το overhead δημιουργίας τους σε κάθε εκτέλεση της advance. Έπειτα χρησιμοποιείται το master directive ώστε η εκτέλεση να συνεχιστεί αποκλειστικά από ένα και μόνο thread. Η χρήση των υπόλοιπων αποτελεί τον παραλληλισμό τόσο στην διαδικασία της αρχικοποίησης όσο και στο update των two dimensional arrays.

3. Ακολουθεί το κάλεσμα της init() όπου δεσμεύεται ο απαραίτητος χώρος (δυναμικά) του two dimensional array και αρχικοποιούνται οι μεταβλητές εκτέλεσης του συστήματος. Με στόχο την χρήση simd operations και τού αποδοτικού loading στο cache hierarchy των rho_ πινάκων η προαναφερθούσα δέσμευση μνήμης γίνεται aligned με την χρήση της posix_memalign (Only for Darwin - Linux systems) ως πολλαπλάσιο των 64 bytes σύμφωνα με τα specifications του συστήματος μας. Έπειτα γίνεται η

αρχικοποίηση του rho_ πίνακα με την χρήση του global index που αναφέραμε. Είναι σημαντικό να αναφέρουμε πως δεν χρειάζεται να αρχικοποιήσουμε με 0 κελιά εφόσον αυτό γίνεται αυτόματα λόγω της posix_memalign(). Τέλος η αρχικοποίηση γίνεται παράλληλα σε επίπεδο node με την χρήση tasks (hybrid κώδικα). Ακόμα και αν για μικρές διαστάσεις blocks όπως 1024χ1024 cells η διαφορά δεν θα είναι αισθητή, για μεγάλα δεδομένα θα μπορούσε να προσφέρει κάποιο speedup λόγω της NUMA αρχιτεκτονικής. Ωστόσο σε μεγάλες διαστάσεις αντιμετωπίζουμε διαφορετικά προβλήματα που θα αναλύσουμε στην συνέχεια. In hind sight η παραλληλοποίηση του initialization δεν θα μας προσφέρει κάποια πρακτική βελτίωση .

4. Ακολουθεί η εκτέλεση της advance, στην διαδικασία της advance (update των τιμών του μητρώου) η μεταφορά των στοιχείων γίνεται ασύγχρονα και ίσως θα μπορούσε να γίνει γρηγορότερα με την χρήση της mpi_put και των mpi_windows ωστόσο καθώς ο χρόνος ήταν περιορισμένος δεν έγινε εκτενής μελέτη των συγκεκριμένων λειτουργιών. Στην συνέχεια έγινε προσπάθεια εκτέλεσης του update υπομητρώου τόσο με την χρήση simd operations όσο και την hybrid παραλληλοποίηση με χρήση tasks. Simd instrinsics πραγματοποιήθηκαν τόσο αυτόματα με την χρήση του Openmp directive call όσο και με ρητό τρόπο στον κώδικα μας. Στην δεύτερη περίπτωση μάλιστα παρουσιάστηκαν ορισμένες αξιοσημείωτες δυσκολίες, οι οποίες αναφαίρονται εκτενώς στο documentation του κώδικα. Μετά την μεταφορά της πληροφορίας των ghost cells μεταξύ των διεργασιών γίνεται και ο υπολογισμός των καινούργιων τιμών των border cells που δεν έγιναν προηγουμένως.

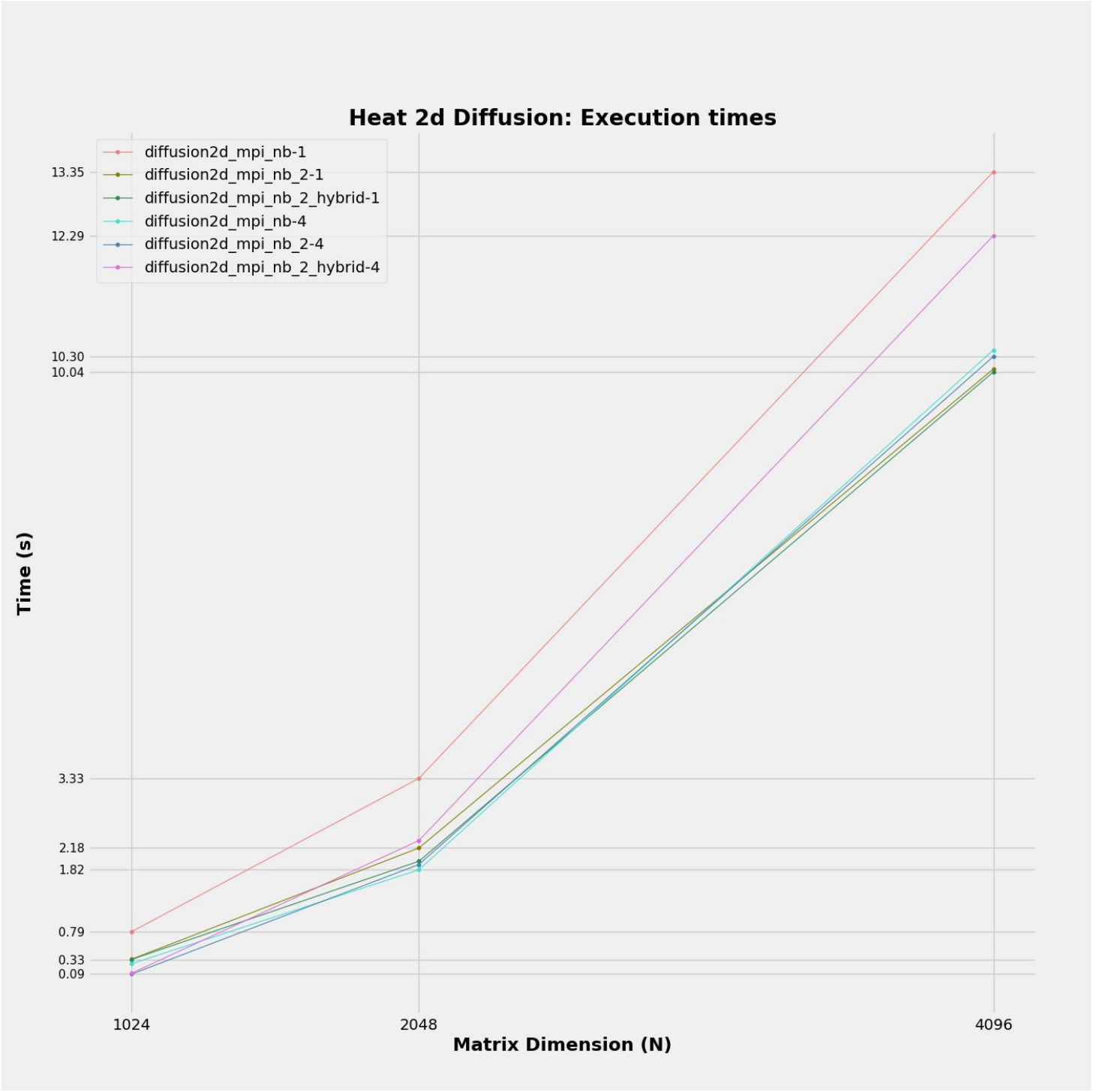
5. Με στόχο την εκτέλεση του δευτέρου ερωτήματος αρχικά συναντάμε δύο καινούργια execution call arguments, τον αριθμό των advances (T’) μέχρι ο κώδικας μας να δημιουργήσει ένα checkpoint και το Resume_signal το οποίο όταν ενεργοποιείται αντί να αρχικοποιηθεί το μητρώο με την διαδικασία των προηγούμενων bullet points (1,3), οι διεργασίες διαβάζουν παράλληλα το binary file και αρχικοποιούν το rho_ array καταλλήλως. Η διαδικασία του checkpointing είναι λίγο περισσότερο περίπλοκη. Για αρχή, η εγγραφή των τιμών του rho_ κάθε διεργασίας στο binary file δεν απαιτεί κάποιο συγχρονισμό γιαυτό και δεν γίνεται η χρήση collective operations οι οποίες θα εισήγαγαν ένα ακόμα μεγαλύτερο overhead. Έπειτα πρέπει να αποφασιστεί αν το checkpointing θα πραγματοποιηθεί ασύγχρονα με την ροή εκτέλεσης κάθε διεργασίας είτε χρησιμοποιώντας κατάλληλο mpi call είτε με την χρήση νημάτων. Αφού πειραματιστήκαμε με τους διάφορους μηχανισμούς παρατηρούμε πως για τα ζητούμενα μεγέθη ο χρόνος εκτέλεσης της εγγραφής στο binary file είναι 2 τάξεις μικρότερος από τον χρόνο μίας advance. Συνυπολογίζοντας το απαιτούμενο κόστος αντιγραφής των δεδομένων του rho_ σε ένα temporary buffer με σκοπό την αποφυγή race conditions, και το sparsity των checkpoint καλεσμάτων δεν κρίναμε απαραίτητη την ασύγχρονη εκτέλεση του.

❖ Παρουσίαση αποτελεσμάτων:

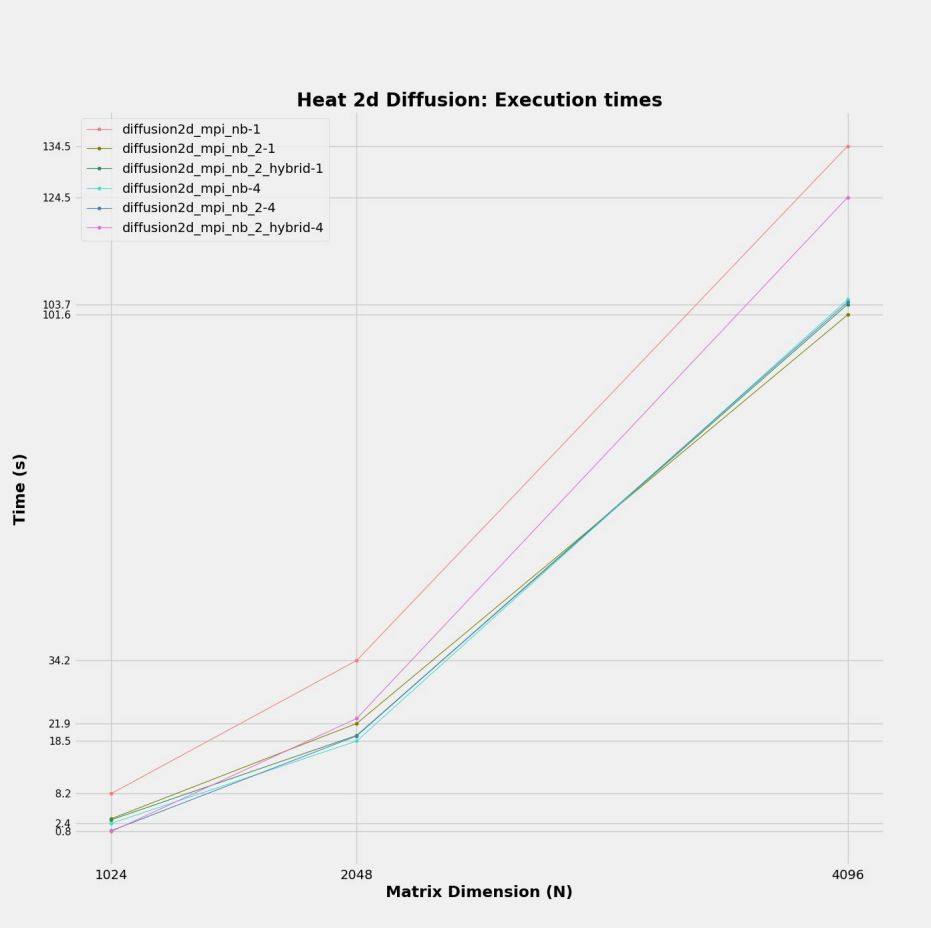
Diffusion2d_mpi_nb:			
P.	**N**	**T**	**Time (s)**
1	1024	1000	0.793049
1	2048	1000	3.325828
1	4096	1000	13.349927
4	1024	1000	0.262256
4	2048	1000	1.815889
4	4096	1000	10.404311

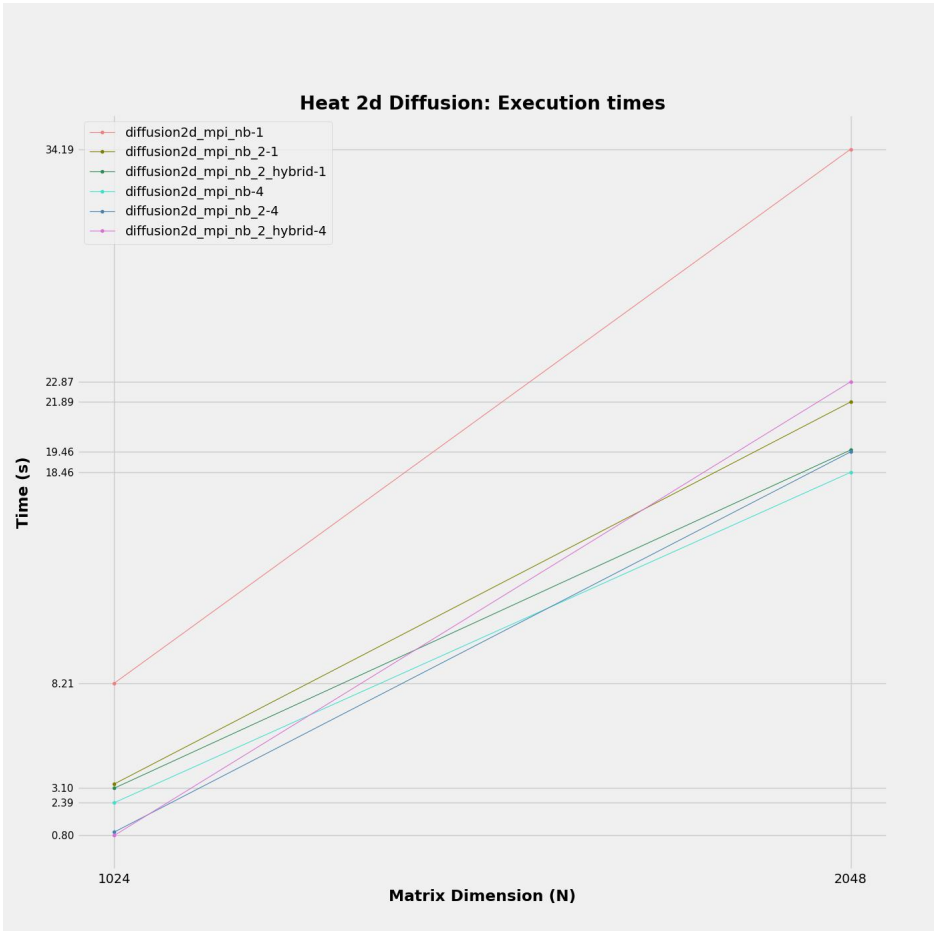
Diffusion2d_mpi_nb_2:			
P.	**N**	**T**	**Time (s)**
1	1024	1000	0.338128
1	2048	1000	2.175518
1	4096	1000	10.090126
4	1024	1000	0.091192
4	2048	1000	1.902814
4	4096	1000	10.296432

Diffusion2d_mpi_nb_2_hybrid:			
P.	**N**	**T**	**Time (s)**
1	1024	1000	0.334066
1	2048	1000	1.955929
1	4096	1000	10.040661
4	1024	1000	0.10722
4	2048	1000	2.298889
4	4096	1000	12.293669



- Καθώς οι χρόνοι εκτέλεσης ειδικά για *N=1024* είναι υπερβολικά μικροί για σωστότερο σχολιασμό ξανά-εκτελέσαμε το ζητούμενο για *T=10000* και παραθέτουμε τα παραγόμενα γραφήματα:





❖ Συμπεράσματα και Παρατηρήσεις:

1. Αρχικά όπως διακρίνουμε οι δύο εκτελέσεις που παρατίθενται είναι πανομοιότυπες όπως και θα περιμέναμε. Καθώς το υπολογιστικό σύστημα εκτέλεσης έχει 6 πυρήνες και 12 threads η εκτέλεση του hybrid μοντέλου με 4 processes χρησιμοποιεί 3 threads σε κάθε ένα από αυτά. Όπως φαίνεται ωστόσο αυτό δεν οδηγάει και σε καλύτερα αποτελέσματα σε κάθε μία εκ των διαστάσεων. Θα μιλήσουμε πιο συγκεκριμένα για τα αποτελέσματα γενικότερα και τον συνδιασμό τους για την εκπόρευση αποτελεσμάτων στην συνέχεια.
2. Αρχικά τα αποτελέσματα μας για *N=1024* παρουσιάζουν ένα μοτίβο μεταξύ των αποτελεσμάτων για 1 και για 4 processes. Οι εκτελέσεις μεταξύ των δύο "δικών μας" προγραμμάτων και στις δύο περιπτώσεις είναι περίπου **3 φορές γρηγορότερες** από τον αρχικό κώδικα. Φαίνεται πώς δεν υπάρχει bottleneck στην διαδικασία μεταφοράς των δεδομένων (ghost cells) μεταξύ των διεργασιών αφού συσχέτιση των χρόνων παραμένει σταθερή για τις εκτελέσεις μεταξύ της 1 και των 4 διεργασίων. Επίσης η χρήση των threads (hybrid εκτελέσιμο) παράγει **ελάχιστα** γρηγορότερη εκτέλεση των αποτελεσμάτων. Φαινόμενο που θα μας προβλημάτιζε αν δεν συνυπολογίζαμε το χρονικό κόστος κάθε μίας advance. Η τάξη μεγέθους του κόστους εκτέλεσης της advance είναι πολύ μικρή για να δημιουργηθεί η ανάγκη για hybrid παραλληλισμό. Η μέση χρονική διάρκεια υπολογισμού της advance είναι *0.000363s* και *0.000279s* για την απλή και την hybrid εκτέλεση αντίστοιχα. Αξίζει να σημειωθεί πώς για *N=1024* το υπομητρώο κάθε διεργασίας έχει μέγεθος *local_N=256.* Η ουσιώδης βελτίωση της απόδοσης λοιπόν οφείλεται στην χρήση των intrinsics (simd). Ελέγχοντας το **Speedup** μπορούμε να παρατηρήσουμε πως το scaling της εφαρμογής μας τουλάχιστον για το συγκεκριμένο N είναι καλύτερο της αρχικής: **3.87 έναντι 3.43.** Αυτό ωστόσο υποθέτουμε πως ευθύνεται περισσότερο στην εκμετάλλευση του simd παρά στο blocking που υλοποιήσαμε. Τέλος πραγματικά ιδιαίτερη σημασία έχει η εκτέλεση των προγραμμάτων χωρίς το performance mode. Με την χρήση του run_once.sh μπορούμε να κάνουμε ουσιαστικά testing, τυπώνοντας τον χρόνο εκτέλεσης κάθε update του κεντρικού submatrix και τον χρόνο που χάνουμε περιμένοντας την μεταφορά των ghost cells. Φαίνεται καθαρά το μέσο busy waiting των προγραμμάτων για την μεταφορά των δεδομένων, το οποίο και στα τρία προγράμματα είναι παρόμοιο και της τάξης του *0.000065s.*
Μεταβαίνοντας σε μεγαλύτερες διαστάσεις (*N = 2048*) παρατηρούμε πως τα αποτελέσματα μας περιπλέκονται. Αρχικά για να κάνουμε ερμήνευση των αποτελεσμάτων πρέπει να ξαναδούμε το κατά πόσο η εφαρμογή μας είναι I/O bounded. Παρατηρούμε πώς στο hybrid κώδικα ο μέσος χρόνος αναμονής των ghost cells είναι τουλάχιστον 5 φορές μεγαλύτερος από τα άλλα δύο προγράμματα. Το φαινόμενο αυτό είναι κλασσική συνέπεια του **oversubscription.** Χρησιμοποιούμε και δεσμεύουμε περισσότερα threads από ότι υποστηρίζει το υπολογιστικό μας σύστημα γιατί δεν λαμβάνουμε υπόψιν μας τα background threads με τα οποία λειτουργεί το MPI. Ως εκτούτου η αποστολή των δεδομένων γίνεται σημαντικά πιο αργή, και κατά συνέπεια το ίδιο το πρόγραμμα. Ωστόσο τίποτα δεν μπορεί για εμάς να εξηγήσει λογικά την

αύξηση της ταχύτητας υπολογισμού του κεντρικού πυρήνα (async update of submatrix cell) από το αρχικό πρόγραμμα (Μέσος χρόνος υπολογισμού για : *mpi_nb = 0.007097s και mpi_nb_2 = 0.007503s*). Υποθέτουμε (χωρίς κάποια βάση) πως ίσως ακόμα και χωρίς την συνοδεία των απαραίτητων flags πέρα από το *O2* δηλαδή, ο gcc κάνει αυτόματα vectorization όταν κρίνει πως το μέγεθος του workload είναι αρκετά μεγάλο. Στις πιο γρήγορες εκτελέσεις βρίσκουμε επίσης και την εκτέλεση του hybrid κώδικα με 3 threads και 1 process. Η εκτέλεση αυτή φυσικά θα παρομοίαζε έναν κώδικα παραλληλοποιημένο αποκλειστικά με Openmp tasks. Παρατηρούμε πως με 3 threads και όχι 4 ταυτίζεται σχετικά σε χρόνο με τις εκτελέσεις των προαναφερθέντων προγραμμάτων με 4 processes, καθώς δεν έχει καθόλου waiting time για την μεταφορά των ghost cells. Τέλος αξίζει να σημειωθεί πως το **speedup** του *mpi_nb_2* κώδικα μειώνεται σημαντικά σε ***1.12*** *έναντι του* ***3.87*** που συναντήσαμε για *N= 1024.* Το φαινόμενο αυτό θεωρούμε πως οφείλεται στην σημαντική αύξηση του waiting time για τα ghost cell transfers μεταξύ των διεργασιών (*0.000195s σχεδόν* ***τριπλάσια μέση καθυστέρηση*** *από ότι είχαμε διαπιστώσει για N=1024.*). Για *N= 4096,* γίνονται παρόμοιες παρατηρήσεις.

- Τι παραπάνω μπορούμε να κάνουμε ?
  Σαφώς ο βασικός στόχος μας αποτελεί η μείωση του μη εκμεταλλεύσιμου χρόνου στον ασύγχρονο υπολογισμό του update " computational kernel ". Αυτό μπορεί να επιτευχθεί με δύο τρόπους είτε μειώνοντας τους χρόνους μεταφορών είτε αυξάνοντας τις πράξεις του υπολογιστικού μας πυρήνα. Αρχικά πέρα από τους πιθανούς πειραματισμούς με τα mpi_windows θα μπορούσαμε να ελέγχουμε για την πραγματοποίηση καθένα από τα status μεταφορών ξεχωριστά το οποίο με την χρήση πιθανώς ενός pointer function θα οδηγούσε στο update της συγκεκριμένης περιοχής των border cells. Ωστόσο αυτή η μέθοδος κρίνουμε πως πρώτον, θα εισήγαγε επιπλέον overhead στους υπολογισμούς και κατά δεύτερον η τάξη μεγέθους του update συγκεκριμένων border cells είναι τόσο μικρή που δεν πιστεύουμε πως θα έλυνε το πρόβλημα. Ένας ακόμα πειραματισμός ωστόσο θα μπορούσε να είναι ο διπλασιασμός του ασύγχρονου workload της advance. Ύστερα από τον υπολογισμό του αρχικού submatrix θα μπορούσαμε να επαναλάβουμε την διαδικασία και να ξανά υπολογίσουμε με την χρήση των καινούργιων τιμών ένα ακόμα μικρότερο κατά 1 border cell submatrix. Θα μειώναμε επίσης ουσιαστικά και τον αριθμό των απαιτούμενων advances σχεδόν στο μισό. Όσο και αν η εκτέλεση μίας τέτοιας μεθοδολογίας/ πειραματισμού κρίνεται δύσκολη και ενδιαφέρουσα φυσικά αυξάνουμε την χωρική πολυπλοκότητα του προβλήματος και δεν υπήρχε ο χρόνος να ερευνηθεί σε βάθος.