MSc – CNT

ΠΑΡΑΛΛΗΛΑ και ΚΑΤΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

Lecture #9

ΑΛΓΟΡΙΘΜΟΙ ΚΑΤΑΝΕΜΗΜΕΝΗΣ ΜΝΗΜΗΣ

(Επίλυση Γραμμικών Συστημάτων)

Επίλυση Γραμμικών Συστημάτων

Αναφερόμενοι σε ένα σύστημα *n* γραμμικών εξισώσεων με *n* αγνώστους, υπονοείται ένα σύνολο *n* εξισώσεων της μορφής που ακολουθεί, το οποίο μπορεί να γραφτεί και ως $Ax = B$, όπου ο πίνακας *A* (διαστάσεων *nxn*) είναι ο πίνακας των συντελεστών (a_{ij} | *i,j=0…n-1*), το διάνυσμα *x* (διαστάσεων *nx1*) είναι το διάνυσμα των αγνώστων (x_i | *i=0…n-1*), και το διάνυσμα *b* (διαστάσεων επίσης *nx1*) είναι το διάνυσμα των σταθερών (b_i | *i=0…n-1*).

$$
\begin{array}{ccccccccc}
a_{0,0}\,x_0 & + & a_{0,1}\,x_1 & + & \dots & + & a_{0,n-1}\,x_{n-1} & = & b_0 \\
a_{1,0}\,x_0 & + & a_{1,1}\,x_1 & + & \dots & + & a_{1,n-1}\,x_{n-1} & = & b_1 \\
\dots & + & \dots & + & \dots & + & \dots & = & \dots \\
a_{n-1,0}\,x_0 & + & a_{n-1,1}\,x_1 & + & \dots & + & a_{n-1,n-1}\,x_{n-1} & = & b_{n-1}
\end{array}
$$

$$
A \quad . \quad x \quad = \quad b
$$

$$
\begin{bmatrix}
a_{0,0} & a_{0,1} & \dots & a_{0,n-1} \\
a_{1,0} & a_{1,1} & \dots & a_{1,n-1} \\
\dots & \dots & \dots & \dots \\
a_{n-1,0} & \dots & \dots & a_{n-1,n-1}
\end{bmatrix}
\cdot
\begin{bmatrix}
x_0 \\ x_1 \\ \dots \\ x_{n-1}
\end{bmatrix}
=
\begin{bmatrix}
b_0 \\ b_1 \\ \dots \\ b_{n-1}
\end{bmatrix}
$$

Σχήμα 5.22 Γραμμικό σύστημα *n* εξισώσεων με *n* αγνώστους

Επίλυση Γραμμικών Συστημάτων

(Παράλληλοι Αλγόριθμοι σε περιβάλλον κατανεμημένης μνήμης)

- Η Μέθοδος Απαλοιφής του Gauss
- Η Μέθοδος Jacobi (barrier)
- Η Μέθοδος της Ανάδρομης Αντικατάστασης (pipeline)

Η Μέθοδος Απαλοιφής του Gauss

```
for i:=0 to n-2 do
   for k:=i+1 to n-1 do
      begin
        c := a[k,i] / a[i,i];
        for j:=i to n-1 do
              a[k,j] := α[k,j] - α[i,j] * c;
        b[k] := b[k] - b[i] * c;
      end
```

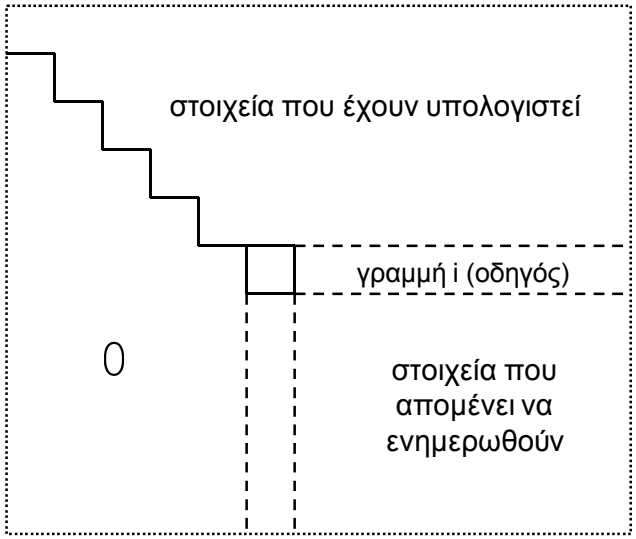
Η Σειριακή Μορφή του Αλγορίθμου

Μετά την εκτέλεση του παραπάνω αλγορίθμου ο πίνακας *A* θα έχει έρθει σε άνω τριγωνική μορφή, όπως φαίνεται ενδεικτικά παρακάτω:

$$
\begin{array}{ccccccccccc}
a'_{0,0}\,x_0 & + & a'_{0,1}\,x_1 & + & a'_{0,2}\,x_2 & + & \dots & + & a'_{0,n-1}\,x_{n-1} & = & b'_0 \\
 & & a'_{1,1}\,x_1 & + & a'_{1,2}\,x_2 & + & \dots & + & a'_{1,n-1}\,x_{n-1} & = & b'_1 \\
 & & & & a'_{2,2}\,x_2 & + & \dots & + & a'_{2,n-1}\,x_{n-1} & = & b'_2 \\
 & & & & & & \dots & + & \dots & = & \dots \\
 & & & & & & & & a'_{n-1,n-1}\,x_{n-1} & = & b'_{n-1}
\end{array}
$$

Η Μέθοδος Απαλοιφής του Gauss

Στη συνέχεια αρκεί η εφαρμογή της μεθόδου της *ανάδρομης αντικατάστασης* για την εύρεση της λύσης του συστήματος. Είναι εύκολο να διαπιστωθεί ότι ο ανωτέρω σειριακός αλγόριθμος (εστιάζοντας στον αριθμό επαναλήψεων που υπαγορεύεται από τους τρεις βρόγχους που εκτελούνται ένθετα ο ένας μέσα στον άλλο) έχει χρόνο πολυπλοκότητας $O(n^3)$



Σχήμα 5.23 Απεικόνιση του βήματος '*i*' της μεθόδου απαλοιφής του Gauss

Η Μέθοδος Απαλοιφής του Gauss

- Ο παραπάνω αλγόριθμος της μεθόδου απαλοιφής του Gauss δεν είναι εύκολο να παραλληλοποιηθεί αποδοτικά. Η δυσκολία οφείλεται στο ότι ο υπολογισμός ενός στοιχείου στην άνω τριγωνική περιοχή απαιτεί να είναι γνωστά όλα τα προηγούμενα (όλες οι προηγούμενες γραμμές).
- Με άλλα λόγια τα βήματα/επαναλήψεις του αλγορίθμου που υπαγορεύονται από το εξωτερικό for loop δεν μπορούν να εκτελεστούν παράλληλα. Για την εκτέλεση του βήματος τάξης *i* πρέπει πρώτα να έχει ολοκληρωθεί το βήμα τάξης *i-1*. Είναι ωστόσο δυνατό να γίνει παραλληλοποίηση των υπολογισμών που ακολουθούν εσωτερικά σε κάθε βήμα.
- Με βάση τις παραπάνω διαπιστώσεις, το τμήμα του αλγορίθμου που μπορεί να παραλληλοποιηθεί είναι αυτό της ενημέρωσης όλων των επόμενων γραμμών όταν υπολογιστούν τα στοιχεία κάποιας γραμμής, δηλαδή οι πράξεις που περιλαμβάνονται μέσα στον ενδιάμεσο βρόγχο του σειριακού αλγόριθμου που δόθηκε παραπάνω:

Gauss -: Βήματα Παραλληλοποίησης

1. Κάθε επανάληψη που υπαγορεύεται από αυτόν τον βρόγχο μπορεί να εκτελεσθεί από διαφορετικό επεξεργαστή. Ένας εμφανής τρόπος παραλληλοποίησης είναι να χρησιμοποιήσουμε $p=n$ επεξεργαστές και να αναθέσουμε μία γραμμή του A σε κάθε επεξεργαστή (π.χ. τη γραμμή i στον επεξεργαστή P_i), στα στοιχεία της οποίας θα επιτελεί ανεξάρτητα την ανωτέρω λειτουργία ενημέρωσης του σειριακού αλγορίθμου, σε κάθε βήμα που παραμένει ενεργός.

2. Το ίδιο μπορούμε να κάνουμε και σε σχέση με την ενημέρωση του διανύσματος b. Δηλαδή να αναθέσουμε και ένα στοιχείο του b σε κάθε επεξεργαστή (το στοιχείο i στον επεξεργαστή P_i), στο οποίο θα επιτελεί την αντίστοιχη λειτουργία ενημέρωσης.

3. Πριν επίσης κάθε επεξεργαστής (αν παραμένει ενεργός) επιτελέσει υπολογισμούς όπως περιγράφεται παραπάνω σε κάθε βήμα i, πρέπει ο επεξεργαστής P_i να στείλει σε όλους τους επόμενους επεξεργαστές (P_{i+1} έως P_{n-1}) τα απαραίτητα στοιχεία της γραμμής του (γραμμή οδηγός), δηλαδή τα στοιχεία $a_i, a_{i+1}, \ldots, a_{n-1}$ της γραμμής i, καθώς και το στοιχείο του διανύσματος b που κατέχει.

Gauss -: Χρόνος Ολοκλήρωσης

- Τα παραπάνω, υπενθυμίζεται ότι αφορούν στη διαδικασία μετατροπής του αρχικού γραμμικού συστήματος σε τριγωνική μορφή. Στη συνέχεια, είναι απαραίτητο για την τελική επίλυση του συστήματος να εφαρμοστεί η μέθοδο της ανάδρομης αντικατάστασης.
- Τα συμπεράσματα ωστόσο σχετικά με τον παράλληλο χρόνο ολοκλήρωσης που διατυπώνονται παρακάτω δεν επηρεάζονται, καθώς ο παράλληλος χρόνος που απαιτείται για την επίλυση του τριγωνικού συστήματος με την ανωτέρω μέθοδο είναι υποδεέστερος (*O(n)*).

Computation Time: $T_{comp} = O(n^2)$

Communication Time: $T_{comm} = O(n^2)$

Ο χρόνος επικοινωνίας με βάση το *πρότυπο του δικτύου* είναι ίσος με:

$$T'_{comm} = (n-1)\, t_0 + r\,(((n+1)(n+2)/2)-3)$$

Gauss -: Επέκταση για n>p

- Ο τρόπος παραλληλοποίησης που παρουσιάστηκε παραπάνω μπορεί εύκολα να επεκταθεί και για τη γενική περίπτωση που το πλήθος διαθέσιμων επεξεργαστών είναι μικρότερο από τον αριθμό των εξισώσεων και αγνώστων του γραμμικού συστήματος ($n>p$).
- Για την υλοποίηση της παραλληλοποίησης σε αυτό το επίπεδο, αρκεί η εφαρμογή της τεχνικής της *κατάτμησης δεδομένων*, και πιο συγκεκριμένα η κατάλληλη διαμοίραση του πίνακα *A* κατά γραμμές σε όλους του επεξεργαστές (n/p γραμμές σε κάθε επεξεργαστή, αν υποθέσουμε χωρίς βλάβη της γενικότητας ότι το n είναι ακέραιο πολλαπλάσιο του p).
- Παρόμοια απαιτείται επίσης η διαμοίραση κατά τον ίδιο τρόπο και του διανύσματος *b* (n/p στοιχεία σε κάθε επεξεργαστή).

Διαμοίραση κατά Γραμμές
(1η εναλλακτική)

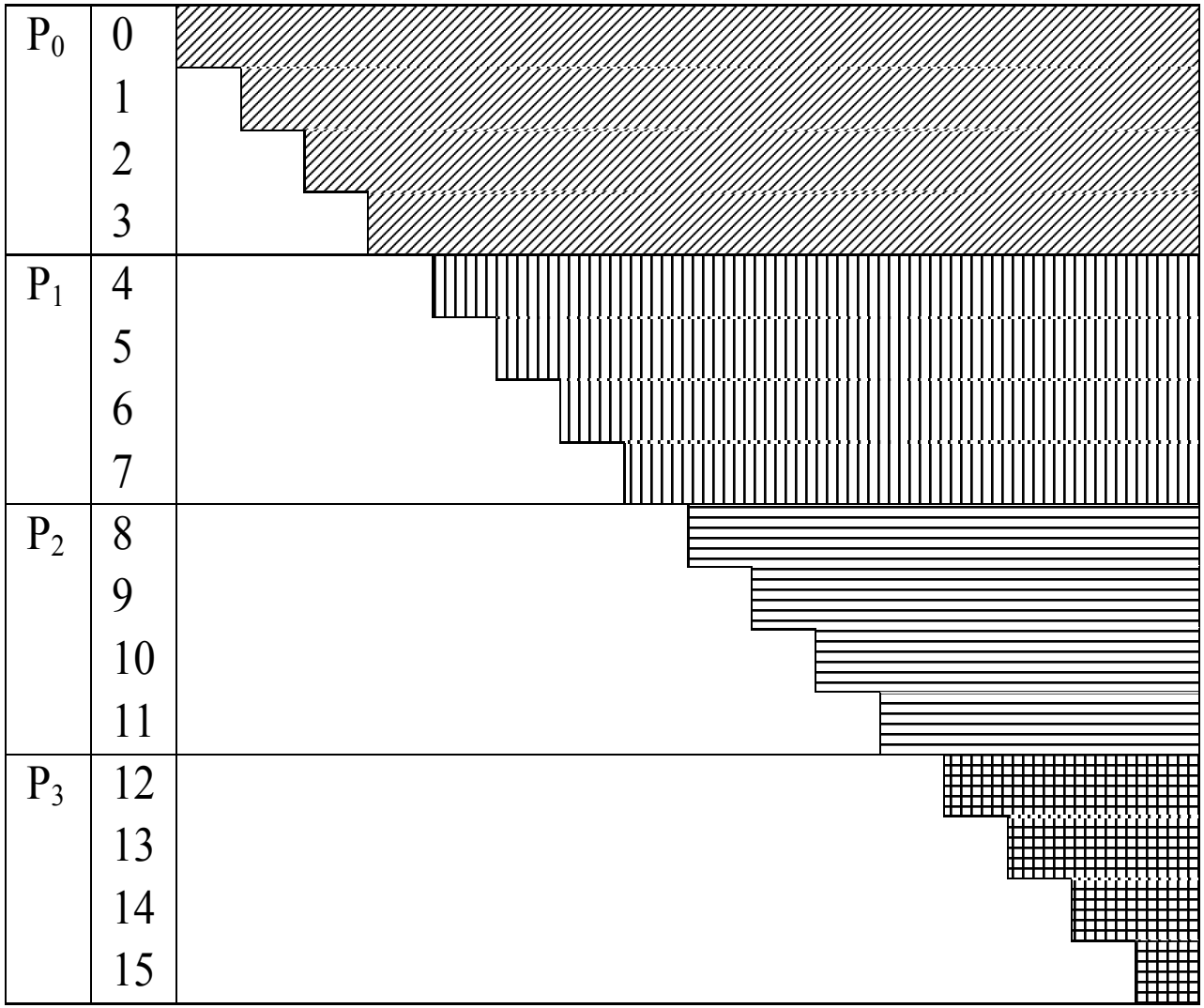
Τμηματική κατανομή:

- Η τμηματική κατανομή είναι ο πιο απλός τρόπος κατανομής των γραμμών του πίνακα στους επεξεργαστές.
- Γίνεται ανάθεση ενός υποσυνόλου συνεχόμενων γραμμών του πίνακα συντελεστών του συστήματος σε κάθε επεξεργαστή.
- Επομένως κάθε επεξεργαστής P_i αναλαμβάνει n/p συνεχόμενες γραμμές, και πιο συγκεκριμένα τις γραμμές τάξης $i(n/p)$ έως $(i+1)(n/p)-1$.

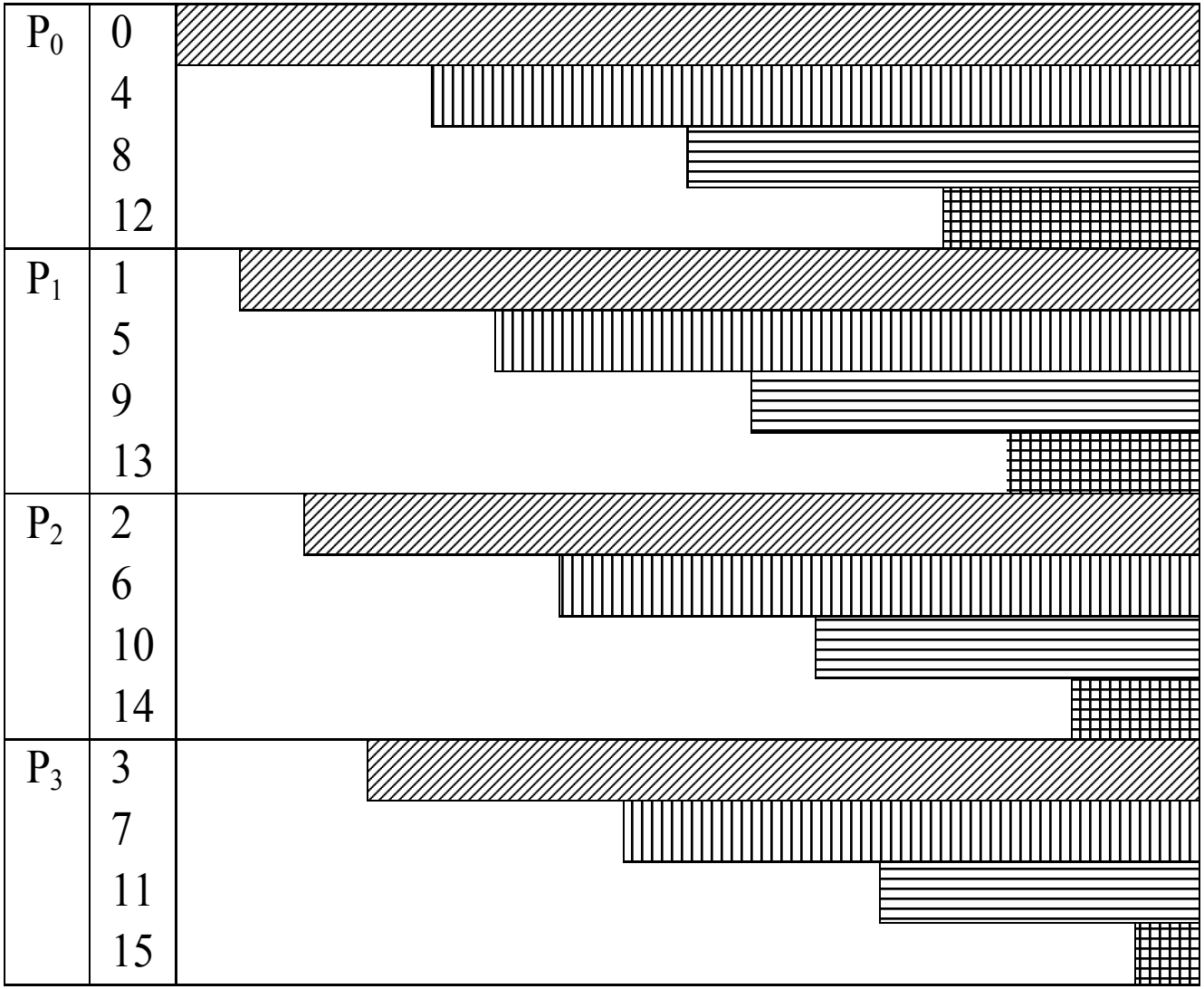
Διαμοίραση κατά Γραμμές
(2η εναλλακτική)

Κυκλική κατανομή:

- Στην κυκλική κατανομή η διαφορά σε σχέση με την τμηματική κατανομή είναι ότι οι γραμμές του πίνακα που αναλαμβάνει κάθε επεξεργαστής δεν είναι συνεχόμενες.
- Κάθε επεξεργαστής αναλαμβάνει γραμμές που απέχουν κατά p μεταξύ τους. Ο επεξεργαστής Pi αναλαμβάνει τις γραμμές τάξης i, i+p, i+2p, κοκ.,
- Δηλαδή η ανάθεση των γραμμών γίνεται κυκλικά (κάθε γραμμή τάξης i κατανέμεται στον επεξεργαστή i mod p).



Σχήμα 5.24 Τμηματική κατανομή γραμμών πίνακα



Σχήμα 5.25: Κυκλική κατανομή γραμμών πίνακα

Η Μέθοδος Jacobi

- Η μέθοδος Jacobi αποτελεί μία από τις απλούστερες επαναληπτικές μεθόδους επίλυσης γραμμικών συστημάτων.
- Είναι κατάλληλη κατά κύριο λόγο για αραιά (sparse) συστήματα (γραμμικά συστήματα με αραιό πίνακα συντελεστών) και παραλληλοποιείται εύκολα λόγω των ανεξάρτητων υπολογισμών στους οποίους μπορεί να διαιρεθεί σε κάθε επανάληψή της.
- Πιο συγκεκριμένα, υποθέτοντας ότι μας δίνεται ένα σύστημα $Ax=b$ όπως ορίστηκε στην αρχή της παρούσας ενότητας, και επιπλέον ότι ο πίνακας A είναι αντιστρέψιμος, έτσι ώστε το ανωτέρω σύστημα να έχει μοναδική λύση, η μέθοδος Jacobi
  - ✓ δέχεται σαν είσοδο ένα διάνυσμα αρχικών τιμών $x(0)$, και
  - ✓ υπολογίζει επαναληπτικά τα διανύσματα $x(t)$, t=1,2,…, χρησιμοποιώντας τον ακόλουθο τύπο:

Η Μέθοδος Jacobi

$$x_i(t+1) = -\frac{1}{a_{ii}}\left[\sum_{j\neq i} a_{ij}x_j(t) - b_i\right]$$

- ✓ όπου με $x_i(t+1)$, $x_j(t)$ συμβολίζονται το *i*-στο και *j*-στο στοιχείο των διανυσμάτων $x(t+1)$ και $x(t)$ αντίστοιχα.
- ✓ Με απλά λόγια σε κάθε επανάληψη υπολογίζονται νέες τιμές για όλους τους αγνώστους με βάση τις τιμές που είχαν με το πέρας της προηγούμενης επανάληψης.
- ✓ Για την πρώτη επανάληψη χρησιμοποιούνται ως προηγούμενες τιμές αυτές του διανύσματος αρχικών τιμών $x(0)$.

Η Μέθοδος Jacobi

Μπορεί να αποδειχθεί ότι αν τα διανύσματα *x(t)* συγκλίνουν μετά από μερικές επαναλήψεις σε ένα διάνυσμα *x*, τότε το *x* αποτελεί λύση του γραμμικού συστήματος *Ax=b.* Η σύγκλιση της επαναληπτικής διαδικασίας εξαρτάται κατά κύριο λόγο από την επιλογή του αρχικού διανύσματος *x(0)*. Ικανή συνθήκη για να υπάρχει σύγκλιση αποτελεί ο πίνακας *A* να είναι *αυστηρά διαγώνια δεσπόζων* (για το λόγο αυτό η συγκεκριμένη μέθοδος είναι κατάλληλη για αραιά συστήματα).

Είναι δυνατό ωστόσο να υπάρξει σύγκλιση ακόμα και αν δεν ισχύει η ανωτέρω ιδιότητα. Η σύγκλιση αυτή μπορεί να διαπιστωθεί υπολογίζοντας την *νόρμα της διαφοράς* μεταξύ των διαδοχικών διανυσμάτων *x(t+1)* και *x(t)*, δηλ. την ποσότητα

$$\left\| x(t+1) - x(t) \right\| = \sqrt{\sum_{i=1}^{n} \left(x_i(t+1) - x_i(t) \right)^2}$$

και ελέγχοντας αν είναι μικρότερη από ένα συγκεκριμένο όριο *ξ*.

Jacobi -: Σειριακή Μορφή

Ένας ενδεικτικός σειριακός αλγόριθμος ο οποίος υλοποιεί την ανωτέρω διαδικασία, θεωρώντας ότι ο έλεγχος για τη σύγκλιση ή όχι σε κάθε επανάληψη (με βάση το όριο *ξ*) πραγματοποιείται από μία συνάρτηση *tolerance()*, και χωρίς να μας απασχολεί η αρχικοποίηση του διανύσματος *x(0)* (συνήθως αρχικοποιείται με τις τιμές του *b*), ακολουθεί:

```
for k:=0 to λ-1 do
begin
    for i:=0 to n-1 do
        begin
            sum = -a[i,i] * x[i];
            for j:=0 to n-1 do
                sum = sum + a[i,j] * x[j];
            new_x[i] = (b[i] - sum) / a[i,i];
        end
    for i:=0 to n-1 do
        x[i] := new_x[i];
    if tolerance() then
        break;
end
```

Jacobi -: Παραλληλοποίηση

- Σε κάθε επεξεργαστή P_i αναθέτουμε την επανάληψη τάξης *i* του ανωτέρω εσωτερικού βρόγχου, έτσι ώστε στα πλαίσια κάθε μίας από τις *λ* εξωτερικές επαναλήψεις να υπολογίζει την νέα τιμή του αγνώστου x_i. Για τον υπολογισμό αυτό, ο επεξεργαστής P_i θα πρέπει να κατέχει τοπικά:
  - μία γραμμή του πίνακα *A* (τη γραμμή τάξης *i*),
  - ένα στοιχείο του διανύσματος *b* (το στοιχείο τάξης *i*), και
  - το διάνυσμα των αρχικών τιμών των *n* αγνώστων
- Με το πέρας κάθε εξωτερικής επανάληψης, κάθε επεξεργαστής Pi στέλνει σε όλους τους υπόλοιπους επεξεργαστές (broadcast operation) τη νέα τιμή που υπολόγισε για τον άγνωστο για τον οποίον είναι υπεύθυνος (xi).
- Έτσι, στο τέλος του βήματος αυτού όλοι οι επεξεργαστές θα έχουν στην κατοχή τους όλες τις νέες τιμές για όλους τους αγνώστους, ώστε με βάση αυτές να είναι ικανοί να ξεκινήσουν ορθά την επόμενη επανάληψή τους.

Jacobi -: Παραλληλοποίηση

Οι ανωτέρω επαναλήψεις πρέπει να γίνονται *συγχρονισμένα*, δηλαδή κάθε επεξεργαστής πριν ξεκινήσει την επόμενη επανάληψή του, θα πρέπει να περιμένει να ολοκληρώσουν την τρέχουσα επανάληψη όλοι οι επεξεργαστές. Η απαίτηση αυτή είναι γνωστή ως *συγχρονισμός φράγματος (barrier synchronization).* Η παραπάνω διαδικασία παραλληλοποίησης μπορεί να αποδοθεί συνοπτικά σε μορφή ψευδοκώδικα όπως φαίνεται παρακάτω, για κάθε επεξεργαστή P_i:

```
for k:=0 to λ-1 do
begin
    sum = -a[i,i] * x[i];
    for j:=0 to n-1 do
        sum = sum + a[i,j] * x[j];
    new_x[i] = (b[i] - sum) / a[i,i];
    broadcast (new_x[i]);
    < αναμονή ολοκλήρωσης της επανάληψης σε όλους τους επ/στές >
    if tolerance() then
        break;
end
```

Jacobi -: Χρόνος Ολοκλήρωσης

$$\textit{Computation Time:} \quad T_{comp} = O(n)$$
$$\textit{Communication Time:} \quad T_{comm} = O(n)$$

$$T'_{comm} = n\,(t_0 + r)\,\lambda$$

Ο τρόπος παραλληλοποίησης που παρουσιάστηκε παραπάνω μπορεί εύκολα να επεκταθεί και για τη γενική περίπτωση που → $n>p$. Αρκεί και εδώ η κατάλληλη διαμοίραση του πίνακα *A* κατά γραμμές σε όλους του επεξεργαστές (n/p γραμμές σε κάθε επεξεργαστή). Παρόμοια απαιτείται επίσης η διαμοίραση κατά τον ίδιο τρόπο και του διανύσματος *b* (n/p στοιχεία σε κάθε επεξεργαστή).

$$\textit{Computation Time:} \quad T_{comp} = O(n^2/p)$$
$$\textit{Communication Time:} \quad T_{comm} = O(n)$$

$$T'_{comm} = (p\,t_0 + n\,r)\,\lambda$$

Όπως βλέπουμε με βάση τους παραπάνω χρόνους, ο χρόνος υπολογισμών σε κάθε επεξεργαστή μεγαλώνει, ενώ ο χρόνος επικοινωνίας παραμένει ο ίδιος. Έτσι, ο *λόγος χρόνου υπολογισμών-προς-χρόνο επικοινωνίας* γίνεται πλέον μεγαλύτερος του ένα, προάγοντας κατ' αυτόν τον τρόπο την απόδοση του αλγορίθμου.

Η Μέθοδος της Ανάδρομης Αντικατάστασης

Η μέθοδος της ανάδρομης αντικατάστασης αποτελεί μία από τις γνωστότερες και απλούστερες μεθόδους επίλυσης γραμμικών συστημάτων είτε αυτόνομα (όταν το προς επίλυση σύστημα βρίσκεται σε τριγωνική μορφή) είτε ως μέρος πιο σύνθετων μεθόδων (όπως αυτής της απαλοιφής του Gauss).

Πιο συγκεκριμένα, ας υποθέτουμε ότι το αρχικό (προς επίλυση) σύστημα γραμμικών εξισώσεων $Ax=B$ βρίσκεται σε τριγωνική μορφή, δηλαδή οι n εξισώσεις του έχουν την ακόλουθη μορφή:

$$
\begin{aligned}
a_{0,0}\,x_0 &= b_0\\
a_{1,0}\,x_0 + a_{1,1}\,x_1 &= b_1\\
a_{2,0}\,x_0 + a_{2,1}\,x_1 + a_{2,2}\,x_2 &= b_2\\
\ldots + \ldots + \ldots + \ldots &= \ldots\\
a_{n-1,0}\,x_0 + a_{n-1,1}\,x_1 + a_{n-1,2}\,x_2 + \ldots + a_{n-1,n-1}\,x_{n-1} &= b_{n-1}
\end{aligned}
$$

Ανάδρομη Αντικατάσταση (σειριακά)

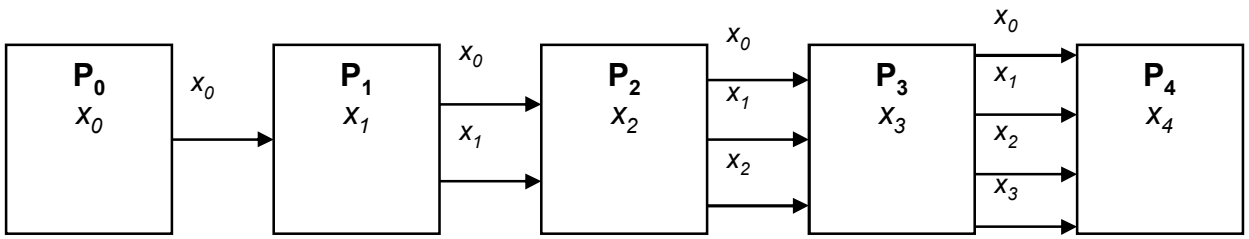
Η επίλυση ενός τέτοιου γραμμικού συστήματος με τη γνωστή σε όλους μας μέθοδο της ανάδρομης αντικατάστασης, υπαγορεύει τον σταδιακό υπολογισμό των αγνώστων, ξεκινώντας από τον x_0 ο οποίος μπορεί να υπολογιστεί άμεσα μέσω της τελευταίας εξίσωσης, και συνεχίζοντας με τους υπόλοιπους, αντικαθιστώντας επαναληπτικά την τιμή του κάθε νέου αγνώστου που υπολογίζεται στις υπόλοιπες εξισώσεις. Ένας ενδεικτικός σειριακός αλγόριθμος ο οποίος υλοποιεί την ανωτέρω διαδικασία, μπορεί να διατυπωθεί ως ακολούθως:

```
x[0] := b[0]/a[0,0];
for i:=1 to n-1 do
    begin
        sum := 0;
        for j:=0 to i-1 do
            sum = sum + a[i,j]*x[j];
        x[i] = (b[i] - sum)/a[i,i];
    end
```

Ανάδρομη Αντικατάσταση (pipeline)

Παρατηρώντας επίσης προσεκτικά τα επιμέρους βήματα του αλγορίθμου είναι εμφανές ότι για τον ορθό υπολογισμό του κάθε αγνώστου x_i (ως έργο της κάθε επανάληψης τάξης i του εξωτερικού βρόγχου) απαιτείται να έχουν υπολογιστεί όλοι οι άγνωστοι $x_j \mid j<i$, δηλαδή να έχουν ολοκληρωθεί όλες οι προηγούμενες επαναλήψεις του εξωτερικού βρόγχου.

Με βάση την παραπάνω διαπίστωση, είναι άμεσα εφικτό να παραλληλοποιηθεί το σύνολο των υπολογισμών του αλγορίθμου με την **τεχνική της σωλήνωσης**, χρησιμοποιώντας $p=n$ επεξεργαστές και αναθέτοντας τον υπολογισμό ενός αγνώστου σε κάθε επεξεργαστή, ως ακολούθως:



Σχήμα 5.29 Η μέθοδος της ανάδρομης αντικατάστασης για $p=5$

Ανάδρομη Αντικατάσταση (pipeline)

- Ο υπολογισμός του πρώτου αγνώστου (x_0), της πρώτης γραμμής δηλαδή του σειριακού αλγορίθμου, ανατίθεται στον επεξεργαστή P_0. Στη συνέχεια, κάθε επανάληψη τάξης i ($i>0$) που υπαγορεύεται από τον εξωτερικό βρόγχο ανατίθεται σε κάθε επεξεργαστή P_i.
- Κάθε επεξεργαστής P_i, όταν ολοκληρώσει τους υπολογισμούς του για τον άγνωστο για τον οποίο είναι υπεύθυνος, στέλνει την τιμή αγνώστου που υπολόγισε (x_i) στον επόμενο επεξεργαστή (P_{i+1}). Η αρχή θα γίνει με τον P_0 ο οποίος θα στείλει το x_0 στον P_1.
- Κάθε επεξεργαστής P_i, όταν παραλάβει ένα μήνυμα με την τιμή ενός αγνώστου από τον προηγούμενο επεξεργαστή, το προωθεί άμεσα και στον επόμενο (κρατώντας ένα αντίγραφο).
- Κάθε επεξεργαστής P_i αφού παραλάβει (μέσω αντίστοιχων μηνυμάτων από τους προηγούμενους) τις τιμές όλων των αγνώστων που χρειάζεται (i συνολικά), πραγματοποιεί τις πράξεις που απαιτούνται για τον υπολογισμό του αγνώστου x_i, για τον οποίο είναι υπεύθυνος.

Ανάδρομη Αντικατάσταση (pipeline)

Σε μορφή ψευδοκώδικα, μπορεί επίσης να αποδοθεί συνοπτικά όπως φαίνεται παρακάτω (εκτελείται από κάθε επεξεργαστή P_i):

```
for j:= 0 to i-1 do
    begin
     receive (x[j], prev_processor);
     if i<>(p-1) then
              send (x[j], next_processor);
    end
sum = 0;
for j:=0 to i-1 do
      sum := sum + a[i,j]*x[j];
x[i] := (b[i] - sum)/a[i,i];
if i<>(p-1) then
     send (x[i], next_processor);
```

Ανάδρομη Αντικατάσταση (pipeline)

P0	**P1**	**P2**	**P3**	**P4**
Πράξη /				
Send(x_0)	Receive(x_0)			
End	Send(x_0)	Receive(x_0)		
	Πράξεις *,+	Send(x_0)	Receive(x_0)	
	Πράξεις /,-		Send(x_0)	Receive(x_0)
	Send(x_1)	Receive(x_1)		
	End	Send(x_1)	Receive(x_1)	
		Πράξεις *,+	Send(x_1)	Receive(x_1)
		Πράξεις *,+		
		Πράξεις /,-		
		Send(x_2)	Receive(x_2)	
		End	Send(x_2)	Receive(x_2)
			Πράξεις *,+	
			Πράξεις *,+	
			Πράξεις *,+	
			Πράξεις /,-	
			Send(x_3)	Receive(x_3)
			End	Πράξεις *,+
				Πράξεις *,+
				Πράξεις *,+
				Πράξεις *,+
				Πράξεις /,-
				End

Σχήμα 5.30 Ανάδρομη αντικατάσταση χωρίς επικάλυψη βημάτων ($p=5$)

Βελτίωση με Επικάλυψη Βημάτων

Αντίθετα, θα μπορούσαμε να ξεφύγουμε από αυτήν την αυστηρή αλληλουχία παραλαβής μηνυμάτων και εκτέλεσης υπολογισμών σε κάθε επεξεργαστή, διαφοροποιώντας λίγο τον αλγόριθμο, έτσι ώστε κάθε επεξεργαστής κάθε φορά που παραλαμβάνει ένα μήνυμα αγνώστου (και αφού πρώτα το προωθήσει στον επόμενο), να μην περιμένει αμέσως μετά το επόμενο μήνυμα αγνώστου αλλά πρώτα να επιτελεί τοπικά ότι υπολογισμούς είναι δυνατοί με τον άγνωστο x_i που μόλις έλαβε.

```
sum = 0;
for j:=0 to i-1 do
    begin
     receive (x[j], prev_processor);
     if i<>(p-1) then
              send (x[j], next_processor);
     sum := sum + a[i,j]*x[j];
    end
x[i] := (b[i] - sum)/a[i,i];
if i<>(p-1) then
     send (x[i], next_processor);
```

Βελτίωση με Επικάλυψη Βημάτων

P0	**P1**	**P2**	**P3**	**P4**
Πράξη /				
Send(x_0)	Receive(x_0)			
End	Send(x_0)	Receive(x_0)		
	Πράξεις *,+	Send(x_0)	Receive(x_0)	
	Πράξεις /,-	Πράξεις *,+	Send(x_0)	Receive(x_0)
	Send(x_1)	Receive(x_1)	Πράξεις *,+	Πράξεις *,+
	End	Send(x_1)	Receive(x_1)	
		Πράξεις *,+	Send(x_1)	Receive(x_1)
		Πράξεις /,-	Πράξεις *,+	Πράξεις *,+
		Send(x_2)	Receive(x_2)	
		End	Send(x_2)	Receive(x_2)
			Πράξεις *,+	Πράξεις *,+
			Πράξεις /,-	
			Send(x_3)	Receive(x_3)
			End	Πράξεις *,+
				Πράξεις /,-
				End

Σχήμα 5.31 Ανάδρομη αντικατάσταση με επικάλυψη βημάτων ($p=5$)

Χρόνος Ολοκλήρωσης

$$\textit{Computation Time:} \qquad T_{comp} = O(n)$$

$$\textit{Communication Time:} \qquad T_{comm} = O(n)$$

Ο ακριβής χρόνος επικοινωνίας με βάση το *πρότυπο του δικτύου* είναι ίσος με:

$$T'_{comm} = (2n-3)\,(t_0 + r)$$

Με βάση τα παραπάνω, προκύπτει ότι επιτυγχάνεται (θεωρητικά) και εδώ σε επίπεδο τάξης πολυπλοκότητας, η επιθυμητή ελάττωση του συνολικού χρόνου ολοκλήρωσης (από $O(n^2)$ που απαιτείτο σειριακά σε $O(n)$ με χρήση n επεξεργαστών). Η παραπάνω απόδοση μπορεί να θεωρηθεί καταρχήν ικανοποιητική, παρότι ο λόγος χρόνου υπολογισμών-προς-χρόνο επικοινωνίας δεν είναι και εδώ (όπως και στους υπόλοιπους αλγόριθμους επίλυσης γραμμικών συστημάτων που παρουσιάστηκαν, για $n=p$) μεγαλύτερος του ένα.

Χρόνος Ολοκλήρωσης - Επέκταση

- Η υπόθεση την οποία κάναμε ωστόσο σε σχέση με την ιδανική εκτέλεση σε μορφή σωλήνωσης όλων των ενδιάμεσων βημάτων του αλγορίθμου, δεν είναι ιδιαίτερα ρεαλιστική καθώς προϋποθέτει τέλεια επικάλυψη μεταξύ μεμονωμένων βημάτων υπολογισμού και επικοινωνίας. Κατά συνέπεια, στην πράξη δεν αναμένεται τόσο υψηλή επιτάχυνση σε σχέση με τη σειριακή εκτέλεση του αλγορίθμου όσο αυτή που υπαγορεύει η υπολογισθείσα κατ' αυτόν τον τρόπο τάξη πολυπλοκότητας (*O(n)*), αλλά αρκετά μικρότερη.
- Τέλος, ο τρόπος παραλληλοποίησης που παρουσιάστηκε παραπάνω μπορεί εύκολα να επεκταθεί και για τη γενική περίπτωση που το πλήθος διαθέσιμων επεξεργαστών είναι μικρότερο από τον αριθμό των εξισώσεων και αγνώστων του γραμμικού συστήματος (*n>p*), αναθέτοντας περισσότερες από μία εξισώσεις (*n/p* αν υποθέσουμε χωρίς βλάβη της γενικότητας ότι *n%p=0*) του συστήματος σε κάθε επεξεργαστή (αναθέτοντας δηλαδή π.χ. τον υπολογισμό των αγνώστων τάξης *i(n/p)* έως *(i+1)(n/p)-1* σε κάθε επεξεργαστή P_i).