ΑΝΑΦΟΡΑ ΑΣΚΗΣΗΣ 2

HPY418

Μανώλης Πετράκος

Δημήτριος Καραμπάσογλου

Περιγραφή κώδικα

Έχει γίνει μια αλλαγή στον σειριακό κώδικα, στον υπολογισμό των Fvec[i] η τιμή 0.01 μετατράπηκε σε 0.01f. Ο compiler αναγνωρίζει την πρώτη ως double, και μετατρέπει τις υπόλοιπες στον ίδιο τύπο για να γίνουν οι πράξεις. Τέλος το αποτέλεσμα γίνεται cast πάλι σε float για να αποθηκευτεί η τιμή. Πέρα του ότι η διαδικασία αυτή είναι αρκετά πιο αργή από το να γίνουν οι πράξεις με float, αλλοιώνονται τα αποτελέσματα λόγο των μετατροπών των τύπων. Συγκεκριμένα σε εμάς, ο πρώτος τρόπος είχε χρόνο 3.5ms και ο δεύτερος 3.1ms. Ως αποτέλεσμα μειώνονται τα σχετικά speedup των άλλων τρόπων, όμως μπορούμε να ελέγξουμε την ορθότητα των αποτελεσμάτων.

SSE

Το SSE προσφέρει παραλληλία στις πράξεις, καθώς οι συναρτήσεις του επεξεργάζονται πολλά δεδομένα ταυτόχρονα.

Στην υλοποίηση με χρήση SSE χρειάζεται να υπάρχουν μερικές επιπλέον μεταβλητές και pointers. Συγκεκριμένα τρεις vectors με τις τιμές 1, 2, 0.01f για να μπορούν να γίνουν οι πράξεις, δύο ακόμα vectors που χρησιμοποιούνται για τις συγκρίσεις και τέλος μια σειρά από pointers, με τους οποίους αποφεύγεται η χρήση load και store, πάνω στους πίνακες των δεδομένων.

Η for() που γίνονται οι υπολογισμοί κάνει το 1/4 των επαναλήψεων σε σχέση με την σειριακή υλοποίηση, επειδή κάθε πράξη με vectors επεξεργάζεται 4 float μεταβλητές ταυτόχρονα. Οι πράξεις είναι ίδιες και γίνονται με την ίδια σειρά όπως και στην αρχική υλοποίηση, διότι για τις πράξεις με αριθμούς κινητής υποδιαστολής δεν ισχύει η προσεταιριστική ιδιότητα. Έπειτα τα 4 αποτελέσματα συγκρίνονται οριζοντίως με το vector που κρατάει τα μέγιστα, και αν αυτά είναι μεγαλύτερα τα αντικαθιστούν. Αυτά τα μέγιστα δεν είναι απαραίτητα τα μεγαλύτερα όλων, αλλά μόνο αυτών που θα συγκριθούν. Δηλαδή στη θέση 0 βρίσκεται το μέγιστο των στοιχείων που είναι στις θέσεις i*4, στη θέση 1 βρίσκεται το μέγιστο των στοιχείων που είναι στις θέσεις (i*4)+1, και ούτω καθεξής.

Αφού τελειώσει η for() πρέπει να βρεθεί το ολικό μέγιστο, και γίνεται συγκρίνοντας καθέτως τις τιμές του vector των μεγίστων. Η συγκεκριμένη διαδικασία είναι αρκετά ακριβή σε χρόνο. Για να μικρύνει αυτός ο χρόνος, μειώνονται οι προσβάσεις στη μνήμη αλλά οι πράξεις έχουν γίνει πιο σύνθετες.

Με τα N που ζητάει η εκφώνηση, δεν δημιουργούνται edge cases γιατί διαιρούνται τέλεια με το 4. Για να είναι πιο ολοκληρωμένος ο κώδικας και να δουλεύει σωστά με όλες τις πιθανές ορθές εισόδους, προστέθηκε ένα κομμάτι κώδικα που τα υπολογίζει. Συνεχίζει από το επόμενο του στοιχείου που σταμάτησε η προηγούμενη for(), υπολογίζει και συγκρίνει σειριακά όλα τα υπόλοιπα, τα οποία είναι το πολύ τρία. Προτιμήθηκε αυτή η λύση αντί του padding γιατί οι πίνακες είναι πολλοί και ο κώδικας θα γινόταν αρκετά πιο σύνθετος, χωρίς κάποιο ιδιαίτερο πλεονέκτημα.

SSE Ενδιάμεσες υλοποιήσεις

Για να υλοποιηθεί ο κώδικας με την χρήση SSE χρειάστηκαν 4 στάδια· το unroll, το jam, η μετατροπή σε SSE και η πρόσθεση των ακραίων περιπτώσεων.

Ο κώδικας του unroll συνοπτικά.

```
for(int i=0; i < N; i += 4){
    num_0[0] = LVec[i]+RVec[i];
    ...
    ...

    num_0[1] = LVec[i+1]+RVec[i+1];
    ...
    ...

    num_0[2] = LVec[i+2]+RVec[i+2];
    ...
    ...

    num_0[3] = LVec[i+3]+RVec[i+3];
    ...
    ...
}
```

Αρχικά οι προσωρινές μεταβλητές μετατρέπονται σε πίνακες με 4 floats. Όπου τελείες συνεχίζει ακριβώς όπως στον σειριακό κώδικα αλλά με τους καινούριους δείκτες. Με το unroll επειδή αυξάνεται το i κατά 4 κάθε φορά μειώνονται οι πράξεις του for() στο 1/4 και βελτιώνεται η ταχύτητα. Ακόμα εκμεταλλεύεται καλύτερα τους καταχωρητές του επεξεργαστή, όμως αν γίνει unroll πολλές φορές υπάρχει η περίπτωση να χρειάζεται περισσότερους καταχωρητές από ότι μπορεί να διαθέσει ο επεξεργαστής.

Ο κώδικας του jam συνοπτικά.

```
for(int i=0; i < N; i += 4){
    num_0[0] = LVec[i]+RVec[i];
    num_0[1] = LVec[i+1]+RVec[i+1];
    num_0[2] = LVec[i+2]+RVec[i+2];
    num_0[3] = LVec[i+3]+RVec[i+3];

    ...
    ...

    ...
    ...
}
```

Μεταβάλλεται η σειρά των εντολών έτσι ώστε οι αντίστοιχες εντολές από κάθε block να είναι μαζί. Έπειτα είναι η δεύτερη εντολή από κάθε block, μετά είναι η τρίτη εντολή από κάθε block και ούτω καθεξής. Αυτή η διαδικασία βελτιώνει το pipelining του επεξεργαστή και επιταχύνει το πρόγραμμα. Αν οι πράξεις σε κάθε νέο block είναι ανεξάρτητες μεταξύ τους, ο κώδικας δυνητικά μπορεί να εκτελεστεί παράλληλα.

Μετατροπή σε SSE εντολές.

```
num_0[0] = LVec[i]+RVec[i];     |
num_0[1] = LVec[i+1]+RVec[i+1]; | => __m128 num_0 = _mm_add_ps( LVec128[i], RVec128[i]);
num_0[2] = LVec[i+2]+RVec[i+2]; |
num_0[3] = LVec[i+3]+RVec[i+3]; |
```

Κάθε τετράδα σειριακών εντολών μετατρέπεται σε μια εντολή SSE όπως φαίνεται παραπάνω. Χρειάζονται καινούριες μεταβλητές vector όπου η κάθε μία περιέχει μια τετράδα floats. Έτσι οι εντολές εκτελούνται παράλληλα και αυξάνεται η επίδοση του προγράμματος.

Οι κώδικες των unroll και jam υπάρχουν στα αρχεία preSSE_unroll.c και preSSE_jam.c τα οποία είναι ολοκληρωμένα και εκτελώντας τα φαίνεται η αύξηση στην επίδοση.

SSE+MPI

Το MPI προσφέρει παραλληλία στις πράξεις, καθώς μοιράζει το φόρτο τους σε διεργασίες. Κάθε κώδικας που χρησιμοποιεί MPI πρέπει να αρχικοποιεί το περιβάλλον όπου επικοινωνούν οι διεργασίες μεταξύ τους χρησιμοποιώντας την εντολή MPI_Init(). Η συγκεκριμένη εντολή δεν δημιουργεί τις διεργασίες, αυτό το κάνει το mpiexec στο script, αλλά επιτρέπει την επικοινωνία μεταξύ τους. Μετά μπορεί να χρησιμοποιηθεί οποιαδήποτε εντολή του API. Ακόμα είναι απαραίτητο κάθε διεργασία να γνωρίζει με πόσες συνυπάρχει και ποια είναι η σειρά της. Αυτό επιτυγχάνεται με τις εντολές MPI_Comm_size και MPI_Comm_rank αντίστοιχα.

Κάθε φορά που μοιράζεται μια δουλειά μεταξύ νημάτων ή διεργασιών πρέπει πρώτα να οριστεί το μέγεθος του task. Στη συγκεκριμένη περίπτωση έχει οριστεί ως μία επανάληψη του εσωτερικού for loop. Κάθε διεργασία υπολογίζει ποιά είναι η πρώτη και η τελευταία επανάληψη που θα επεξεργαστεί. Τα tasks μοιράζονται διαιρώντας το πλήθος τους με τον αριθμό των διεργασιών και στρογγυλοποιώντας προς τα πάνω. Για παράδειγμα εάν υπάρχουν 4 διεργασίες, η πρώτη θα εκτελέσει τις επαναλήψεις από 0 έως N/4. Επειδή εκτελούν ένα block από συγκεντρωμένες επαναλήψεις, χρειάζονται ένα κομμάτι των πινάκων και έτσι μειώνεται η πιθανότητα για cache miss. Εσωτερικά το for() και η διαδικασία εύρεσης του μέγιστου είναι ίδια με την προηγούμενη υλοποίηση.

Όπως και στη προηγούμενη υλοποίηση, προστέθηκε ένα κομμάτι κώδικα που υπολογίζει τα edge cases. Η διαφορά είναι ότι οι διεργασίες μοιράζονται τα στοιχεία που περισσεύουν και τα επεξεργάζονται σειριακά. Για παράδειγμα, αν υπάρχουν 4 διεργασίες μπορούν να περισσέψουν μέχρι 3 στοιχεία. Τότε η πρώτη διεργασία θα πάρει το πρώτο, η δεύτερη το επόμενο κλπ.

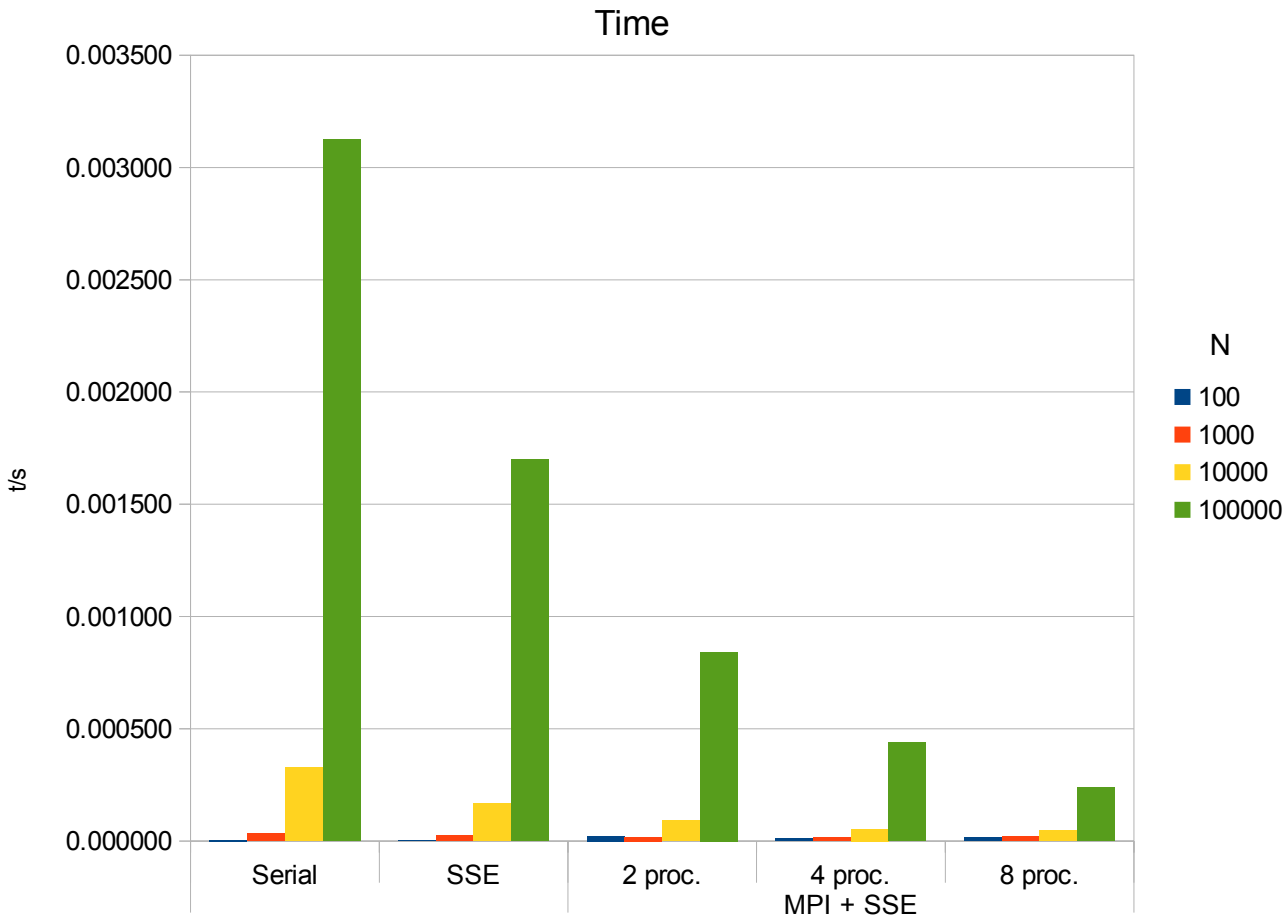
Για την εύρεση του μέγιστου από όλες τις διεργασίες χρησιμοποιείται η συνάρτηση MPI_Reduce() του API με είσοδο το MPI_MAXLOC. Ακόμα χρειάζεται ένα struct που περιέχει τον αριθμό και το rank της διεργασίας που το στέλνει. Ορίζονται οι τύποι αυτών των μεταβλητών με το MPI_FLOAT_INT και το περιβάλλον των διεργασιών με το MPI_COMM_WORLD. Επίσης πρέπει να οριστεί πόσα μηνύματα θα στείλει η κάθε διεργασία και ποια θα λάβει το αποτέλεσμα. Τέλος αυτό αποθηκεύεται σε ένα άλλο struct και η κύρια διεργασία μπορεί να το εκτυπώσει. Η συγκεκριμένη συνάρτηση είναι πιο σύνθετη από να στείλουν όλες οι διεργασίες το μέγιστο τους στη κύρια και αυτή να βρεί το τελικό, αλλά είναι πιο γρήγορη. Η αποστολή των μυνημάτων γίνεται μια φορά πρίν την εκτύπωση του μέγιστου ώστε η επικοινωνία και η ανταλλαγή δεδομένων να είναι η ελάχιστη δυνατή.

Τέλος καλείται η συνάρτηση MPI_Finalize() για να καταστραφεί το περιβάλλον επικοινωνίας των διεργασιών. Αυτή δεν εγγυάται αν συνεχίζουν ή σταματάνε οι διεργασίες παρά μόνο ότι συνεχίζει η κύρια. Για αυτό το λόγο πρέπει να μπει μετά τα free() των πινάκων.

Αποτελέσματα

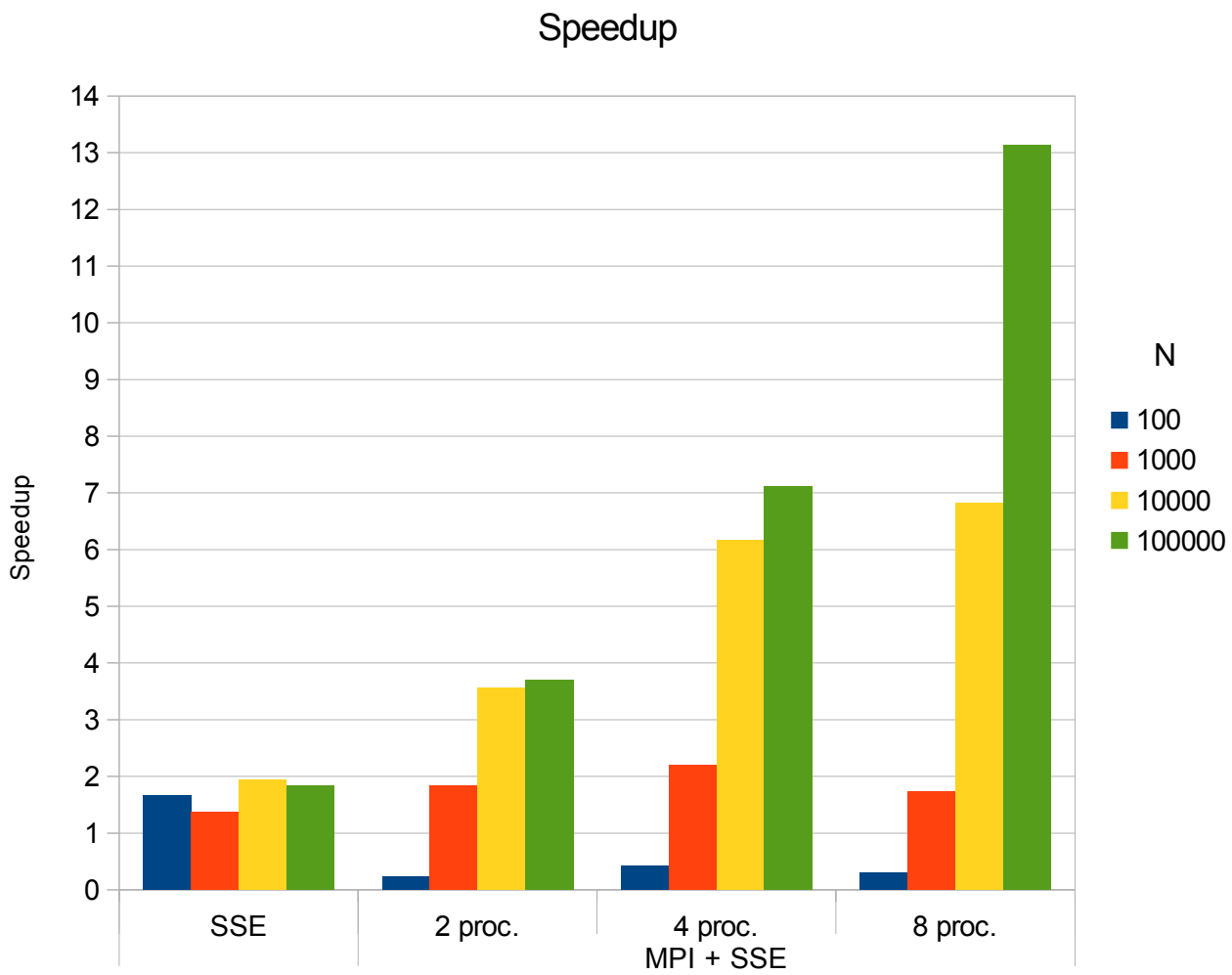
Ο υπολογιστής όπου γίνονται οι μετρήσεις έχει 8 πυρήνες.

Το παρακάτω γράφημα δείχνει τους χρόνους υπολογισμού(milisecond) για κάθε υλοποίηση με εισόδους: N = 100, 1000, 10000, 10000. Επισυνάπτεται και ο πίνακας με τις μετρήσεις επειδή στο γράφημα λόγο πολύ μεγάλων διαφορών δεν φαίνονται οι χρόνοι για μικρές εισόδους.



time			MPI + SSE		
N	Serial	SSE	2 proc.	4 proc.	8 proc.
100	0.000005	0.000003	0.000022	0.000012	0.000017
1000	0.000033	0.000024	0.000018	0.000015	0.000019
10000	0.000327	0.000168	0.000092	0.000053	0.000048
100000	0.003123	0.001698	0.000842	0.000439	0.000238

Με μικρές εισόδους οι χρόνοι δεν είναι ιδιαίτερα αξιόπιστοι και μεταβάλλονται αρκετά σε κάθε μέτρηση. Επίσης οι χρόνοι εκτέλεσης χρησιμοποιώντας MPI με 8 διεργασίες αυξάνονται αν λειτουργούν ταυτόχρονα και άλλα προγράμματα, διότι χρησιμοποιεί όλους τους επεξεργαστικούς πόρους του συστήματος μας.



Καταρχάς, από το παραπάνω γράφημα παρατηρούμε ότι χρησιμοποιώντας μόνο SSE το speedup είναι σταθερό για οποιοδήποτε μέγεθος εισόδου και περίπου στο 2. Δεν φτάνει το 4 που περιμένουμε ως max speedup γιατί η διαδικασία εύρεσης μέγιστου είναι αρκετά σύνθετη και οι εντολές της έχουν υψηλό latency, χωρίς αυτή το speedup πλησίαζε το 3. Γενικά το SSE δεν προσφέρει ιδιαίτερη βελτίωση όταν οι μεταβλητές σε ένα vector εξαρτώνται η μία από την άλλη. Αντίθετα, αν είναι ανεξάρτητες, προσφέρει μια εύκολη και αξιόπιστη λύση για την επιτάχυνση του προγράμματος.

Χρησιμοποιώντας MPI με μικρή είσοδο ο χρόνος εκτέλεσης μεγαλώνει γιατί το κόστος του overhead για την δημιουργία του περιβάλλοντος και την επικοινωνία είναι πολύ μεγαλύτερο από το κέρδος της παραλληλοποίησης του προγράμματος. Με N = 1000 το κέρδος με το κόστος εξισώνονται και ο χρόνος εκτέλεσης είναι περίπου ίδιος με αυτόν χωρίς τη χρήση του API. Με είσοδο N = 10000 αρχίζει να φαίνεται η βελτίωση που προσφέρει το MPI σε όλους τους αριθμούς διεργασιών, αλλά δεν πλησιάζει το max speedup me 8 διεργασίες. Τέλος για όλους τους αριθμούς διεργασιών και πολύ μεγάλη είσοδο το speedup πλησιάζει το μέγιστο, καθώς ο χρόνος του overhead έχει γίνει πολύ μικρός σε σχέση με τον χρόνο επεξεργασίας. Δηλαδή αυξάνοντας τα δεδομένα και άρα τις πράξεις, το computation to communication ratio βελτιώνεται, αφού η επικοινωνία είναι πάντα ίδια.

Γενικά το speedup του SSE είναι ανεξάρτητο του μεγέθους των δεδομένων, αλλά επηρεάζεται από την φύση των υπολογισμών. Αντίθετα το speedup του MPI είναι ανεξάρτητο από τις πράξεις, αλλά

επηρεάζεται από το μέγεθος των δεδομένων, την ανάγκη επικοινωνίας και τους πόρους που του διαθέτονται.

Συμπεράσματα και παρατηρήσεις

Το SSE είναι αξιόπιστο και αποδοτικό αν έχει δομηθεί ο κώδικας σωστά. Προσφέρει ένα πολύ μεγάλο πλήθος από εντολές το οποίο αυξάνεται με νέους επεξεργαστές της intel. Ακόμα λόγο του ακριβές και εκτεταμένου documentation, το API συμπεριφέρεται ακριβώς όπως περιμένουμε και δεν δημιουργούνται προβλήματα πέρα από αυτά του κώδικά μας, όπως με άλλα API σαν το OpenMP. Ένα μειονέκτημα είναι η αδυναμία εκτέλεσης αποδοτικών πράξεων μεταξύ των στοιχείων ενός vector, κάτι που αποφεύγεται ή μειώνεται δομώντας τον κώδικα καλύτερα. Τέλος το speedup έχει ανώτατο όριο πόσα δεδομένα χωράνε ταυτόχρονα στο pipeline του επεξεργαστή.

Το MPI προσφέρει μεγάλη μείωση του χρόνου λειτουργίας, αλλά εξαρτάται από πολλούς παράγοντες όπως αναφέρθηκε στα αποτελέσματα των speedups. Με τη βοήθεια προγραμμάτων όπως το LAM, το MPI έχει πολλές προοπτικές για speedup, γιατί μπορεί να συγχρονίσει παραπάνω από έναν υπολογιστή και να δημιουργήσει ένα cluster επεξεργαστών. Ακόμα υπάρχουν έτοιμες συναρτήσεις όπως η MPI_Reduce() που προσφέρουν παραπάνω ταχύτητα και λειτουργικότητα. Τέλος ένα αρνητικό του είναι υπάρχει αβεβαιότητα για το πως επηρεάζουν μερικές συναρτήσεις τον κώδικα. Για παράδειγμα σύμφωνα με το man page δε μπορούμε να γνωρίζουμε ακριβώς ποιες διεργασίες συνεχίζουν μετά το MPI_Finalize().

Εν κατακλείδι, το SSE, και γενικά τα SIMD πρότυπα, είναι ανεξάρτητο από το μεγέθους των δεδομένων, όμως θέτεται ανώτατο όριο στο speedup από τον επεξεργαστή και το πάχος του pipeline του. Αντίθετα, το MPI αντιμετωπίζει δυσκολίες ανάλογα τα δεδομένα αλλά έχει πολύ υψηλό scalability, με το ανώτατο όριο του speedup να ορίζεται από τους πόρους που του διαθέτονται.