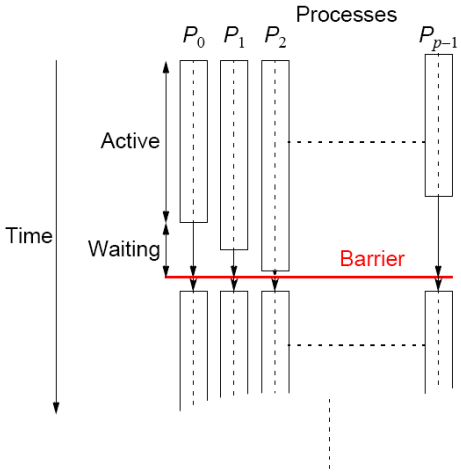
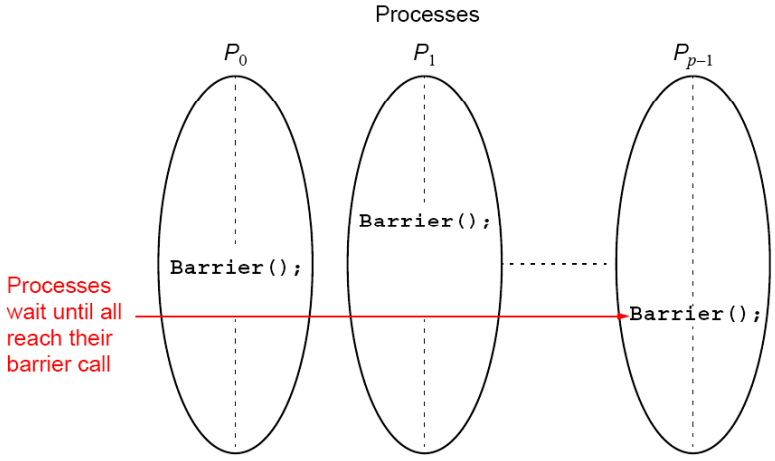
Σύγχρονισμένοι Υπολογισμοί

Σύγχρονισμένοι Υπολογισμοί

• Σε πολλά προβλήματα, οι διεργασίες του παράλληλου προγράμματος αφού εκτελέσουν μία σειρά από υπολογισμούς θα πρέπει να περιμένουν η μία την άλλη πριν προχωρήσουν περαιτέρω τους υπολογισμούς τους.
• Αυτή η αναμονή συνήθως επιβάλλεται από το γεγονός ότι κάθε διεργασία χρειάζεται τα αποτελέσματα των διεργασιών
• Σε μία πλήρως σύγχρονη εφαρμογή, όλες οι διεργασίες συγχρονίζονται ανά τακτά χρονικά διαστήματα.
• Ο βασικός μηχανισμός με τον οποίο οι διεργασίες συγχρονίζονται είναι ο μηχανισμός Barrier (φράγμα). Ο μηχανισμός εισάγεται στο σημείο εκτέλεσης στο οποίο κάθε διεργασία πρέπει να περιμένει
• Οι διεργασίες μπορούν να συνεχίσουν την εκτέλεση των εντολών τους πέρα από αυτό το σημείο όταν όλες οι διεργασίες έχουν φτάσει στο σημείο αυτό κατά την εκτέλεσή τους (ή, σε μερικές υλοποιήσεις όταν ένα συγκεκριμένος πλήθος διεργασιών έχει φτάσει σε αυτό το σημείο).



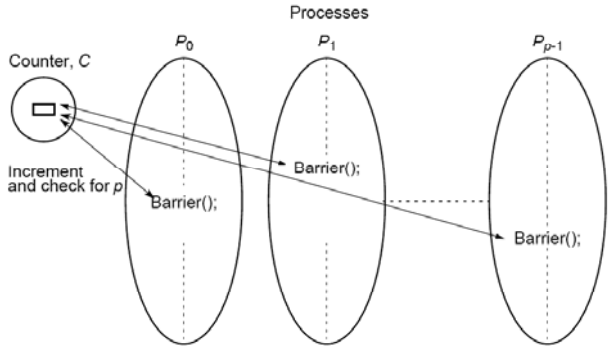
• Σε αυτό το παράδειγμα οι διαδικασίες φτάνουν στο Barrier σε διαφορετικές χρονικές στιγμές. Συγκεκριμένα η P_2 φτάνει αργότερα στο φράγμα σε σχέση με τις υπόλοιπες διεργασίες και έτσι όλες οι υπόλοιπες διεργασίες περιμένουν την P_2 να φτάσει στο φράγμα. Από τη στιγμή που όλες οι διεργασίες φτάσουν στο φράγμα, όλες οι διεργασίες μπορούν να συνεχίσουν την εκτέλεση των εντολών τους.
• Ο μηχανισμός των φραγμάτων βρίσκει εφαρμογή τόσο στο προγραμματισμό συστημάτων διαμοιραζόμενης μνήμης όσο και στα συστήματα κατανεμημένης μνήμης. Προς το παρόν, θα δοθούν λεπτομέρειες για τα φράγματα σε συστήματα κατανεμημένης μνήμης. Αργότερα, θα περιγραφεί αυτός ο μηχανισμός και στα συστήματα διαμοιραζόμενης μνήμης



• Στα συστήματα κατανεμημένης μνήμης, τα φράγματα παρέχονται μέσω συναρτήσεων βιβλιοθηκών
• Το MPI προσφέρει το μηχανισμό φράγματος μέσω της συνάρτησης MPI_Barrier(). Το μοναδικό όρισμα της συνάρτησης είναι ο communicator στα όρια του οποίου λαμβάνει χώρα ο συγχρονισμός. Κάθε διεργασία που ανήκει στο group του communicator εκτελεί την MPI_Barrier και αναστέλλει την εκτέλεσή της μέχρι όλες οι διεργασίες του group να εκτελέσουν τη συγκεκριμένη εντολή.
• Το MPI δεν προσδιορίζει τον τρόπο υλοποίησης του MPI_Barrier. Υπάρχουν πολλοί τρόποι υλοποίησης του βασικού μηχανισμού στα συστήματα κατανεμημένης μνήμης

Υλοποίηση φράγματος

• Η πρώτη υλοποίηση είναι συγκεντρωτική (Centralized) με τη βοήθεια μετρητή (γραμμικό φράγμα)
• Συγκεκριμένα, υπάρχει ένας μετρητής που μετράει το πλήθος των διεργασιών που έχουν φτάσει το φράγμα.
• Κάθε διεργασία που φτάνει στο φράγμα αυξάνει το μετρητή και στη συνέχεια ελέγχει αν ο μετρητής είναι μικρότερος από το συνολικό πλήθος p των διεργασιών. Σε αυτή τη περίπτωση, αναστέλλει τη εκτέλεσή της.
• Όταν ο μετρητής γίνει ίσος με p, όλες οι διεργασίες που έχουν ακινητοποιηθεί στο φράγμα, ενεργοποιούνται και συνεχίζουν την εκτέλεσή τους.



• Μία καλή υλοποίηση για το μηχανισμό του φράγματος θα πρέπει να παίρνει υπόψη ότι μια διεργασία μπορεί να χρησιμοποιήσει το φράγμα περισσότερες από μία φορά
• Είναι επίσης πιθανόν μία διεργασία να φτάσει στο φράγμα και δεύτερη φορά πριν οι προηγούμενες διεργασίες να έχουν φύγει από το φράγμα από τη πρώτη φορά.
• Οι δύο παραπάνω περιπτώσεις μπορούν εύκολα να αντιμετωπιστούν με τη χρήση δύο φάσεων στην υλοποίηση του γραμμικού φράγματος:
• Φάση άφιξης: Μία διεργασία εισέρχεται στη φάση άφιξης και δεν εγκαταλείπει αυτή τη φάση μέχρι όλες οι διεργασίες να εισέρθουν σε αυτή τη φάση
• Φάση αναχώρησης: Οι διεργασίες μεταβαίνουν στη φάση αναχώρησης και εγκαταλείπουν το φράγμα.

•Παράδειγμα:

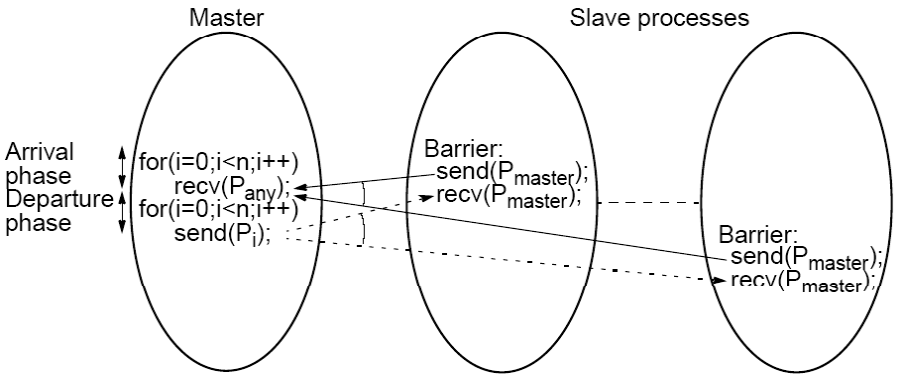
Master:

```
for (i = 0; i < n; i++)      /*count slaves as they reach barrier*/
        recv(Pany);
for (i = 0; i < n; i++)     /* release slaves */
        send(Pi);
```

Slave processes:

```
send(Pmaster);
recv(Pmaster);
```

Υλοποίηση του γραμμικού φράγματος με master-slave τεχνική

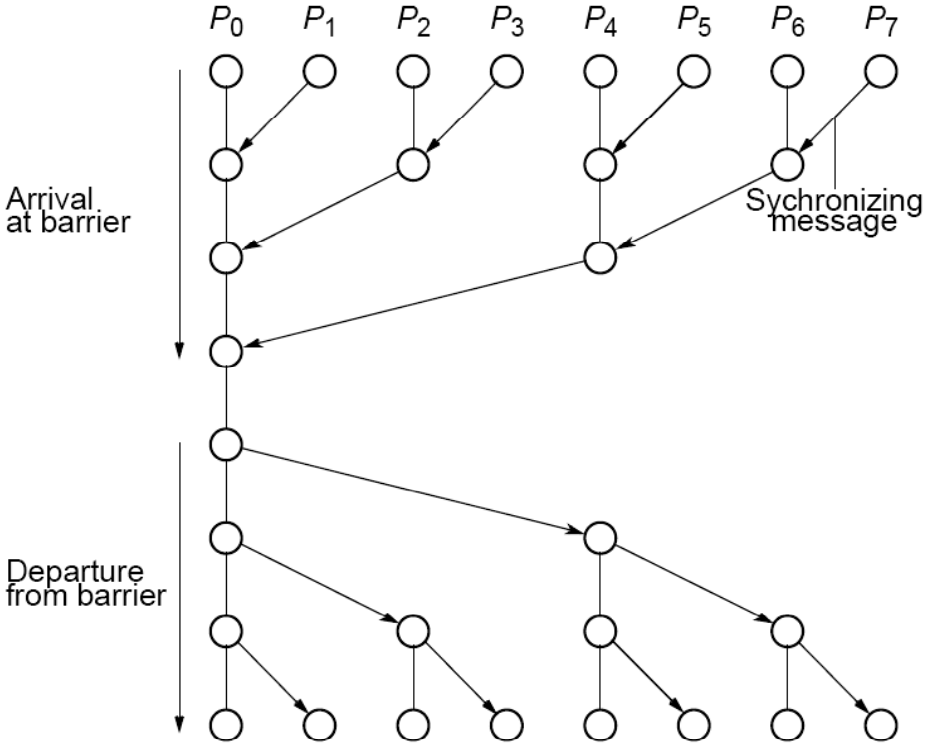


• Η master διεργασία ελέγχει τη μεταβλητή i η οποία είναι ο μετρητής που μετράει πόσες διεργασίες έχουν φτάσει στο φράγμα.
• Οι slave διεργασίες εισέρχονται με τυχαία σειρά στη φάση άφιξης ενώ εγκαταλείπουν τη φάση αναχώρησης με συγκεκριμένη σειρά η οποία καθορίζεται από τη ακολουθία εκτέλεσης των εντολών send από τη master διεργασία.
• Σημειώνεται ότι οι εντολές recv είναι blocking δηλ. η διεργασία που την εκτελεί περιμένει μέχρι να ληφθούν τα δεδομένα τα οποία αναμένει η recv

Η υλοποίηση φράγματος με δέντρο

• Το γραμμικό φράγμα έχει πολυπλοκότητα O(p) όπου p είναι το πλήθος των διεργασιών που συγχρονίζονται. Πιο αποδοτική υλοποίηση O(log p) βημάτων μπορούμε να πετύχουμε με υλοποίηση με τη βοήθεια δέντρου.
• Ας υποθέσουμε ότι έχουμε 8 διεργασίες, *P*0, *P*1, *P*2, *P*3, *P*4, *P*5, *P*6, *P*7:
• Σε πρώτο στάδιο, οι διεργασίες εκτελούν τις ακόλουθες εντολές:
Η *P*1 στέλνει μήνυμα στη *P*0 (όταν η *P*1 φτάσει στο φράγμα)
*H P*3 στέλνει μήνυμα στη *P*2 (όταν η *P*3 φτάσει στο φράγμα)
*H P*5 στέλνει μήνυμα στη *P*4 (όταν η *P*5 φτάσει στο φράγμα)
*H P*7 στέλνει μήνυμα στη *P*6 (όταν η *P*7 φτάσει στο φράγμα)
• Σε δεύτερο στάδιο, οι διεργασίες εκτελούν τις ακόλουθες εντολές:
Η *P*2 στέλνει μήνυμα στη *P*0 (Οι *P*2 κ' *P*3 έχουν φτάσει στο φράγμα)
*H P*6 στέλνει μήνυμα στη *P*4 (Οι *P*6 κ' *P*7 έχουν φτάσει στο φράγμα)
• Σε τρίτο στάδιο, οι διεργασίες εκτελούν τις ακόλουθες εντολές:
Η *P*4 στέλνει μήνυμα στη *P*0 (Οι *P*4, *P*5, *P*6, κ' *P*7 έχουν φτάσει στο φράγμα)
• Τέλος, η P0 τερματίζει τη φάση άφιξης (όταν η *P*0 φτάσει στο φράγμα και λάβει μήνυμα από την P4)
• Στη φάση αναχώρησης, η παραπάνω διαδικασία αντιστρέφεται με τη διεργασία P0 να στέλνει μήνυμα «αναχώρησης» σε όλες τις άλλες διεργασίες

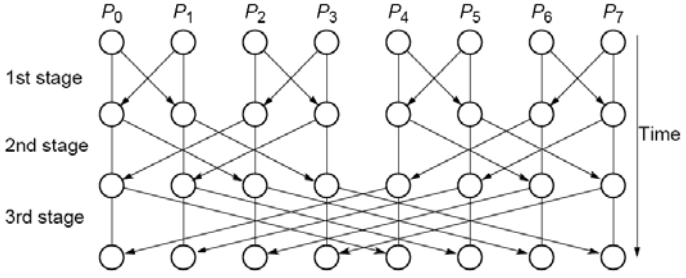
Υλοποίηση φράγματος με δέντρο



Υλοποίηση φράγματος με πεταλούδα

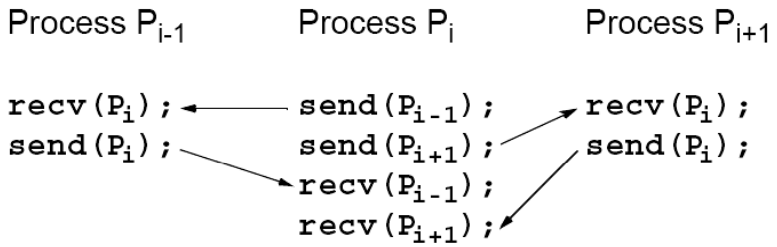
• Σε αυτή την υλοποίηση, ζεύγη διεργασιών συγχρονίζονται σε κάθε στάδιο. Κάθε τόξο στο σχήμα είναι επικοινωνία μεταξύ διεργασιών με send()/recv()
• Μετά την ολοκλήρωση των τριών σταδίων, κάθε διεργασία γνωρίζει ότι όλες οι άλλες διεργασίας έχουν φτάσει επίσης στο φράγμα
• Στη φάση s, διεργασία i συγχρονίζεται με τη διεργασία $i+2^{s-1}$ αν το πλήθος p των διεργασιών είναι δύναμη του 2. Αν το πλήθος p δεν είναι δύναμη του 2, η επικοινωνία είναι μεταξύ της διεργασίας i και της διεργασίας $(i+ 2^{s-1})$ mod p.
• Η χρονική πολυπλοκότητα της υλοποίησης είναι O(log p)

1st stage	$P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$
2nd stage	$P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$
3rd stage	$P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$



Τοπικός Συγχρονισμός

• Σε κάποια προβλήματα, οι διεργασίες πρέπει να συγχρονίζονται με ορισμένες μόνο διεργασίες και όχι με άλλες. Τέτοια παραδείγματα προκύπτουν όταν οι διεργασίες οργανώνονται σε πλέγμα ή σε διάταξη σωλήνωσης. Σε αυτές τις περιπτώσεις χρειάζεται να συγχρονιστούν μόνο με τους γείτονες τους.
• Στο παρακάτω παράδειγμα, η διεργασία P_i πρέπει να συγχρονιστεί και ανταλλάξει δεδομένα με τη διεργασία P_{i-1} και τη διεργασία P_{i+1} πριν συνεχίσει:

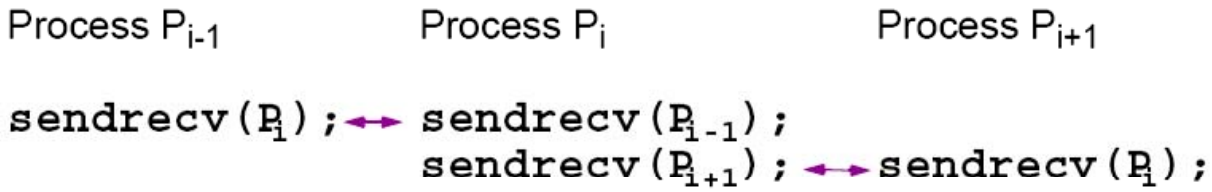


Στην ουσία δεν πρόκειται για τέλειο συγχρονισμό μεταξύ των τριών διεργασιών επειδή η διεργασία P_{i-1} συγχρονίζεται μόνο με την P_i και συνεχίζει μόλις η P_i το επιτρέψει. Παρόμοια, η διεργασία P_{i+1} συγχρονίζεται μόνο με την P_i.

Αδιέξοδο - Deadlock

• Στις υλοποιήσεις φράγματος με δέντρο, πεταλούδα ή στο τοπικό συγχρονισμό, όταν ένα ζεύγος διεργασιών επικοινωνεί με send()/recv(), υπάρχει πιθανότητα να προκύψει αδιέξοδο που θα ακινητοποιήσει τις εμπλεκόμενες διεργασίες.
• Το αδιέξοδο θα συμβεί εάν και οι δύο διεργασίες εκτελούν την εντολή send, χρησιμοποιώντας σύγχρονες ρουτίνες πρώτα (ή blocking ρουτίνες χωρίς επαρκή αποθηκευτικό χώρο (buffering)). Αυτό συμβαίνει επειδή καμία διεργασία δεν επιστρέφει από την εντολή send αφού θα περιμένει για το αντίστοιχο recv() που ποτέ δεν εκτελείται.
• Μία λύση στο πρόβλημα είναι η μία διεργασία να εκτελέσει την εντολή recv() πρώτα και στη συνέχεια την εντολή send() ενώ η άλλη διεργασία να εκτελέσει πρώτα την εντολή send() και μετά την εντολή recv().
• Για παράδειγμα, σε διάταξη σωλήνωσης, το αδιέξοδο μπορεί να αποφευχθεί, αν οι «ζυγές» διεργασίες εκτελούν την εντολή send() πρώτα και ενώ οι «μονές» διεργασίες εκτελούν την εντολή recv() πρώτα.

Επειδή η ανταλλαγή δεδομένων μεταξύ δύο διεργασιών είναι συχνή επικοινωνία, το MPI παρέχει την MPI_Sendrecv() που στέλνει και λαμβάνει δεδομένα και αποκλείει την πιθανότητα αδιεξόδου.
Το προηγούμενο παράδειγμα του τοπικού συγχρονισμού μπορεί να υλοποιηθεί με την εντολή MPI_Sendrecv() ως εξής:



Συγχρονισμένοι Υπολογισμοί

Οι συγχρονισμένοι υπολογισμοί μπορούν να χωριστούν σε δύο κατηγορίες:

- Πλήρως συγχρονισμένοι υπολογισμοί
- Τοπικά συγχρονισμένοι υπολογισμοί

Στους πλήρως συγχρονισμένους υπολογισμούς, όλες οι διεργασίες που εμπλέκονται στον υπολογισμό πρέπει να συγχρονίζονται.

Στους τοπικά συγχρονισμένους υπολογισμούς, οι διεργασίες πρέπει να συγχρονίζονται μόνο με ένα σύνολο λογικά «κοντινών» διεργασιών, και όχι με όλες τις διεργασίες που εμπλέκονται στον υπολογισμό

Παραδείγματα πλήρους συγχρονισμένου υπολογισμού

Παράλληλοι Υπολογισμοί Δεδομένων (Data Parallel Computations)

• Οι παράλληλοι υπολογισμοί δεδομένων απαιτούν έμμεσο συγχρονισμό μεταξύ των διεργασιών. Σε αυτούς τους υπολογισμούς, η ίδια λειτουργία εκτελείται σε διαφορετικά δεδομένα ταυτόχρονα, δηλ. παράλληλα.

• Αυτός ο τύπος παράλληλου υπολογισμού έχει σημαντικά πλεονεκτήματα επειδή:

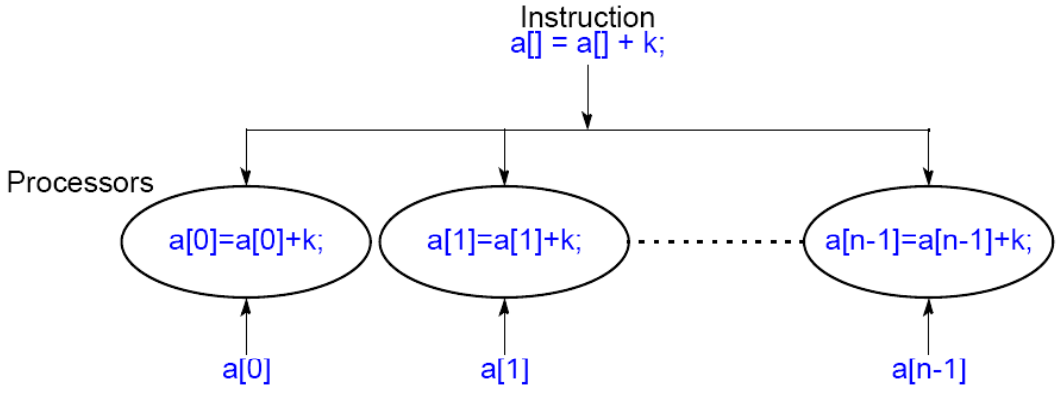
- Είναι εύκολος ο προγραμματισμός σε αυτό το μοντέλο παράλληλης εκτέλεσης. Ουσιαστικά έχουμε ένα μόνο πρόγραμμα.
- Μπορεί εύκολα να κλιμακωθεί σε μεγάλου μεγέθους προβλήματα.
- Πολλοί αριθμητικοί και μη αριθμητικοί υπολογισμοί μπορούν κωδικοποιηθούν ως παράλληλοι υπολογισμοί δεδομένων.

Παράδειγμα

Ένα παράδειγμα παράλληλου υπολογισμού δεδομένων είναι η πρόσθεση της ίδιας σταθεράς σε κάθε στοιχείο ενός πίνακα:

```
for (i = 0; i < n; i++)
        a[i] = a[i] + k;
```

Η πρόταση a[i] = a[i] + k μπορεί να εκτελείται ταυτόχρονα από πολλαπλές διεργασίες με κάθε μία διεργασία να αναλαμβάνει ένα ξεχωριστό στοιχείο a[i] (0 < i <= n).



Δομή forall

Σε πολλές γλώσσες παράλληλου υπολογισμού, ειδικές δομές χρησιμοποιούνται για να δηλώσουν παράλληλους υπολογισμούς δεδομένων.
Για παράδειγμα η δομή forall στο κώδικα που ακολουθεί

```
forall (i = 0; i < n; i++) {
        body
}
```

δηλώνει ότι μπορούν να εκτελεστούν ταυτόχρονα n στιγμιότυπα των προτάσεων που περιέχονται στο body.

Σε κάθε ένα από τα παραπάνω στιγμιότυπα, μία μόνο τιμή της μεταβλητής i είναι έγκυρη. Συγκεκριμένα, στο πρώτο στιγμιότυπο i=0, στο δεύτερο στιγμιότυπο i=1 κοκ. Η διαφορετική τιμή της μεταβλητής i σε κάθε στιγμιότυπο, επιτρέπει διαφοροποίηση της εκτέλεσης των παράλληλων διεργασιών (π.χ. μπορούν να προσπελαύνουν διαφορετικά στοιχεία ενός πίνακα)
Π.χ. για να προσθέσουμε μία σταθερά k σε κάθε στοιχείο ενός πίνακα a, μπορούμε να γράψουμε

```
forall (i = 0; i < n; i++)
        a[i] = a[i] + k;
```

Όταν η forall δομή υλοποιείται σε συστήματα κατανεμημένης μνήμης, απαιτείται συγχρονισμός μεταξύ των παράλληλων διεργασιών. Π.χ στο προηγούμενο παράδειγμα της άθροισης της σταθεράς k στα στοιχεία του πίνακα, η υλοποίηση της forall θα έχει ως εξής:

```
i = myrank;
a[i] = a[i] + k;        /* body */
barrier(mygroup);
```

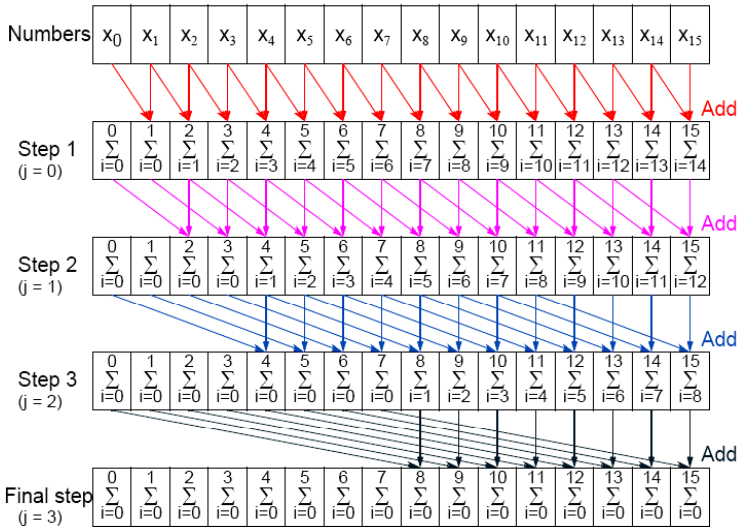
όπου **myrank** είναι η ταυτότητα της διεργασίας, ένας αριθμός μεταξύ 0 και n-1. Βασική υπόθεση στο παραπάνω κώδικα είναι ότι κάθε διεργασία i έχει πρόσβαση στα αντίστοιχο στοιχείο a[i] του πίνακα.

Ο Υπολογισμός Prefix Sum

• Ένα άλλο παράδειγμα παράλληλου υπολογισμού δεδομένων είναι ο υπολογισμός prefix sum. Στο πρόβλημα αυτό, δίνεται μία λίστα από n αριθμούς, *x*0, …, *xn*-1, και υπολογίζονται όλα τα μερικά αθροίσματα (δηλ., *x*0 + *x*1; *x*0 + *x*1 + *x*2; *x*0 + *x*1 + *x*2 + *x*3; …).

• Ο ίδιος υπολογισμός μπορεί να ορισθεί και με άλλους αντιμεταθετικούς τελεστές όπως ο πολλαπλασιασμός, το μέγιστο, το ελάχιστο κτλ.

• Ακολουθεί παράδειγμα υπολογισμού των μερικών αθροισμάτων για 16 στοιχεία



• Γενικά, ο υπολογισμός αυτός απαιτεί log n βήματα όπου υπάρχουν n αριθμοί (και n είναι δύναμη του 2). Στο βήμα j (0≤j<log n), n-2^j προσθέσεις εκτελούνται στις οποίες το στοιχείο x[i-2^j] προστίθεται στο στοιχείο x[i] για 2^j ≤ i <n.

• Ο ακολουθιακός κώδικας για τον υπολογισμό prefix sum

```
for (j=0; j <log n ; j++)
    for (i=2^j ; i < n; i++)
        x[i] = x[i] + x[i -2^j]
```

• Ο παράλληλος κώδικας για τον ίδιο υπολογισμό είναι:

```
for (j=0; j < log n; j++)
    forall (i=0; i < n; i++)
        if (i ≥ 2^j) x[i] = x[i] + x[i – 2^j]
```

Σύγχρονη Επανάληψη
(Σύγχρονος Παραλληλισμός)

• Ο όρος σύγχρονη επανάληψη ή σύγχρονος παραλληλισμός χρησιμοποιείται για να περιγράψει την επίλυση ενός προβλήματος με ένα επαναληπτικό αλγόριθμο όπου κάθε επανάληψη αποτελείται από ένα πλήθος διεργασιών οι οποίες ξεκινούν μαζί στην αρχή κάθε επανάληψης. Κάθε επανάληψη δεν μπορεί να αρχίσει μέχρι όλες οι διεργασίες να έχουν τελειώσει την προηγούμενη επανάληψη.

• Χρησιμοποιώντας τη δομή forall, μία σύγχρονη επανάληψη θα έχει ως εξής:

```
for (j = 0; j < n; j++)                    /*for each synch. iteration */
        forall (i = 0; i < N; i++) {    /*N procs each using*/
                body(i);                    /* specific value of i */
        }
```

• Η σύγχρονη επανάληψη σε συστήματα κατανεμημένης μνήμης θα έχει ως εξής:

```
for (j = 0; j < n; j++) {          /*for each synchr.iteration */
        i = myrank;                  /*find value of i to be used */
        body(i);
        barrier(mygroup);
}
```

Άλλο ένα παράδειγμα σύγχρονου υπολογισμού

Επίλυση ενός γενικού συστήματος γραμμικών εξισώσεων με επαναληπτική διαδικασία

Υποθέτουμε ότι έχουμε n εξισώσεις γενικής μορφής με n αγνώστους

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \ldots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

$$\vdots$$

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \ldots + a_{2,n-1}x_{n-1} = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \ldots + a_{1,n-1}x_{n-1} = b_1$$

$$a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \ldots + a_{0,n-1}x_{n-1} = b_0$$

όπου οι άγνωστοι είναι οι $x0$, $x1$, $x2$, … xn-1 ($0 <= i < n$).

Επιλύοντας την i-οστη εξίσωση ως προς τη μεταβλητή x_i θα έχουμε:

$$x_i = (1/a_{i,i})[b_i-(a_{i,0}x_0+a_{i,1}x_1+a_{i,2}x_2\ldots a_{i,i-1}x_{i-1}+a_{i,i+1}x_{i+1}\ldots+a_{i,n-1}x_{n-1})]$$

ή

$$x_i = \frac{1}{a_{i,i}}\left[b_i - \sum_{j \neq i} a_{i,j}x_j\right]$$

• Αυτή η εξίσωση δίνει τον άγνωστο x_i συναρτήσει των άλλων αγνώστων. Αυτή η εξίσωση μπορεί να χρησιμοποιηθεί επαναληπτικά για κάθε ένα από τους αγνώστους προκειμένου να πετύχουμε καλύτερη προσέγγιση στις τιμές των αγνώστων.

• Αυτή η επαναληπτική μέθοδος είναι γνωστή ως επανάληψη Jacobi. Σε κάθε επανάληψη της μεθόδου όλες οι τιμές των x ενημερώνονται μαζί.

• Μπορεί να αποδειχθεί ότι η μέθοδος Jacobi θα συγκλίνει αν οι διαγώνιες τιμές των *a* έχουν απόλυτη τιμή μεγαλύτερη από το άθροισμα των απολύτων τιμών όλων των υπόλοιπων a στη γραμμή (ο πίνακας των a λέγεται σε αυτή την περίπτωση διαγώνια κυρίαρχος (*diagonally dominant)*) δηλ. αν

$$\sum_{j \neq i} |a_{i,j}| < |a_{i,i}|$$

• Αυτή η συνθήκη είναι ικανή αλλά όχι απαραίτητη συνθήκη

Συνθήκη τερματισμού

Η επαναληπτική διαδικασία θα πρέπει να τερματίζει όταν έχει επιτευχθεί σύγκλιση στις τιμές των x. Μία απλή προσέγγιση είναι να συγκρίνουμε τις τιμές των x που προέκυψαν στη τρέχουσα επανάληψη με τις τιμές των x από τη προηγούμενη επανάληψη. Η επαναληπτική διαδικασία τερματίζεται όταν όλες οι τιμές διαφέρουν κάτω από ένα όριο, δηλ., όταν

$$|x_i^t - x_i^{t-1}| < \text{error tolerance}$$

για όλες τιμές του i, όπου x_i^t είναι η τιμή του x_i μετά από την t-οστη επανάληψη και x_i^{t-1} είναι η τιμή του x μετά την (t-1)-οστη επανάληψη

Παράλληλος κώδικας για τη τεχνική Jacobi

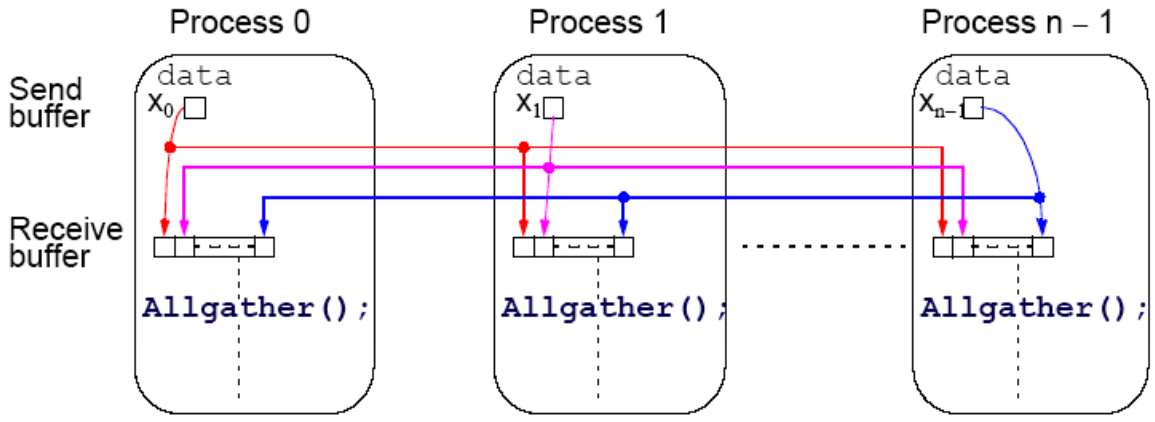
• Υποθέτουμε για κάθε άγνωστο υπάρχει μία ξεχωριστή διεργασία που υπολογίζει την τιμή του και ότι κάθε διεργασία θα κάνει το ίδιο πλήθος επαναλήψεων.
• Μετά το τέλος μίας επανάληψης, οι τρέχουσες τιμές των αγνώστων θα πρέπει να μεταδοθούν σε όλες τις διεργασίες του προγράμματος, αφού κάθε διεργασία πρέπει να γνωρίσει τις τιμές αυτές για να εκτελέσει την επόμενη επανάληψη. Στο τέλος κάθε επανάληψης πρέπει να υπάρχει ένα φράγμα για να επιτυγχάνεται συγχρονισμός μεταξύ των διεργασιών
• Η διεργασία *Pi* του παράλληλου προγράμματος εκτελεί τον ακόλουθο κώδικα:

```
x[i] = b[i];                                    /*initialize unknown*/
for (iteration = 0; iteration < limit; iteration++) {
        sum = -a[i][i] * x[i];
        for (j = 0; j < n; j++)                 /* compute summation */
                sum = sum + a[i][j] * x[j];
        new_x[i] = (b[i] - sum) / a[i][i];      /* compute unknown */
        allgather(&new_x[i]);                   /*bcast/rec values */
        global_barrier();                       /* wait for all procs */
}
```

• Με την εντολή **allgather()**, κάθε διεργασία στέλνει τη νέα τιμή για το **x[i]** σε κάθε άλλη διεργασία και λαμβάνει τις τιμές όλων των υπολοίπων αγνώστων από τις άλλες διεργασίες. Ανάλογα την υλοποίηση, η εντολή allgather μπορεί προσφέρει έμμεσο συγχρονισμό στις διεργασίες αφού μία διεργασία δεν μπορεί να επιστρέψει από αυτή την εντολή πριν λάβει όλες τις τιμές από τις υπόλοιπες διεργασίες. Σε αυτή τη περίπτωση το φράγμα δεν χρειάζεται

Η λειτουργία Allgather.

• Τα ίδια δεδομένα συλλέγονται από τις διεργασίες. Το δεδομένο από τη διεργασία 0 τοποθετείται πρώτο στους buffers λήψης, το δεδομένο από τη διεργασία 1 τοποθετείται δεύτερο σε αυτούς τους buffers κοκ.



Διαμέριση -Partitioning

Στη περιγραφή της παράλληλης υλοποίησης της Jacobi τεχνικής, θεωρήσαμε ότι κάθε διεργασία αναλαμβάνει ένα μόνο στοιχείο εισόδου (άγνωστος x). Συνήθως όμως το πλήθος των επεξεργαστών (διεργασιών) είναι πολύ μικρότερο από το πλήθος των δεδομένων εισόδου του προβλήματος και επομένως οι επεξεργαστές πρέπει να αναλάβουν περισσότερα τους ενός στοιχεία εισόδου.

Υπάρχουν δύο τρόποι διαμοιρασμού των αγνώστων στις διαθέσιμες διεργασίες (υποθέτοντας ότι το πλήθος των στοιχείων n διαιρείται ακριβώς με το πλήθος p των διεργασιών):

block κατανομή (allocation) – Σε αυτή την κατανομή, ομάδες διαδοχικών αγνώστων κατανέμονται στις διεργασίες με αυξανόμενη σειρά. Πιο συγκεκριμένα η διεργασία P_i αναλαμβάνει τους αγνώστους $x_{i*n/p},...,x_{(i+1)*n/p-1}$ (i=0,..,p-1) .

*κυκλική (cyclic) κατανομή (*allocation) – Σε αυτή την κατανομή, οι άγνωστοι κατανέμονται κυκλικά στις διεργασίες. Συγκεκριμένα, η διεργασία P_0 αναλαμβάνει τους αγνώστους x_0, x_p, x_{2p}, …, $x_{(n/p-1)p}$, η διεργασία P_1 αναλαμβάνει τους αγνώστους x_1, x_{p+1}, x_{2p+1}, …, $x_{(n/p-1)p+1}$, κοκ.

Η κυκλική κατανομή δεν έχει κάποιο συγκεκριμένο πλεονέκτημα σε σχέση με τη block κατανομή. Στη πράξη, η κυκλική κατανομή μπορεί να παρουσιάζει δυσκολίες υλοποίησης αφού απαιτείται πιο πολύπλοκος χειρισμός των δεικτών των αγνώστων.

Ανάλυση πολυπλοκότητας της παράλληλης υλοποίησης της τεχνικής Jacobi

• Σε κάθε επανάληψη, κάθε μία από τις p συνολικά διεργασίες αναλαμβάνει τον υπολογισμό των τιμών n/p αγνώστων.

• Για το υπολογισμό της τιμής ενός αγνώστου απαιτούνται O(n) πράξεις. Άρα συνολικά κάθε διεργασία εκτελεί $O(n^2/p)$ πράξεις σε κάθε επανάληψη.

• Στο τέλος κάθε επανάληψης υπάρχει η εντολή all-gather και ας υποθέσουμε ότι η υλοποίηση της παρέχει έμμεσο συγχρονισμό στις εμπλεκόμενες διεργασίες και επομένως δεν χρειάζεται η εντολή του φράγματος.

• Στη λειτουργία all-gather κάθε διεργασία στέλνει στις υπόλοιπες διεργασίες τις τιμές των n/p αγνώστων που έχει υπολογίσει στη τρέχουσα επανάληψη. Αν αυτή η επικοινωνία υλοποιηθεί με p διαδοχικές εκπομπές (broadcasts) ο συνολικός χρόνος επικοινωνίας θα είναι $p(T_{startup} + n/p * t_{data})$ όπου η παράσταση μέσα στη παρένθεση είναι ο χρόνος για να στείλει μία διεργασία n/p στοιχεία σε όλες τις υπόλοιπες. Γενικά, ο χρόνος εκπομπής εξαρτάται από το διασυνδετικό δίκτυο που συνδέει τους επεξεργαστές.

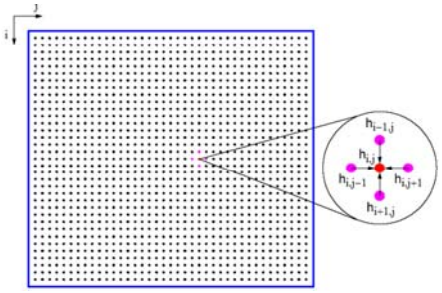
• Για t επαναλήψεις της μεθόδου Jacobi, οι παραπάνω χρόνοι πρέπει να πολλαπλασιαστούν επί t.

Τοπικός Σύγχρονος Υπολογισμός

Το πρόβλημα της κατανομής θερμότητας

• Στο πρόβλημα αυτό, έχουμε μία περιοχή στην οποία οι τιμές της θερμοκρασίας είναι γνωστές μόνο στις άκρες της. Το ζητούμενο είναι εύρεση των τιμών της θερμοκρασίας και στο εσωτερικό της περιοχής.

• Για την εύρεση της κατανομής θερμοκρασίας, διαιρούμε την περιοχή όπως φαίνεται στο σχήμα σε ένα λεπτό πλέγμα σημείων. Έστω h_{ij} η θερμοκρασία της περιοχής στο σημείο (i,j).

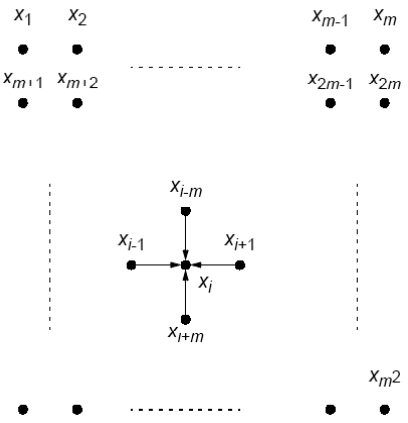


•Γίνεται η υπόθεση ότι η θερμοκρασία σε ένα εσωτερικό σημείο είναι ο μέσος όρος των θερμοκρασιών των τεσσάρων γειτονικών σημείων δηλ.

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

• Οι μόνες θερμοκρασίες που γνωρίζουμε είναι οι θερμοκρασίες των εξωτερικών σημείων δηλ. τις τιμές των μεταβλητών h_{i*} και h_{*j} για i=0,m και για j=0,m. Τα εσωτερικά σημεία έχουν συντεταγμένες 0< i,j <m.

• Εφαρμόζοντας επαναληπτικά τη παραπάνω σχέση, μπορούμε να υπολογίσουμε τις τιμές h_{ij} οι οποίες θα συγκλίνουν μετά από μερικές επαναλήψεις



• Αν αριθμήσουμε τα σημεία του πλέγματος από αριστερά προς τα δεξιά και από πάνω προς τα κάτω, τότε η προηγούμενη εξίσωση που δίνει τη θερμοκρασία σε ένα σημείο συναρτήσει των θερμοκρασιών σε γειτονικά σημεία, μπορεί να γραφεί ως εξής:

$$x_i = \frac{x_{i-1} + x_{i+1} + x_{i-m} + x_{i+m}}{4}$$

• Μπορεί να γραφεί επίσης ως γραμμική εξίσωση των αγνώστων x_{i-m}, x_{i-1}, x_{i+1}, and x_{i+m} :

$$x_{i-m} + x_{i-1} - 4x_i + x_{i+1} + x_{i-m} = 0$$

• Συνολικά θα έχουμε ένα αραιό σύστημα με m^2 εξισώσεις και m^2 αγνώστους. Αυτός ο τρόπος επίλυσης είναι γνωστός και ως μέθοδος πεπερασμένων διαφορών (finite difference method)

Ακολουθιακός κώδικας

Χρησιμοποιώντας ένα σταθερό πλήθος επαναλήψεων και χρησιμοποιώντας τη δισδιάστατη αναπαράσταση για τη δεικτοδότηση των αγνώστων, ο ακολουθιακός κώδικας για το πρόβλημα της κατανομής θερμότητας έχει ως εξής:

```
for (iteration = 0; iteration < limit; iteration++) {
  for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
        g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
    for (i = 1; i < n; i++)                    /* update points */
      for (j = 1; j < n; j++)
        h[i][j] = g[i][j];
}
```

Αν αντίθετα η μέθοδος σταματάει μόλις η επιθυμητή ακρίβεια στις τιμές της θερμοκρασίας επιτευχθεί, ο κώδικας θα έχει ως εξής.

```
do {
  for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
      g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);

    for (i = 1; i < n; i++)                  /* update points */
     for (j = 1; j < n; j++)
       h[i][j] = g[i][j];

    continue = FALSE; /* indicates whether to continue */
    for (i = 1; i < n; i++)/* check each pt for convergence */
      for (j = 1; j < n; j++)
        if (!converged(i,j) {/* point found not converged */
          continue = TRUE;
          break;
        }
} while (continue == TRUE);
```

Η σύγκλιση ελέγχεται αφότου οι θερμοκρασίες σε όλα τα σημεία έχουν υπολογιστεί. Η ρουτίνα converged(i,j) επιστρέφει TRUE αν το στοιχείο g[i][j] έχει συγκλίνει στην απαιτούμενη ακρίβεια.

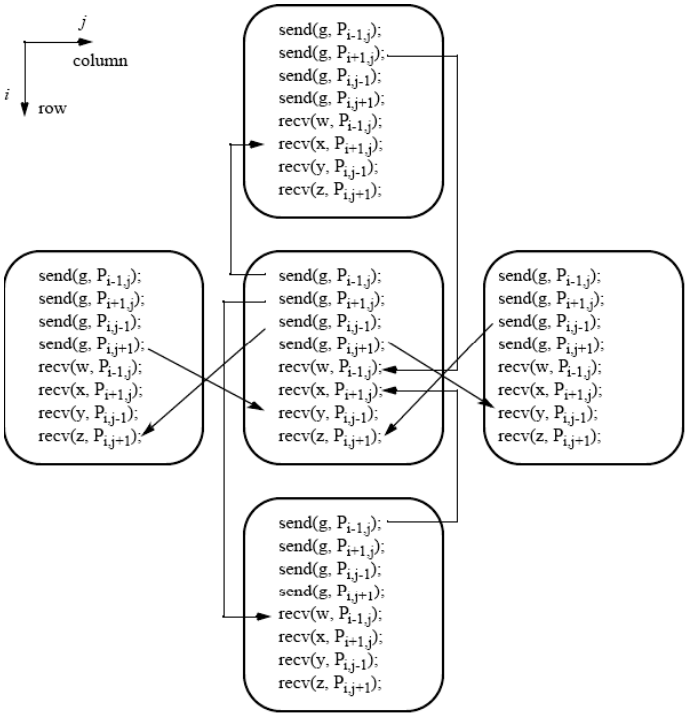
Παράλληλος κώδικας

• Θεωρούμε ότι κάθε διεργασία αναλαμβάνει ένα σημείο της επιφάνειας. Έστω $P_{i,j}$ είναι η διεργασία που αναλαμβάνει το σημείο με συντεταγμένες (i,j).
• Εκτός από τα εξωτερικά σημεία στα οποία χρειάζεται ειδικός χειρισμός, ο κώδικας που εκτελεί μία διεργασία $P_{i,j}$ που αντιστοιχεί σε ένα εσωτερικό σημείο (i,j) θα έχει ως εξής :

```
for (iteration = 0; iteration < limit; iteration++) {
    g = 0.25 * (w + x + y + z);
    send(&g, P_{i-1,j});   /* non-blocking sends */
    send(&g, P_{i+1,j});
    send(&g, P_{i,j-1});
    send(&g, P_{i,j+1});
    recv(&w, P_{i-1,j});   /* synchronous receives */
    recv(&x, P_{i+1,j});
    recv(&y, P_{i,j-1});
    recv(&z, P_{i,j+1});
}
```

Local barrier

• Στο παραπάνω κώδικα, είναι σημαντικό να χρησιμοποιήσουμε **send()**s τα οποία δεν αναστέλλουν την εκτέλεση της διεργασίας περιμένοντας τη λήψη δεδομένων από την αντίστοιχη λειτουργία **recv()**. Σε διαφορετική περίπτωση θα είχαμε αδιέξοδο αφού κάθε διεργασία θα περίμενε για ένα recv που ποτέ δεν εκτελείται.
• Αντίθετα για τη λήψη των δεδομένων πρέπει να χρησιμοποιήσουμε σύγχρονα **recv()**s τα οποία θα περιμένουν για τα αντίστοιχα **send()**s. Με τη χρήση σύγχρονων recvs, επιτυγχάνεται ο συγχρονισμός κάθε διεργασίας με τους τέσσερις γείτονες.



Η μεταφορά δεδομένων μεταξύ μίας διεργασίας και των γειτόνων της φαίνεται στο σχήμα

• Για την υλοποίηση της περίπτωσης όπου οι διεργασίες σταματούν όταν φτάνουν στην απαιτούμενη ακρίβεια, πρέπει να υπάρχει μία διεργασία master η οποία πρέπει να ειδοποιείται όταν οι διεργασίες έχουν σταματήσει. Μία διεργασία μπορεί να στέλνει τα δεδομένα της στη διεργασία master όταν η επιθυμητή ακρίβεια έχει επιτευχθεί τοπικά. Ο παράλληλος κώδικας θα έχει ως εξής

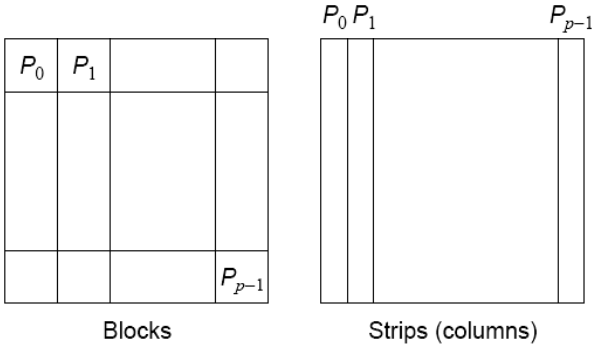
```
iteration = 0;
do {
    iteration++;
    g = 0.25 * (w + x + y + z);
    send(&g, P_{i-1,j});        /* locally blocking sends */
    send(&g, P_{i+1,j});
    send(&g, P_{i,j-1});
    send(&g, P_{i,j+1});
    recv(&w, P_{i-1,j});        /* locally blocking receives */
    recv(&x, P_{i+1,j});
    recv(&y, P_{i,j-1});
    recv(&z, P_{i,j+1});
} while((!converged(i, j)) || (iteration < limit));
send(&g, &i, &j, &iteration, P_master);
```

Για να χειριστούμε τις διεργασίες που αναλαμβάνουν εξωτερικά σημεία, μπορούμε να χρησιμοποιήσουμε τις συντεταγμένες της διεργασίας P_{ij} για να προσδιορίσουμε τη θέση της. Ο κώδικας έχει ως εξής:

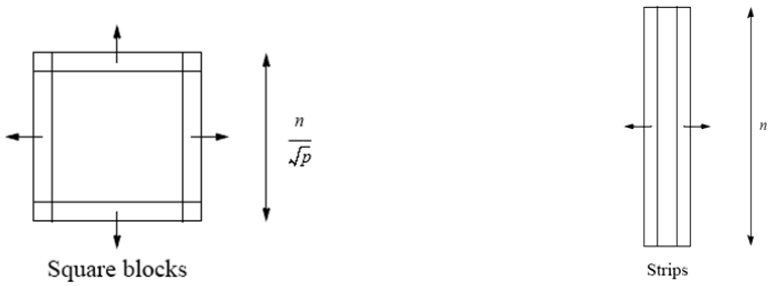
```
if (last_row) w = bottom_value;
if (first_row) x = top_value;
if (first_column) y = left_value;
if (last_column) z = right_value;
iteration = 0;
do {
  iteration++;
  g = 0.25 * (w + x + y + z);
  if !(first_row) send(&g, P_{i-1,j});
  if !(last_row) send(&g, P_{i+1,j});
  if !(first_column) send(&g, P_{i,j-1});
  if !(last_column) send(&g, P_{i,j+1});
  if !(last_row) recv(&w, P_{i-1,j});
  if !(first_row) recv(&x, P_{i+1,j});
  if !(first_column) recv(&y, P_{i,j-1});
  if !(last_column) recv(&z, P_{i,j+1});
} while((!converged) || (iteration < limit));
send(&g, &i, &j, iteration, P_master);
```

Διαμέριση

Στις περισσότερες περιπτώσεις, περισσότερα του ενός σημείου αντιστοιχούν σε κάθε διεργασία αφού το πλήθος των σημείων της υπό μελέτης περιοχής είναι πολύ περισσότερα από το πλήθος των διαθέσιμων διεργασιών
Τα σημεία της περιοχής μπορούν σε τετράγωνες περιοχές (blocks) ή σε κατακόρυφες λωρίδες:



Και στις δύο περιπτώσεις, κάθε διεργασία αναλαμβάνει n^2/p στοιχεία



• Στη block διαμέριση, κάθε διεργασία ανταλλάσει δεδομένα από τέσσερις πλευρές. Αν τα δεδομένα κάθε πλευράς τοποθετούνται σε ένα μήνυμα, κάθε διεργασία στέλνει/λαμβάνει 4 συνολικά μηνύματα από τις τέσσερις γειτονικές διεργασίας. Το πλήθος των στοιχείων κάθε πλευράς είναι $n/\sqrt{p}$. Συνολικά, ο χρόνος επικοινωνίας σε μία επανάληψη είναι:

$$t_{comm} = 4(t_{startup} + \frac{n}{\sqrt{p}} t_{data})$$

• Στη διαμέριση σε λωρίδες, κάθε διεργασία ανταλλάσει δεδομένα και από δύο πλευρές. Το πλήθος των στοιχείων κάθε πλευράς είναι n και επομένως ο συνολικός χρόνος επικοινωνίας θα είναι:

$$t_{comm} = 2(t_{startup} + n\, t_{data})$$

• Γενικά, η διαμέριση σε λωρίδες προτιμάται όταν ο χρόνος έναρξης $t_{startup}$ είναι σχετικά μεγάλος ενώ η block διαμέριση είναι προτιμότερη στην αντίθετη περίπτωση. Με βάση τις προηγούμενες εξισώσεις, η block διαμέριση έχει μεγαλύτερο χρόνο επικοινωνίας από την διαμέριση σε λωρίδες αν

$$t_{startup} > n\left(1 - \frac{2}{\sqrt{p}}\right) t_{data}$$

Ασφάλεια και Αδιέξοδο

• Στο κώδικα που έχουμε περιγράψει, όλες οι διεργασίες εκτελούν πρώτα όλες τις εντολές send() και στη συνέχεια όλες οι λειτουργίες recv(). Αυτός ο τρόπος επικοινωνίας δεν αποκλείει τη πιθανότητα αδιεξόδου. Αυτό οφείλεται στο ότι η ολοκλήρωση της λειτουργίας send() εξαρτάται από το διαθέσιμο αποθηκευτικό χώρο που υπάρχει κατά τη στιγμή της εκτέλεσης της send(). Αν δεν υπάρχει επαρκής διαθέσιμος χώρος, η λειτουργία send() καθυστερεί μέχρι ο διαθέσιμος χώρος να αυξηθεί ή μέχρι το μήνυμα να μπορεί να σταλεί κατευθείαν χωρίς να χρειάζεται εκ των προτέρων αποθήκευση.
• Σε αυτή τη περίπτωση η λειτουργία send() λειτουργεί ως συγχρονισμένη λειτουργία και επιστρέφει μόνο όταν το αντίστοιχο recv() εκτελείται. Αφού, όμως το αντίστοιχο recv() δεν εκτελείται ποτέ, συμβαίνει αδιέξοδο.

• Ένας τρόπος για να αποφύγουμε το παραπάνω πρόβλημα είναι να εναλλάξουμε τη σειρά των sends και recvs σε γειτονικές διεργασίες έτσι ώστε μόνο μία διεργασία να εκτελεί πρώτα τις εντολές send. Σε αυτή τη περίπτωση οι λειτουργίες send() δεν μπορούν να προκαλέσουν αδιέξοδο.
• Εναλλακτικά το MPI προσφέρει παραλλαγές της βασικής send και recv με τις οποίες αποφεύγουμε την εμφάνιση αδιεξόδου:

- Η εντολή MPI_Sendrecv() η οποία συνδυάζει την αποστολή και τη λήψη και εγγυάται την απουσία αδιεξόδου
- Η εντολή MPI_Bsend() όπου ο ίδιος ο χρήστης παρέχει αποθηκευτικό χώρο για την προσωρινή αποθήκευση του προς αποστολή μηνύματος
- Οι μη αναστέλλουσες εντολες: MPI_Isend() και MPI_Irecv() οι οποίες επιστρέφουν κατευθείαν.
- Ξεχωριστές εντολές χρησιμοποιούνται για να διαπιστωθεί αν η λειτουργία που άρχισε με την MPI_Isend() και την MPI_Irecv() έχει ολοκληρωθεί:
  - MPI_Wait, MPI_Waitall(), MPI_Waitany(): οι εντολές αυτές περιμένουν μέχρι να ολοκληρωθεί η λειτουργία/ες send ή recv που είναι σε εξέλιξη
  - MPI_Test(), MPI_Testall(), MPI_Testany(): οι εντολές αυτές απλά ελέγχουν αν έχει ολοκληρωθεί η λειτουργία send ή recv και επιστρέφουν ανάλογα TRUE ή FALSE

Άλλα προβλήματα που απαιτούν συγχρονισμό μεταξύ των διεργασιών - Κυψελιδικά Αυτόματα (Cellular Automata)

• Σε αυτά τα προβλήματα, ο χώρος του προβλήματος διαιρείται σε κυψέλες. Κάθε κυψέλη μπορεί να είναι σε μία κατάσταση από ένα πεπερασμένο πλήθος καταστάσεων.
• Οι κυψέλες επηρεάζονται από τους γείτονές της σύμφωνα με μερικούς κανόνες, και όλες οι κυψέλες επηρεάζονται ταυτόχρονα ως μία «γενιά».
• Οι ίδιοι κανόνες επαναλαμβάνονται στις επόμενες γενιές και έτσι με αυτό τον τρόπο οι κυψέλες εξελίσσονται ή αλλάζουν κατάσταση από γενιά σε γενιά.
• Τα πιο γνωστό πρόβλημα που μπορεί να περιγραφεί με όρους κυψελιδικού αυτομάτου είναι το «Παιχνίδι της ζωής».

Σε αυτό το πρόβλημα υπάρχει μία θεωρητικά άπειρη δισδιάστατη διάταξη από κυψέλες. Κάθε κυψέλη τηρεί ένα «οργανισμό» και έχει οκτώ γειτονικές κυψέλες, συμπεριλαμβανομένων και των διαγώνιων κυψελών. Αρχικά, κάποιες κυψέλες είναι κατειλημμένες.

Οι παρακάτω κανόνες εφαρμόζονται:

1. Κάθε οργανισμός με δύο ή τρεις γειτονικούς οργανισμούς επιζεί και στην επόμενη γενιά.
2. Κάθε οργανισμός με τέσσερις ή περισσότερους γειτονικούς οργανισμούς πεθαίνει λόγω υπερπληθυσμού.
3. Κάθε οργανισμός με ένα ή κανένα γειτονικό οργανισμό πεθαίνει από απομόνωση.
4. Σε κάθε άδεια κυψέλη γειτονική με ακριβώς τρείς οργανισμούς δημιουργείται ένας νέος οργανισμός.

Τα κυψελιδικά αυτόματα έχουν εφαρμογές σε πολλά προβλήματα:

• μηχανική υγρών και αερίων

• η κίνηση υγρών και αερίων γύρω από αντικείμενα

• διάχυση αερίων

• προσομοίωση βιολογικών διαδικασιών

• προβλήματα αεροδυναμικής, π.χ. η ροή του αέρα σε ένα φτερό αεροπλάνου

• η διάβρωση/κίνηση του εδάφους σε μία παραλία ή σε μία όχθη ποταμού.

Μερικώς συγχρονισμένοι υπολογισμοί

• Στους υπολογισμούς αυτούς, κάθε διεργασία εκτελείται χωρίς να χρειάζεται να συγχρονίζεται με άλλες διεργασίες μετά από κάθε επανάληψη.
• Ο μερικός συγχρονισμός επιταχύνει σημαντικά την εκτέλεση του παράλληλου προγράμματος αφού σε αυτή την περίπτωση, οι διεργασίες συγχρονίζονται σε πιο αραιά διαστήματα και δεδομένου ότι ο συγχρονισμός γενικά είναι μία χρονοβόρα λειτουργία η οποία επιβραδύνει σημαντικά τον υπολογισμό.
• Ο καθολικός συγχρονισμός πραγματοποιείται με την εκτέλεση λειτουργιών φράγματος οι οποίες μερικές φορές έχουν ως αποτέλεσμα την αναίτια καθυστέρηση των διεργασιών.
• Το πρόβλημα της κατανομής θερμότητας στο επίπεδο μπορεί επίσης να επιλυθεί με μία τεχνική μερικού συγχρονισμού.
• Όπως έχουμε αναφέρει, για να λύσουμε το πρόβλημα κατανομής θερμότητας, ο χώρος του προβλήματος διαιρείται σε μία δισδιάστατη διάταξη από σημεία. Η τιμή κάθε σημείου υπολογίζεται από το μέσο όρο των τιμών των τεσσάρων γειτονικών σημείων και η διαδικασία αυτή επαναλαμβάνεται μέχρι οι τιμές να συγκλίνουν σε μία λύση με ικανοποιητική ακρίβεια.
• Ο υπολογισμός μπορεί να επιταχυνθεί σημαντικά αν οι διεργασίες δεν συγχρονίζονται μετά από κάθε επανάληψη.

Sequential code

```
do {
 for (i = 1; i < n; i++)
   for (j = 1; j < n; j++)
     g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);

  for (i = 1; i < n; i++)        /* find max divergence/update pts */
    for (j = 1; j < n; j++) {
     dif = h[i][j] - g[i][j];
     if (dif < 0) dif = -dif;
     if (dif < max_dif) max_dif = dif;
     h[i][j] = g[i][j];
    }
} while (max_dif > tolerance);    /* test convergence */
```

• Αν οι διεργασίες δεν συγχρονίζονται μετά από κάθε επανάληψη, μερικές διεργασίες μπορούν να προχωρήσουν στην επόμενη επανάληψη πριν όλες οι διεργασίες να έχουν τελειώσει την τρέχουσα επανάληψη.
• Έτσι, αυτές οι διεργασίες που προχωρούν στην επόμενη επανάληψη χρησιμοποιούν τιμές που έχουν υπολογιστεί όχι μόνο στη τελευταία επανάληψη αλλά προέρχονται και από παλαιότερες επαναλήψεις.
• Μετά από αυτές τις αλλαγές, η μέθοδος γίνεται μία ασύγχρονη επαναληπτική μέθοδος.
• Στις ασύγχρονες επαναληπτικές μεθόδους, η σύγκλιση σε μία λύση είναι πιο δύσκολη να επιτευχθεί και απαιτούνται αυστηρότερες συνθήκες για να επίτευξή της.
• Προκειμένου να είναι εφικτή η σύγκλιση, δεν επιτρέπεται γενικά, μία διεργασία να χρησιμοποιεί τιμές από αρκετά παλαιές επαναλήψεις. Συγκεκριμένα έχει αποδειχθεί ότι:
• Υπάρχει πάντα μία θετική ακέραια σταθερά *s* τέτοια ώστε, αν κατά τους υπολογισμούς στην i-οστη επανάληψη, κάθε διεργασία δεν χρησιμοποιήσει τιμές που προέκυψαν στην j-οστη επανάληψη όπου $j < i - s$, η σύγκλιση της μεθόδου είναι εφικτή (Baudet, 1978).
• Με βάση την παραπάνω αρχή, επιτρέπεται κάθε διεργασία να συγχρονίζεται με τις γειτονικές διεργασίες κάθε *s* επαναλήψεις. Στις ενδιάμεσες επαναλήψεις, κάθε διεργασία χρησιμοποιεί για κάθε γειτονικό σημείο όποια τιμή έχει εκείνη τη στιγμή στη διάθεσή της. Έτσι οι τιμές των γειτονικών σημείων που χρησιμοποιούνται για την ενημέρωση τιμής ενός σημείου μπορεί να προέρχονται από διαφορετικές επαναλήψεις. Η πιο παλαιά από αυτές μπορεί να απέχει το πολύ s επαναλήψεις από την τρέχουσα
• Επίσης κάθε s επαναλήψεις ελέγχεται αν η λύση έχει αποκτήσει την επιθυμητή ακρίβεια δηλ. έχει συγκλίνει.
• Αυτή η ασύγχρονη επαναληπτική διαδικασία είναι γνωστή επίσης και με τον όρο χαοτική χαλάρωση (chaotic relaxation)