Lectia 6. UN ALGORITM PARALEL PENTRU SOLUTIONAREA JOCURILOR BIMATRICEALE IN STRATEGII PURE. LUCRAREA DE LABORATOR NR. 1

6.1. Jocuri bimatriceale si situatii Nash de echilibru.

Fie dat un joc bimatriceal $\Gamma = \langle I, J, A, B \rangle$ unde I – mulţimea de indici ai liniilor matricelor, J – mulţimea de indici ai coloanelor matricelor, iar $A = \|a_{ij}\|_{\substack{i \in I \\ j \in J}}$, $B = \|b_{ij}\|_{\substack{i \in I \\ j \in J}}$ reprezintă matricele de câştig ale jucătorilor. Elementul $i \in I$ respectiv $j \in J$, se mumeste strategie pura a juvatorrului 1, respectiv a jucatorului 2, perechea de indici (i, j) reprezinta o situatie in strategii pure. Jocul se realizeaza astfel: fiecare jucator independent si „concomitent" (adica alegereile de strategii nu depind de timp) alege strategie sa, dupa care se obtine o situatie in baza careia jucatorii calculeaza castigurile care reprezinta elementul a_{ij} pentru jucatarul 1, respectiv b_{ij} pentru jucatorul 2 si cu aceasta jocul ea sfarsit.

Situaţia de echilibru este perechea de indici (i^*, j^*), pentru care se verifică sistemul de inegalităţi:

$$(i^*, j^*) \Leftrightarrow \begin{cases} a_{i^* j^*} \geq a_{ij^*} & \forall i \in I \\ b_{i^* j^*} \geq b_{i^* j} & \forall j \in J \end{cases}.$$

Vom spune că *linia i strict domină linia k* în matricea *A* dacă şi numai dacă $a_{ij} > a_{kj}$ pentru orice $j \in J$. Dacă există *j* pentru care inegalitatea nu este strictă, atunci vom spune că linia *i* domină linia *k.* Similar, vom spune: *coloana j strict domină coloana l* în matricea *B* dacă şi numai dacă $b_{ij} > b_{il}$ pentru orice $i \in I$. Dacă există *i* pentru care inegalitatea nu este strictă, atunci vom spune: *coloana j domină coloana l.*

In baza definitiei prezentam urmatorul algoritm secvential pentru determinarea situaţiei de echilibru.

Algoritmul 6.1

a) In cazul cand nu se doreste determinarea tuturor situatiilor de echilibru se eliminarea din matricea *A* şi *B* a liniilor care sunt dominate în matricea *A* şi din matricea *A* şi *B* a coloanelor care sunt dominate în matricea *B*.

b) In cazul cand se doreste determinarea tuturor situatiilor de echilibru se eliminarea din matricea *A* şi *B* a liniilor care sunt strict dominate în matricea *A* şi din matricea *A* şi *B* a coloanelor care sunt strict dominate în matricea *B*.

c) Se determină situaţiile de echilibru pentru matricea $(A', B'), A' = \|a'_{ij}\|_{\substack{i \in I' \\ j \in J'}}$ şi $B' = \|b'_{ij}\|_{\substack{i \in I' \\ j \in J'}}$ obţinută din pasul a) sau pasul b). Este clar că $|I'| \leq |I|$ si $|J'| \leq |J|$.

- Pentru orice coloană fixată în matricea A' notăm (evidenţiem) toate elementele maximale după linie. Cu alte cuvinte, se determină $i^*(j) = Arg \max_{i \in I'} a'_{ij}$ pentru orice $j \epsilon J'$.
- Pentru orice linie fixată în matricea B' notăm toate elementele maximale de pe coloane. Cu alte cuvinte, se determină $j^*(i) = Arg \max_{j \in J'} b'_{ij}$ pentru orice $i \epsilon I'$.
- Selectăm acele perechi de indici care concomitent sunt selectate atât în matricea A' cât şi în matricea B'. Altfel spus, se determină $\begin{cases} i^* \equiv i^*(j^*) \\ j^* \equiv j^*(i^*) \end{cases}$. Pentru aceasta se poate proceda astfel. Se construieşte graficul aplicaţiei i^*, adica $gri^* = \{(i, j): i = i^*(j), \forall j \in J\}$ şi corespunzător graficul aplicaţiei j^*, adica $grj^* = \{(i, j): j = j^*(i), \forall i \in I\}$. Situatiile de echilibru sunt toate situatiile cate apartin intersecţiei acestor doua grafice, adică $NE = gri^* \cap grj^*$.

d) Se construiesc situaţiile de echilibru pentru jocul cu matricele iniţiale *A* şi *B*.

Vom analiza următoarele exemple.

Exemplul 6.1. Situaţia de echilibru se determină numai în baza eliminării liniilor şi a coloanelor dominate. Considerăm următoarele matrici:

$$A=\begin{pmatrix} 400 & 0 & 0 & 0 & 0 & 0 \\ 300 & 300 & 0 & 0 & 0 & 0 \\ 200 & 200 & 200 & 0 & 0 & 0 \\ 100 & 100 & 100 & 100 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -100 & -100 & -100 & -100 & -100 & -100 \end{pmatrix}. \quad B=\begin{pmatrix} 0 & 200 & 100 & 0 & -100 & -200 \\ 0 & 0 & 100 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & 0 & -200 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Vom elimina liniile şi coloanele dominate în următoarea ordine: *linia 5, coloana 5, linia 4, coloana 4, coloana 3, linia 3, coloana 0, linia 0, coloana 1, linia 1.* Astfel obţinem matricele $A'=(200)$, $B'=(0)$ şi situația de echilibru este $(i^*, j^*)=(2,2)$ şi câştigul jucătorului *1* este *200*, al jucătorului *2* este *0.*

Exemplul 6.2. Considerăm următoarele matrice $A=\begin{pmatrix} 2 & 0 & 1 \\ 1 & 2 & 0 \\ 0 & 1 & 2 \end{pmatrix}$, $B=\begin{pmatrix} 1 & 0 & 2 \\ 2 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}$. În matricea *A* nu există linii dominate, în matricea *B* nu există colane dominate. Pentru comoditate, vom reprezenta acest joc astfel: $AB=\begin{pmatrix} (\underline{2},1) & (0,0) & (1,\underline{2}) \\ (1,\underline{2}) & (\underline{2},1) & (0,0) \\ (0,0) & (1,\underline{2}) & (\underline{2},1) \end{pmatrix}$. Uşor se observă că în acest joc nu există situaţii de echilibru în strategii pure.

Exemplul 6.3. Considerăm următoarele matrice: $A=\|a_{ij}\|_{i\in I}^{j\in J}$, $B=\|b_{ij}\|_{i\in I}^{j\in J}$, unde $a_{ij}=c, b_{ij}=k$ pentru orice $i\in I, j\in J$ şi orice constante *c,k*. Atunci mulţimea de situaţii de echilibru este $\{(i,j): \forall i\in I, \forall j\in I\}$.

6.2 Algoritmul paralel pentru determinarea situatiilor Nash de echilibru.

Structura algoritmului paralel construit va fi determinat de modul de paralelizare la nivel de date. Adica se pot utiliza urmatoarele modalitati de divizare si distribuire a matricelor A si B:

- Matricele se divizeaza in submatrici dreptunghiulare de orice dimensiune[1]. In acest caz se complica foarte tare modalitatea de construire a situatiilor de echilibru petru jocul cu matricele initiale. Penrtu lucrarea de laborator nu este obligatoriul utilizarea acestui mod de divizare a matricelor;
- Matricele se divizeaza in submatrici de tip linii sau submatrici de tip coloana. In acest caz construirea situatiilor de echilibru pentru jocul initial este destul de simpla.

Vom descrie matematic algoritmul paralel pentru determinarea situatiilor Nash de echilibru în strategii pure pentru jocul bimatriceal $G=\langle I, j, A, B\rangle$, unde $A=\|a_{ij}\|_{i\in I}^{j\in J}$, $B=\|b_{ij}\|_{i\in I}^{j\in J}$. Vom presupune ca matricea A este difizata în submatrici de tip coloana și matricea B este divizata în submatrici de tip linii. Adica vom obține un șir de submatrici $SubA^t=\|a_{ij}\|_{i\in I}^{j\in J_k}$ și $SubB^t=\|b_{ij}\|_{i\in I_k}^{j\in J}$, unde $I_k=\{i_k, i_{k+1}, \dots, i_{k+p}\}$ și $J_k=\{j_k, j_{k+1}, \dots, j_{k+p}\}$. $SubA^t$ este o submatrice care consta din p coloane ale matrice A incepand cu coloana numarul *k* și este „distribuita” procesului cu rancul *t*. Similar $SubB^t$ este o submatrice care consta din p linii ale matrice B incepand cu linia *k* si este la fel distribuita procesului cu rancul t. Folosind algoritmul 6.1 descris mai sus procesul cu rancul *t* va determina pentru orice $j_k\in J_k$ graficul aplicatiei mutivoce $i^*(j_k)=Argmax_{i\in I}a_{ij_k}$, adica $gr_k i^*=\{(i,j): i=i^*(j_k), j=j_k\}$. Similar, procesul cu rancul *t* va determina pentru orice $i_k\in I_k$ graficul aplicatiei mutivoce $j^*(i_k)=Argmax_{j\in J}b_{i_k j}$, adica $gr_k j^*=\{(i,j): i=i_k, j=j^*(i_k)\}$. In final acelaș proces *t* va determina $LineGr^t=\bigcup_k gr_k i^*$ și $ColGr^t=\bigcup_k gr_k j^*$. *Se observa ca aceasta modalitate de divizare a matricelor permite ca fiecare proces sa determine* $LineGr^t$ *si* și $ColGr^t$ *fara a executa alte operatii suplimentare. Usor se poate*

[1] Mai tarziu se va studia algoritmul 2D-ciclic de divizre a matricelor in submatrici.

arata ca exista modalitati de divizare a matricelor in care deja procesul cu rancul t nu poate determina „de unul singur” $i^*(j_k)$ *si* $j^*(i_k)$ *(de exemplu daca matricele sunt divizate in submatrici linii).*

Algoritmul paralel pentru determinarea situaţiilor de echilibru trebuie sa contina urmatoarele:

a) Eliminarea[2], în paralel, din matricea *A* şi *B* a liniilor care sunt dominate (sau strict) în matricea *A* şi din matricea *A* şi *B* a coloanelor care sunt dominate (sau strict) în matricea *B*.
b) Pentru orice proces *t* se determina submatricele SubA^t si SubB^t.
c) fiecare proces *t* determină $i^*(j_k)$ si $j^*(i_k)$ pentru orice *k*. Pentru aceasta se va folosi funcţia **`MPI_Reduce`** şi operaţia **`ALLMAXLOC`**[3] care determina toate indicele elementelor maximale si este creata cu ajutorul functiei MPI **`MPI_Op_create.`** Dupa procesul *t* determina LineGr^t si ColGr^t **in indici globali**, adica indicii elementelor din matricela A si B.
d) Procesul cu rankul 0 va determina multimea de situatii Nash de echilibru care este $NE = (\cup_t \mathrm{LineGr}^t) \cap (\cup_t \mathrm{ColGr}^t)$.

Vom exemplifica algoritmul descris mai sus. Consideram jocul din Exemplu 6.1 si fie ca *t=0,1,2*. Atunci submatricele corespunzatoare proceselor vor fi:

$$A^0 = \begin{pmatrix} 400 & 0 \\ 300 & 300 \\ 200 & 200 \\ 100 & 100 \\ 0 & 0 \\ -100 & -100 \end{pmatrix}, A^1 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 200 & 0 \\ 100 & 100 \\ 0 & 0 \\ -100 & -100 \end{pmatrix} A^2 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ -100 & -100 \end{pmatrix}$$

$$B^0 = \begin{pmatrix} 0 & 200 & 100 & 0 & -100 & -200 \\ 0 & 0 & 100 & 0 & -100 & -200 \end{pmatrix}, B^1 = \begin{pmatrix} 0 & 0 & 0 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & -100 & -200 \end{pmatrix},$$

$$B^2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & -200 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Procesul cu **rancul 0** determina: gr_0i^*={(0,0)}, gr_1i^*={(1,1)}, $\mathrm{LineGr}^0 = \{(0,0),(1,1)\}$; gr_0j^*={(0,1)}, gr_1j^*={(1,2)}, $\mathrm{ColGr}^0 = \{(0,1),(1,2)\}$.

Procesul cu **rancul 1** determina: gr_0i^*={(2,0)}, gr_1i^*={(3,1)}, $\mathrm{LineGr}^1 = \{(2,0),(3,1)\}$; gr_0j^*={(0,0), (0,1), (0,2), (0,3)}, gr_1j^*={(1,0), (1,1), (1,2), (1,3)},
$\mathrm{ColGr}^1 = \{(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3)\}$. **In indici „globali”** (adica a matricelor A si B) vom avea: $\mathrm{LineGr}^1 = \{(2,2),(3,3)\}$;
$\mathrm{ColGr}^1 = \{(2,0),(2,1),(2,2),(2,3),(3,0),(3,1),(3,2),(3,3)\}$

Procesul cu **rancul 2** determina: gr_0i^*={(0,0),(1,0), (2,0), (3,0), (4,0)}, gr_1i^*={(0,1),(1,1), (2,1), (3,1), (4,1)}, $\mathrm{LineGr}^2 = \{(0,0),(1,0),(2,0),(3,0),(4,0),(0,1),(1,1),(2,1),(3,1),(4,1)\}$;
gr_0j^*={(0,0), (0,1), (0,2), (0,3),(0,4)}, gr_1j^*={(1,0), (1,1), (1,2), (1,3),(1,4),(1,5)}, $\mathrm{ColGr}^2 = \{(0,0),(0,1),(0,2),(0,3),(0,4),(1,0),(1,1),(1,2),(1,3),(1,4),(1,5)\}$. **In indici „globali”** (adica a matricelor A si B) vom avea:
$\mathrm{LineGr}^2 = \{(0,4),(1,4),(2,4),(3,4),(4,4),(0,5),(1,5),(2,5),(3,5),(4,5)\}$; $\mathrm{ColGr}^2 = \{(4,0),(4,1),(4,2),(4,3),(4,4),(5,0),(5,1),(5,2),(5,3),(5,4),(5,5)\}$.

Procesul **cu rancul 0** va determina pentru indici globali:

$$\mathrm{LineGr} = \mathrm{LineGr}^0 \cup \mathrm{LineGr}^1 \cup \mathrm{LineGr}^2 =$$
$$\{(0,0),(1,1),(2,2),(3,3),(0,4),(1,4),(2,4),(3,4),(4,4),(0,5),(1,5),$$

[2] Acest punc nu este obligatoriu.
[3] În cazul utilizării operaţiei MAXLOC rezultatele pot fi incorecte.

$(2,5), (3,5), (4,5)\}$

si

$$\text{ColGr} = \text{ColGr}^0 \cup \text{ColGr}^1 \cup \text{ColGr}^2 =$$

$$= \{(0,1), (1,2), (2,0), (2,1), (2,2), (2,3), (3,0), (3,1), (3,2), (3,3), (4,0), (4,1), (4,2), (4,3), (4,4), (5,0), (5,1), (5,2), (5,3), (5,4), (5,5)\}.$$

Atunci NE= $\text{LineGr} \cap \text{ColGr} = \{(2,2), (3,3), (4,4)\}$

Pentru realizarea acestui algoritm pe clustere paralele sunt obligatorii următoarele:

1) Paralelizarea la nivel de date se realizează in urmatoarele moduri:
   a) Procesul cu rankul 0 iniţializează valorile matricelor *A* şi *B*, construieste submatricele SubA^t, SubB^t si le distribuie tuturor proceselor mediului de comunicare.
   b) Fiecare proces din mediul de comunicare construieste submatricele SubA^t, SubB^t si le initializeaza cu valori.
   c) Distribuirea matricelor pe procese se face astfel încât să se realizeze principiul *load balancing*.
2) Paralelizarea la nivel de operaţii se realizează şi prin utilizarea funcţiei **`MPI_Reduce`** şi a operaţiilor nou create.
3) Sa se realizeza o analiza comparativa a timpului de executie a programelor realizate cand paralelizarea la nivel de date se realizează in baza punctelor a) si b).

Mai jos vom prezenta codurile de programe in care se realizeaza variante simple ale lucrarii de laborator, si anume:

- In programul Laborator_1_var_0_a.cpp se realizeaza paralelizarea descrisa in a), matricele sunt de dimensiunea numtask*numtask, sunt initializate de procesul cu rankul 0 si submatricele sunt numai dintr-o singura linie , corespunzator coloana.
- In programul Laborator_1_var_0_b.cpp se realizeaza paralelizarea descrisa in b), matricele sunt de dimensiunea numtask*numtask, sunt initializate de procesul cu rankul 0 si submatricele sunt numai dintr-o singura linie , corespunzator coloana.

Programul Laborator_1_var_0_a.cpp.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
void invertMatrix(double* m, int mRows, int mCols, double* rez)
{
   for (int i = 0; i < mRows; ++i)
      for (int j = 0; j < mCols; ++j)
         rez[j * mRows + i] = m[i * mCols + j];
}
int main(int argc, char* argv[])
{
   int numtask, sendcount, reccount, source;
   double *A;
   double *B, *Binv;
   int myrank, root = 1;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numtask);
   double ain[numtask], aout[numtask], bin[numtask], bout[numtask];
```

```
int indA[numtask], indB[numtask];
struct {
   double val;
   int rank;
} inA[numtask], inB[numtask], outA[numtask], outB[numtask];
sendcount = numtask;
reccount = numtask;
if (myrank == root) {
   srand(time(NULL));
   printf("===== RESULTS OF THE PROGRAMM '%s' =====\n", argv[0]);
   A = (double*)malloc(numtask * numtask * sizeof(double));
   B = (double*)malloc(numtask * numtask * sizeof(double));
   for (int i = 0; i < numtask * numtask; i++) {
      A[i] = rand() / 1000000000.0;
      B[i] = rand() / 1000000000.0;
   }
   printf("Initial data\n");
   for (int i = 0; i < numtask; i++) {
      printf("\n");
      for (int j = 0; j < numtask; j++)
         printf("A[%d,%d]=%.2f ", i, j, A[i * numtask + j]);
   }
   printf("\n");
   printf("\n");
   for (int i = 0; i < numtask; i++) {
      printf("\n");
      for (int j = 0; j < numtask; j++)
         printf("B[%d,%d]=%.2f ", i, j, B[i * numtask + j]);
   }
   printf("\n");
   MPI_Barrier(MPI_COMM_WORLD);
}
else {
   MPI_Barrier(MPI_COMM_WORLD);
}
MPI_Scatter(A, sendcount, MPI_DOUBLE, ain, reccount, MPI_DOUBLE, root, MPI_COMM_WORLD);
for (int i = 0; i < numtask; ++i) {
   inA[i].val = ain[i];
   inA[i].rank = myrank;
}
MPI_Reduce(inA, outA, numtask, MPI_DOUBLE_INT, MPI_MAXLOC, root, MPI_COMM_WORLD);
if (myrank == root) {
   printf("\n");
   printf("For Matrix A: maximal element on the column and row index\n");
   for (int j = 0; j < numtask; ++j) {
      aout[j] = outA[j].val;
      indA[j] = outA[j].rank;
      printf("Column=%d, value=%.2f, row=%d\n", j, aout[j], indA[j]);
   }
```

```
    MPI_Barrier(MPI_COMM_WORLD);
  }
  else {
    MPI_Barrier(MPI_COMM_WORLD);
  }
  if (myrank == root) {
       Binv = (double*)malloc(numtask * numtask * sizeof(double));
       invertMatrix(B, numtask, numtask, Binv);
  }
  MPI_Scatter(Binv, sendcount, MPI_DOUBLE, bin, reccount, MPI_DOUBLE, root, MPI_COMM_WORLD);
  for (int i = 0; i < numtask; ++i) {
    inB[i].val = bin[i];
    inB[i].rank = myrank;
  }
  MPI_Reduce(inB, outB, numtask, MPI_DOUBLE_INT, MPI_MAXLOC, root, MPI_COMM_WORLD);
  if (myrank == root) {
    printf("\n");
    printf("For Matrix B: maximal element on the row  and column index:\n");
    for (int i = 0; i < numtask; ++i) {
      bout[i] = outB[i].val;
      indB[i] = outB[i].rank;
      printf("Row=%d, value=%.2f, column=%d\n", i, bout[i], indB[i]);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
  else {
    MPI_Barrier(MPI_COMM_WORLD);
  }
  if (myrank == root) {
       printf("\n");
    printf("Nash equilibrium are:\n");
    for (int i = 0; i < numtask; ++i) {
       if (i == indB[indA[i]]) {
         printf("{%d, %d}\n", indA[i], i);
       }
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
  else {
    MPI_Barrier(MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

Programul Laborator_1_var_0_b.cpp.

```
#include <mpi.h>
#include <stdio.h>
```

```
#include <stdlib.h>
void invertMatrix(double* m, int mRows, int mCols, double* rez)
{
   for (int i = 0; i < mRows; ++i)
      for (int j = 0; j < mCols; ++j)
         rez[j * mRows + i] = m[i * mCols + j];
}
int main(int argc, char* argv[])
{
   int numtask, sendcount, reccount, source;
   double *A;
   double *B, *Binv;
   int myrank, root = 1;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numtask);
   double ain[numtask], aout[numtask], bin[numtask], bout[numtask];
   int indA[numtask], indB[numtask];
   struct {
      double val;
      int rank;
   } inA[numtask], inB[numtask], outA[numtask], outB[numtask];
   sendcount = numtask;
   reccount = numtask;
   sleep(myrank);
      srand(time(NULL));
       if (myrank == root) {
    printf("===== RESULTS OF THE PROGRAMM '%s' =====\n", argv[0]);
      A = (double*)malloc(numtask * numtask * sizeof(double));
      B = (double*)malloc(numtask * numtask * sizeof(double));
   }
    MPI_Barrier(MPI_COMM_WORLD);
      for(int i=0;i<numtask;i++) {
      //  srand(time(NULL));
      ain[i]=rand()/1000000000.0;
      bin[i]=rand()/1000000000.0;
                                                    }
     MPI_Gather(ain, sendcount, MPI_DOUBLE, A, reccount, MPI_DOUBLE, root, MPI_COMM_WORLD);
     MPI_Gather(bin, sendcount, MPI_DOUBLE, B, reccount, MPI_DOUBLE, root, MPI_COMM_WORLD);
     if (myrank == root) {
         printf("Initial data\n");
      for (int i = 0; i < numtask; i++) {
         printf("\n");
         for (int j = 0; j < numtask; j++)
            printf("A[%d,%d]=%.2f ", i, j, A[i * numtask + j]);
                                                                  }
      Binv = (double*)malloc(numtask * numtask * sizeof(double));
         invertMatrix(B, numtask, numtask, Binv);
      printf("\n");
```

```
   for (int i = 0; i < numtask; i++) {
      printf("\n");
      for (int j = 0; j < numtask; j++)
         printf("B[%d,%d]=%.2f ", i, j, Binv[i * numtask + j]);
                                                                  }
   printf("\n");
   MPI_Barrier(MPI_COMM_WORLD);
}
else {
   MPI_Barrier(MPI_COMM_WORLD);
}
for (int i = 0; i < numtask; ++i) {
   inA[i].val = ain[i];
   inA[i].rank = myrank;
}
MPI_Reduce(inA, outA, numtask, MPI_DOUBLE_INT, MPI_MAXLOC, root, MPI_COMM_WORLD);
if (myrank == root) {
   printf("\n");
   printf("For Matrix A: maximal element on the column and row index\n");
   for (int j = 0; j < numtask; ++j) {
      aout[j] = outA[j].val;
      indA[j] = outA[j].rank;
      printf("Column=%d, value=%.2f, row=%d\n", j, aout[j], indA[j]);
   }
   MPI_Barrier(MPI_COMM_WORLD);
}
else {
   MPI_Barrier(MPI_COMM_WORLD);
}
   for (int i = 0; i < numtask; ++i) {
   inB[i].val = bin[i];
   inB[i].rank = myrank;
}
MPI_Reduce(inB, outB, numtask, MPI_DOUBLE_INT, MPI_MAXLOC, root, MPI_COMM_WORLD);
if (myrank == root) {
   printf("\n");
   printf("For Matrix B: maximal element on the row  and column index:\n");
   for (int i = 0; i < numtask; ++i) {
      bout[i] = outB[i].val;
      indB[i] = outB[i].rank;
      printf("Row=%d, value=%.2f, column=%d\n", i, bout[i], indB[i]);
   }
   MPI_Barrier(MPI_COMM_WORLD);
}
else {
   MPI_Barrier(MPI_COMM_WORLD);
}
if (myrank == root) {
       printf("\n");
```

```
    printf("Nash equilibrium are:\n");
    for (int i = 0; i < numtask; ++i) {
        if (i == indB[indA[i]]) {
          printf("{%d, %d}\n", indA[i], i);
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
  else {
    MPI_Barrier(MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```