Παράλληλη και Διανεμημένα Συστήματα

2η Εργασία : Distributed Bitonic Sort

Φλώρος Μαλιβίτσης Ορέστης 7796

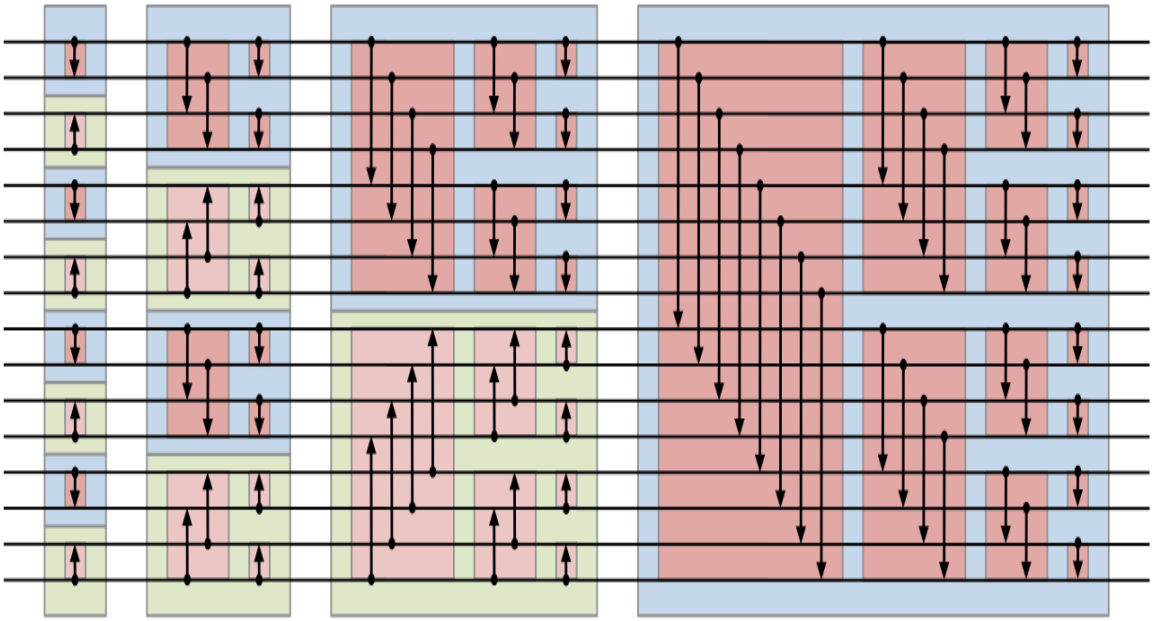
Χούτας Βασίλειος 7800

Contents

Περιγραφή Προβλήματος

Στο πλαίσιο αυτής της εργασίας αναπτύχθηκε κατανεμημένο πρόγραμμα σε **MPI** του οποίου ο σκοπός είναι η ταξινόμηση σε αύξουσα σειρά ενός συνόλου N ακεραίων αριθμών κάνοντας χρήση του αλγορίθμου **bitonic sort**, ο οποίος εκμεταλλεύεται τις ιδιότητες της διτονικής ακολουθίας για να ταξινομήσει την λίστα των αριθμών που δέχεται σαν είσοδο. Μία ακολουθία **x** λέγεται **διτονική** αν ισχύει $x_0 \leq x_1 \leq \cdots \leq x_k \geq x_{k+1} \geq \cdots \geq x_{n-1}$ για κάποιο $0 \leq k < n$ ή αν είναι μία κυκλική αντιμετάθεση μίας διτονικής ακολουθίας.

Στην πρώτη φάση του αλγορίθμου , τα στοιχεία της ακολουθίας αναδιατάσσονται με τέτοιο τρόπο ώστε να δημιουργηθεί μία διτονική ακολουθία. Αρχικά , θεωρούμε ότι έχουμε N ακολουθίες του ενός στοιχείου οι οποίες συγκρίνονται ανά δύο(όπως φαίνεται και στην παρακάτω εικόνα, όπου το τέλος του βέλους δείχνει πάντα προς την θέση όπου πρέπει να βρίσκεται το μεγαλύτερο στοιχείο[1]) και δημιουργούνται $\frac{N}{2}$ διτονικές ακολουθίες. Η διαδικασία συνεχίζεται και δημιουργούνται $\frac{N}{4}$ διτονικές ακολουθίες , όπου αρχικά συγκρίνουμε ανά 2 στοιχεία τα οποία «απέχουν» απόσταση 2 στο δίκτυο και στην συνέχεια σε απόσταση 1 για να είναι σε σωστή διάταξη , $\frac{N}{8}$ διτονικές ακολουθίες , κ.ο.κ. , ώσπου τελικά η συνολική ακολουθία είναι διτονική, οπότε και ξεκινάει το επόμενο στάδιο του αλγορίθμου. Εκεί , κάθε αριθμούς του $1^{ου}$ μισού της προκύπτους ακολουθίας συγκρίνεται με τον αντίστοιχο στο $2^{ο}$ ώστε να τοποθετηθεί στο σωστό ήμισυ της συνολικής λίστας. Τα παραπάνω βήματα επαναλαμβάνονται για κάθε υπο-λίστα ($\frac{N}{2}, \frac{N}{4}, \frac{N}{8}$, ….,1) ώσπου τελικά έχουμε την ταξινομημένη ακολουθία στην διάθεση μας .



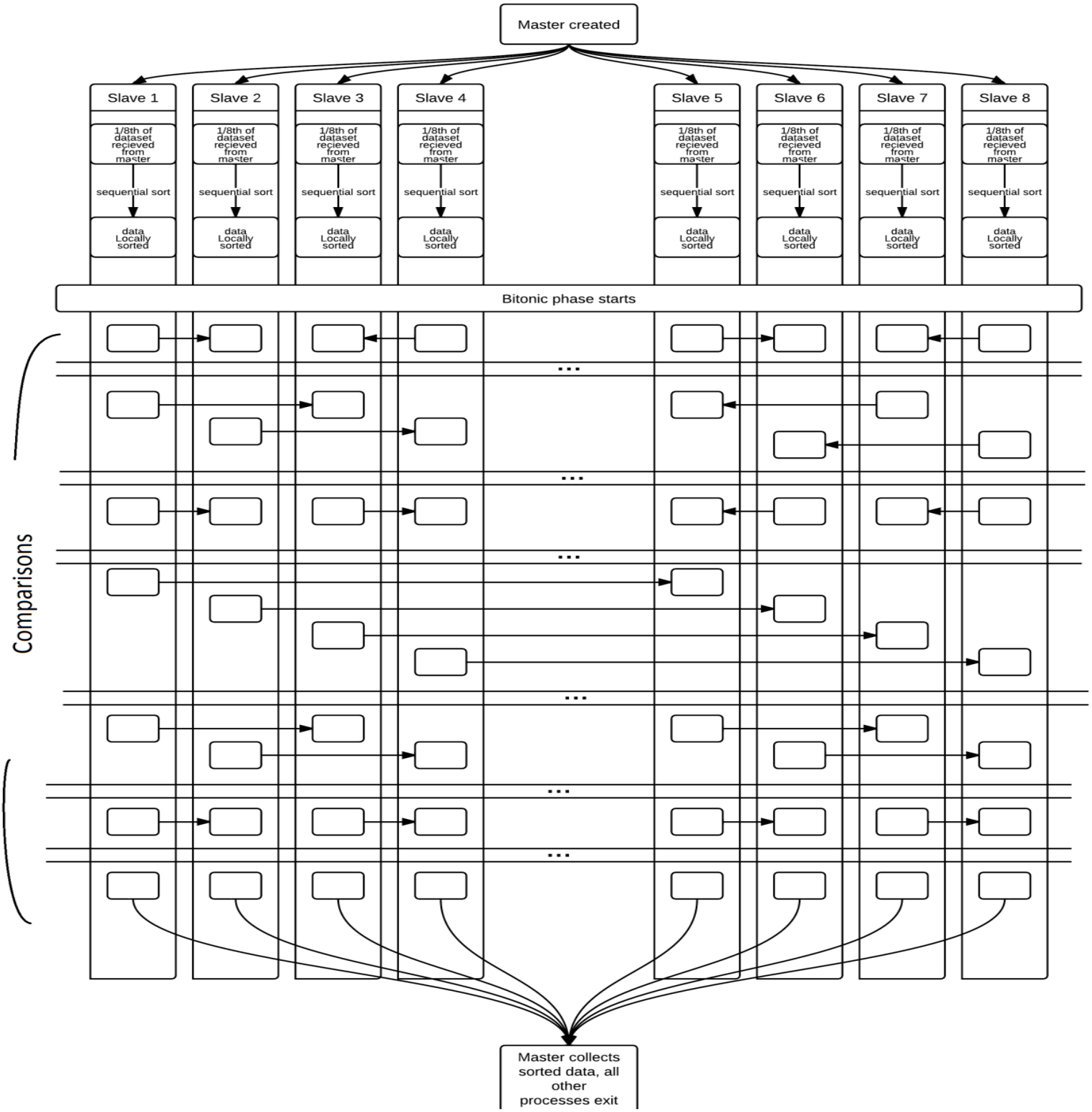
[1] Πηγή Εικόνας : http://en.wikipedia.org/wiki/Bitonic_sorter

Επειδή όμως ο αριθμός των στοιχείων είναι πολύ μεγάλος απαιτείται μία τροποποίηση του παραπάνω αλγορίθμου που θα μας επιτρέψει να τον υλοποιήσουμε σε κατανεμημένη μορφή. Έτσι , κάθε κόμβος του δικτύου,στον καθένα από τους οποίους αντιστοιχούμε και έναν μοναδικό δείκτη i ο οποίος καθορίζει την ταυτότητα του , έχει ένα μέρος των συνολικών στοιχείων ($N_i = \frac{N}{P}$, $i = 1,2,..,P$, $όπου\ P\ ο\ αριθμός\ των\ κόμβων/επεξεργαστών$) τα οποία σε πρώτη φάση ταξινομεί σε αύξουσα σειρά με κάνοντας χρήση του αλγορίθμου **qsort** της **standard library** της **C.** Στην συνέχεια , κατά την i-οστή φάση του αλγορίθμου ο κάθε κόμβος επικοινωνεί με αυτόν ο οποίος έχει το συμπληρωματικό του i-οστού bit του δείκτη του κόμβου(π.χ. στην $2^η$ φάση ο κόμβος 3(00**1**1) επικοινωνεί με τον 1(00**0**1)) και ο καθένας στέλνει όλα του τα στοιχεία στον άλλον, με τον έναν από αυτούς να κρατάει τα μεγαλύτερα από αυτά και τον άλλα τα μικρότερα. Η συγχώνευση των δύο λιστών γίνεται με τέτοιον τρόπο ώστε η προκύπτουσα ακολουθία να είναι ήδη ταξινομημένη σε αύξουσα σειρά και χρειάζεται 3N θέσεις στην μνήμη(N για τα στοιχεία του,N για τα στοιχεία του άλλου κόμβου και N για την νέα συγχωνευμένη λίστα με τα N αρχικά να αποσδεσμεύονται στην ολοκλήρωση για εξοικονόμηση χώρου). Με αυτόν τον τρόπο κατασκευάζεται η διτονική ακολουθία η οποία όπως παραπάνω δίνεται σαν είσοδος στο $2^ο$ στάδιο του αλγορίθμου. Έπειτα πραγματοποιούνται αμοιβαίες ανταλλαγές ώστε οι τοπικές ακολουθίες των κόμβων/επεξεργαστών να βρεθούν στην κατάλληλη θέση ώστε η προκύπτουτσα ακολουθία να είναι ταξινομημένη. Μία συνοπτική περιγραφή σε ψευδοκώδικα του παραπάνω αλγορίθμου είναι [2]:

Parallel Bitonic Sort Algorithm for processor Pk (for k := 0 : : : P 1)
d:= log P /* Διαστάσεις Υπερκύβου/πλήθος κόμβων */
sort(local datak) /* sequential sort */
/* Bitonic Sort follows */
for i:=1 to d do
window-id = Most Significant (d-i) bits of Pk
for j:=(i-1) down to 0 do
/* Η πράξη XOR χρησιμοποιείται για να γίνει toggle το κατάλληλο bit της ταυτότητας του κόμβου */
partner = Pk ^ (1 << j)
if ((window-id is even AND jth bit of Pk = 0) OR (window-id is odd AND jth bit of Pk = 1))
/* Καλείται η συνάρτηση η οποία πραγματοποιεί την επικοινωνία μεταξύ των 2 κόμβων και συγχωνεύει τα στοιχεία τους κρατώντας τα μικρότερα. */
then call **CompareLow(partner)**
/* Όμοια με την παραπάνω μονο που κατά την συγχώνευση κρατάει τα μεγαλύτερα στοιχεία */
else call **CompareHigh(partner)**
endif
endfor
endfor

[2] Πηγή : http://www.cs.rutgers.edu/~venugopa/parallel_summer2012/mpi_bitonic.html

Μία απλή απεικόνιση των παραπάνω είναι[3] :



[3] http://www.cs.rutgers.edu/~venugopa/parallel_summer2012/mpi_bitonic.html

Για τον έλεγχο ορθότητας των αποτελεσμάτων ελέγχουμε αρχικά αν η σειρά των στοιχείων είναι πράγματι αύξουσα (μέθοδος **ascendingSort**) και στην συνέχεια όταν ο κεντρικός κόμβος συγκεντρώσει τα στοιχεία ελέγχουμε ότι είναι ίσα με το αποτέλεσμα μίας quicksort η οποία εφαρμόζεται στο σύνολο της ακολουθίας.

Για την πραγματοποίηση των ελέγχων πρέπει να γίνει **compile** ο κώδικας με τις σημαίες **–DTEST** και **-DCOMPARE** . Το μέτρο αυτό λαμβάνεται καθώς η συγκέντρωση των στοιχείων στον κόμβο που ορίζεται ως master μπορεί να μην είναι δυνατή εξαιτίας του μεγάλου πλήθους των στοιχείων που μπορεί να είναι απαγορευτικό για την μνήμη ενός κόμβου.

Δομή του project

- **/omp/omp.c :** Ο αρχικός κώδικας της εργασίας με την προσθήκη παράλληλης υλοποίησης με openmp και σειριακής υλοποίησης με qsort
- **/results:** περιλαμβάνει αρχεία σχετικά με την εκτέλεση του κώδικα στο hellas.grid
- **/results/bitonic.c:** Η τελική μορφή της υλοποίησης σε MPI με μικρές αλλαγές.
- **/results/edit.py:** Αρχείο python για παραγωγή γραφημάτων.
- **/results/omp_run.py:** Script σε python για την παραγώγη και υποβολή εργασίων στο PBS. Χρήση κωδικα omp.c
- **/results/run.py:** Script σε python για την παραγώγη και υποβολή εργασίων στο PBS. Χρήση κωδικα bitonic.c
- **/src:** Όλα τα αρχεία για την υλοποίηση σε MPI
- **/src/bitonic.c:** Το κυρίως αρχείο με την main() και τις κλήσεις στις συναρτήσεις compare. Περιλαμβάνει επίσης TEST και συγκρίσεις με qsort.
- **/src/bitonic_merge.c:** Περιλαμβάνει την συνάρτηση compare().
- **/src/bitonic_tests.c:** Διάφορα tests. Χρησιμοποιείται μόνο μέσω 'make test'.
- **/src/general_functions.c:** Βοηθητικές συναρτήσεις.
- **/src/merge.c:** Περιλαμβάνει τις συναρτήσεις merge(), merge_low() και merge_high() για τις συγχωνεύσεις δύο τοπικών arrays σε μία. Επίσης την (αργή) merge_2N() που προσπαθεί να μην χρησιμοποίησει επιπλέον μνήμη για την συχώνευση.
- **/exercise.pdf:** Η εκφώνηση

Εξωτερικά Λινκ για τον κώδικα της εργασίας :

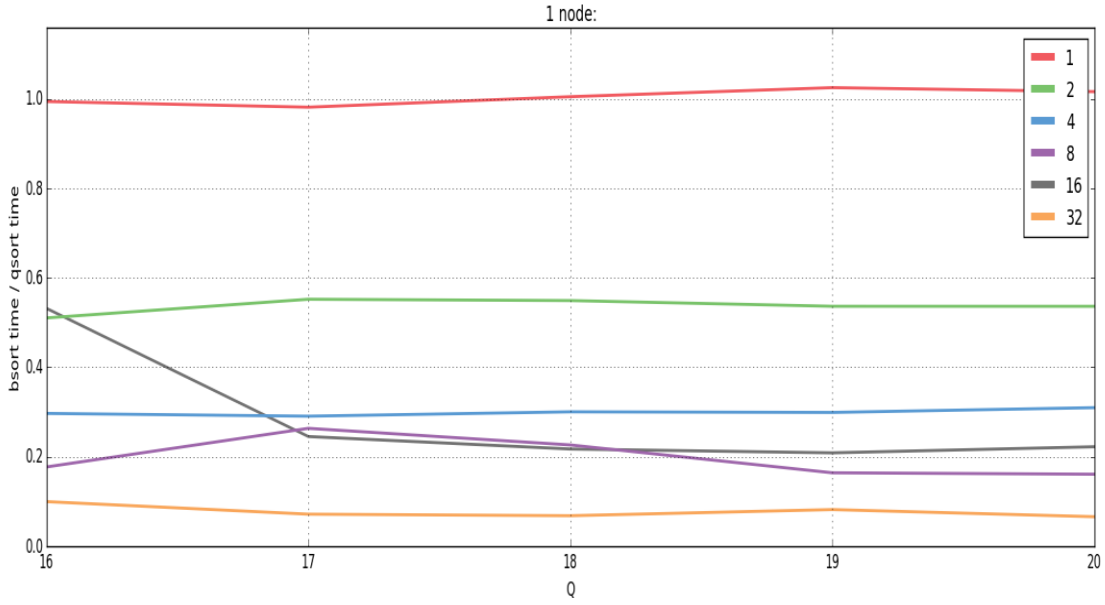
1. https://github.com/orestisf1993/mpi_bitonic_sort
2. https://github.com/vasilish/MPI_bitonic_sort

Γραφήματα

Σε αυτή την ενότητα παρουσιάζουμε τα γραφήματα με τα αποτελέσματα των εκτελέσεων της υλοποίησης μας που πραγματοποιήθηκαν στο **hellasgrid.** Ο άξονας y κάθε γραφήματος είναι ο λόγος του χρόνου που χρειάστηκε το κατανεμημένο πρόγραμμα προς τον χρόνο που χρειάζεται η σειριακή quicksort για την ταξινόμηση του συνόλου των δεδομένων.

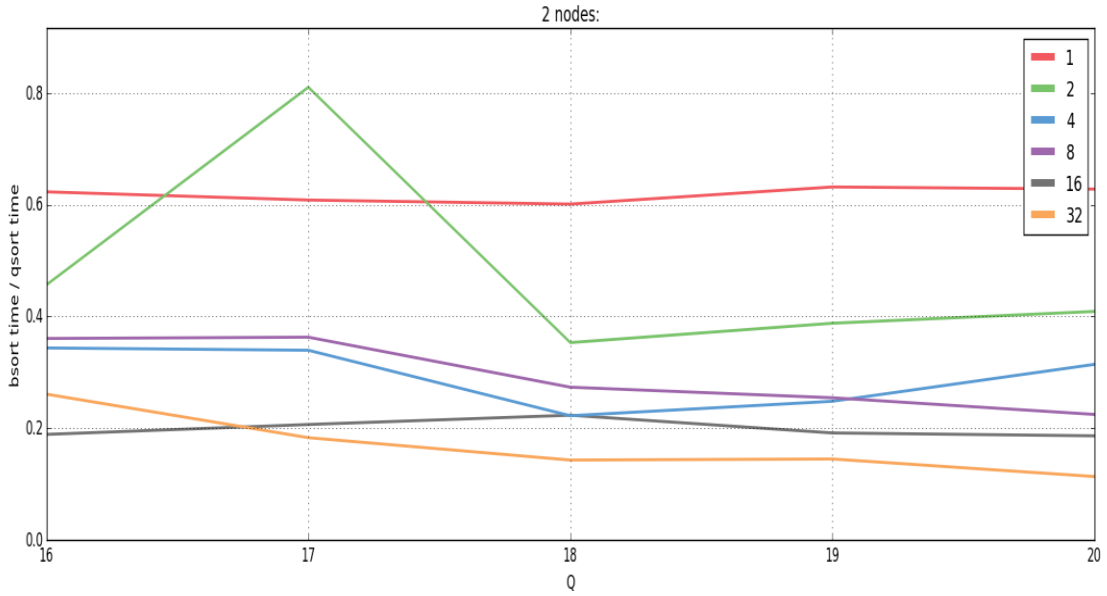
Αρχικά , παραθέτουμε τα διαγράμματα που δείχνουν τις επιδόσεις του αλγορίθμου καθώς αυξάνεται το πλήθος των στοιχείων.

Στο παρακάτων διάγραμμα οι εκτελέσεις των πειραμάτων έγιναν σε έναν κόμβο με αριθμό MPI processes από 1 εως και 32.

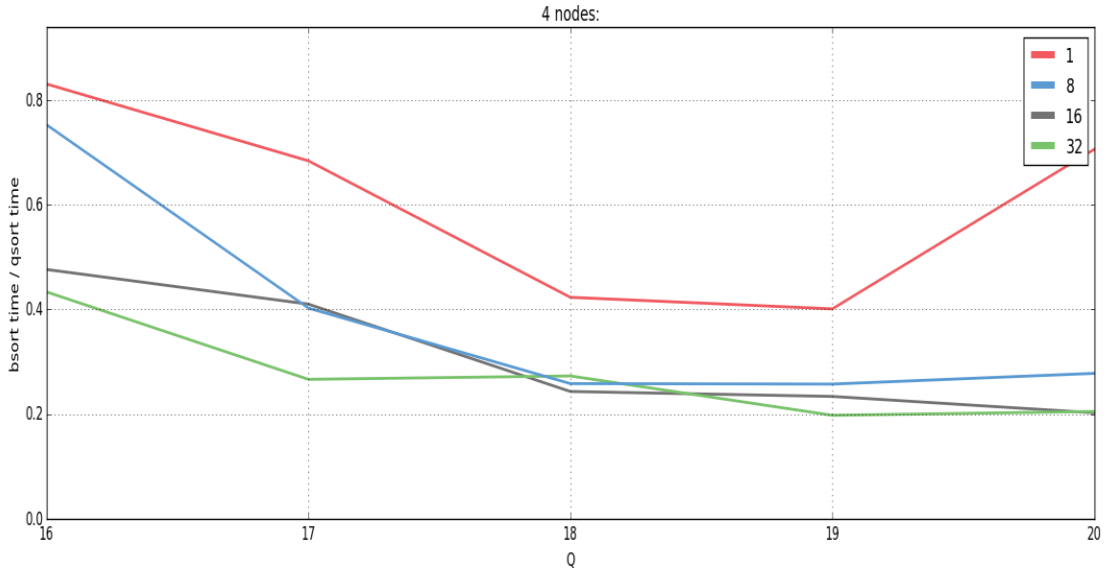


Όπως αναμενόταν , καθώς αυξάνουμε τον αριθμό των MPI processes βελτιώνεται η επίδοση του αλγορίθμου.

Ακολουθεί το αντίστοιχο διάγραμμα για την εκτέλεση των πειραμάτων σε 2 κόμβους όπου πάλι παρατηρούμε ότι η αυήξηση των διεργασιών οδηγεί σε βελτίωση της επίδοσης του αλγορίθμου :

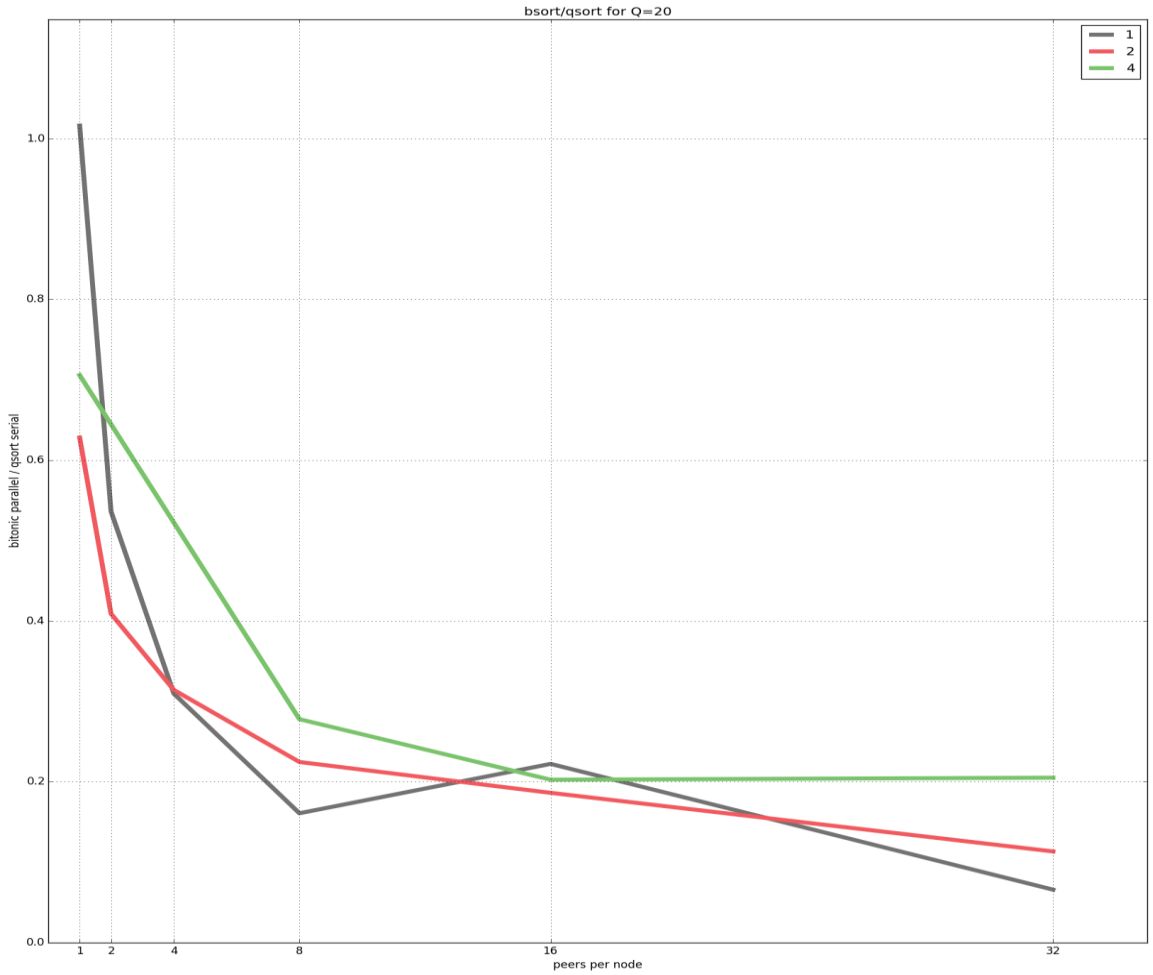


Στην συνέχεια , παρατίθεται το διάγραμμα για 4 κόμβους :



Παρατηρούμε ότι παρά την χρήση μεγάλου αριθμού διεργασιών (4*32 = 128) η επίδοση του αλγορίθμου για τα συγκεκριμένα μεγέθη προβλήματος είναι κατώτερη από την αντίστοιχη με 2 κόμβους (2*32 = 64) πιθανότατα λόγω του κόστους επικοινωνίας.

Έπειτα, παρουσιάζουμε ένα γράφημα όπου φαίνεται η αλλαγή της επίδοσης του αλγορίθμου καθώς αυξάνουμε τον αριθμό των επεξεργαστών σε κάθε κόμβο για σταθερό **Q=20**. Ταυτόχρονα σχεδιάζεται και διαφορετική καμπύλη για κάθε διαφορετικό αριθμό κόμβων .



Όπως αναμενόταν καθώς αυξάνεται ο αριθμός των επεξεργαστών ανά κόμβο η απόδοση του κατανεμημένου αλγορίθμου βελτιώνεται. Βέβαια για 4 κόμβους δεν μπορούμε να εξάγουμε πλήρη συμπεράσματα για μικρό αριθμό επεξεργαστώνκαθώς δεν δόθηκαν οι αντίστοιχοι πόροι από το hellasgrid(για 4 κόμβους με 2 και 4 επεξεργαστές) ενώ καθώς αυξάνεται το p φαίνεται να υστερεί καθώς το κόστος της επικοινωνίας ξεπερνά το όφελος από την παραλληλία για το συγκεκριμένο μέγεθος προβλήματος.

Τέλος , στο παρακάτω διάγραμμα επιβεβαιώνεται ότι η bitonic sort, είτε σε αναδρομική μορφή,είτε αν υλοποιηθεί με χρήση βρόγχων επανάληψης , είναι πολύ πιο αργή από την quicksort. Ακόμη , συγκρίνεται η σειριακή quicksort με μία έκδοση της iterative bitonic στην οποία έχει παραλληλοποιηθεί ο εσωτερικός βρόγχος που πραγματοποιεί τις συγκρίσεις και αλλαγές θέσεις των στοιχείων κάνοντας χρήση **OpenMP** πράγμα το οποίο φαίνεται να βελτιώνει την απόδοση του αλγορίθμου , χωρίς βέβαια να είναι συγκρίσιμη με την πλήρως παράλληλη εκδοχή του.