Performance Evaluation of Parallel Programs on an Experimental GRID Computer

Ioan Ungurean, Ştefan Gheorghe Pentiuc, and Vasile Găitan

***Abstract*—In this paper there will be a discussion about parallel programs performance. These programs were developed using the OpenMPI implementation of the Message Passing Interface (MPI) standard and are being run on an experimental grid computer made of 7 desktop computers. These test applications measure execution times for applications implementing the Jacobi approximation for a linear system of equations. The output of the test application is the processing speed, measured in MFlops and obtained using more methods in order to test more aspects of inter-process communication. The final purpose of this paper is to see increasing performance in a Grid by increasing the number of computing nodes.**

***Index Terms*—grid computing, Globus Toolkit, Message Passing Interface, OpenMPI, parallel computing**

I. Introducere

ÎN această lucrare ne propunem testarea performanţelor programelor paralele[1][2]. Aceste programe sunt realizate folosind implementarea OpenMPI [3][4] a standardului Message Passing Interface (MPI) [5] şi sunt rulate pe un grid experimental format din şapte calculatoare desktop. Aplicaţiile măsoară timpul de execuţie a aplicaţiilor care implementează iteraţiile Jacobi [6] pentru aproximarea soluţiei unui sistem liniar de ecuaţii. Aplicaţiile au ca ieşire viteza de procesare calculată în MFlops şi sunt realizate în mai multe variante pentru a testa toate mecanismele de comunicaţie între procese. Scopul acestei lucrări este urmărirea creşterii performanţei de procesare a unui grid [7] prin mărirea numărului de noduri. Lucrarea a fost elaborată în urma studierii conceptelor de grid computing [7] şi programare paralelă [8]. Un GRID reprezintă o infrastructură hardware şi software care oferă acces consistent la capacităţile de calcul. De fapt ideea principală a GRID-ului o reprezintă dezvoltarea unui sistem în care accesul la resursele de calcul să fie la fel de facil ca şi accesul la resursele de electricitate. Unul din lucrurile remarcabile la infrastructura electrică este faptul că nu presupune cunoaşterea locaţiei generatorului de energie electrică şi a detaliilor privind

infrastructura reţelei electrice. Din păcate acest deziderat încă nu este îndeplinit în totalitate. Arhitectura unui GRID este structurată pe mai multe nivele, fiecare nivel având câte o funcţie specifică. În general nivelul cel mai înalt este focalizat spre utilizator. În cadrul acestei lucrări a fost realizată testarea performanţelor programelor paralele pe un grid experimental. Grid-ul experimental este realizat din calculatoare legate între ele folosind o reţea ethernet de 100Mbit şi o reţea ethernet de 1Gbit. Ca pachet software se foloseşte Globus Toolkit, iar pentru dezvoltarea aplicaţiilor paralele pe GRID este instalată o implementarea (OpenMPI) a standardului MPI (Message Passing Interface).

Ioan Ungurean – SC PROING SRL Suceava, bd. G. Enescu nr. 38, RO-720253 (e-mail: ioanu32@yahoo.com)

Ştefan Gheorghe Pentiuc – Universitatea Ştefan Cel Mare Suceava, str.Universitatii nr.13, RO-720229 Suceava (e-mail: pentiuc@eed.usv.ro)

Găitan Vasile – Universitatea Ştefan Cel Mare Suceava, str.Universitatii nr.13, RO-720229 Suceava (e-mail: gaitan@eed.usv.ro)

II. Legea lui Amdalh şi legea lui Gustafson

Teoretic, viteza obţinută prin paralelizare ar trebui să fie liniară – prin dublarea numărului de procesare ar trebui ca timpul de execuţie să se înjumătăţească, iar prin dublarea din nou a numărului de procesoare ar trebui ca timpul de execuţie să se înjumătăţească din nou. În realitate foarte puţini algoritmii pot obţine această viteză ideală. Cei mai mulţi algoritmi au o evoluţie liniară a vitezei de procesare pentru un număr mic de procesoare, care se nivelează într-o constantă pentru un număr mare de procesoare.

Viteza de execuţie pe o platforma de execuţie paralelă a unui algoritm este dată de legea lui Amdahl [9], lege care a fost formulată de Gene Amdahl in anii 1960. Această lege spune că o parte mică a programului care nu poate fi paralelizată va limita creşterea vitezei de execuţie prin paralelizare. Orice problemă matematică sau inginerească de dimensiunii mari va fi formată din părţi care pot fi paralelizate şi părţi care nu pot fi paralelizate (părţi secvenţiale) [9]. Această relaţie este dată de ecuaţia:

$$S = \frac{1}{1-P} \tag{1}$$

unde S este creşterea vitezei de execuţie a programului (ca un factor al vitezei de execuţie a programului secvenţial original) şi P este fracţia care este paralelizată. Dacă partea secvenţială a programului reprezintă 10% din timpul de execuţie, nu se poate obţine o creştere a vitezei mai mare de 10 ori, indiferent

de numărul de procesoare care sunt adăugate.

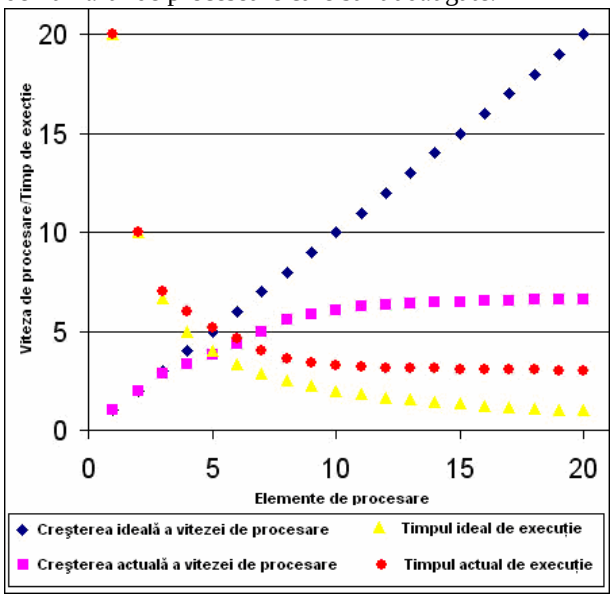

Fig. 1 Reprezentarea grafică a legii lui Amdahl

În Fig. 1 curba formată din punctele albastre ilustrează curba (liniară) care trebuie urmată de creşterea vitezei de execuţie în cazul ideal, iar punctele de culoare roz indică curba (ne-optimală) care este urmată în prezent de creşterea vitezei de execuţie. Curba formată de punctele galbene reprezintă evoluţia timpului de execuţie în cazul ideal (ca o asimptopă care se apropie de zero), în timp ce curba formată de punctele roşii indică timpul de execuţie actual (ca o asipmtopă care se apropie de o valoare mai mare de zero).

Legea lui Gustafson [9] este o altă lege din ştiinţa calculatoarelor foarte apropiată de legea lui Amdahl. Ea poate fi formulată astfel:

$$S(P) = P - a\,(P-1) \qquad (2)$$

unde P reprezintă numărul de procesoare, S este creşterea vitezei de execuţie şi partea ne-paralizabilă din procese [2]. Legea lui Amdahl presupune problema fixă şi faptul că mărimea secţiunii secvenţiale este independentă de numărul de procesoare, în timp ce legea lui Gustafson nu realizează această presupunere [9].

III. Arhitectura GRID-ului experimental

Grid-ul experimental este format din 7 calculatoare, din care unul reprezintă serverul iar celelalte 6 reprezintă nodurile de calcul. Pe fiecare calculator este instalat ca sistem de operare Scientific Linux 5.0 şi pachetele Globus Toolkit [10] şi OpenMPI (reprezintă o implementare open-source a standardului MPI).

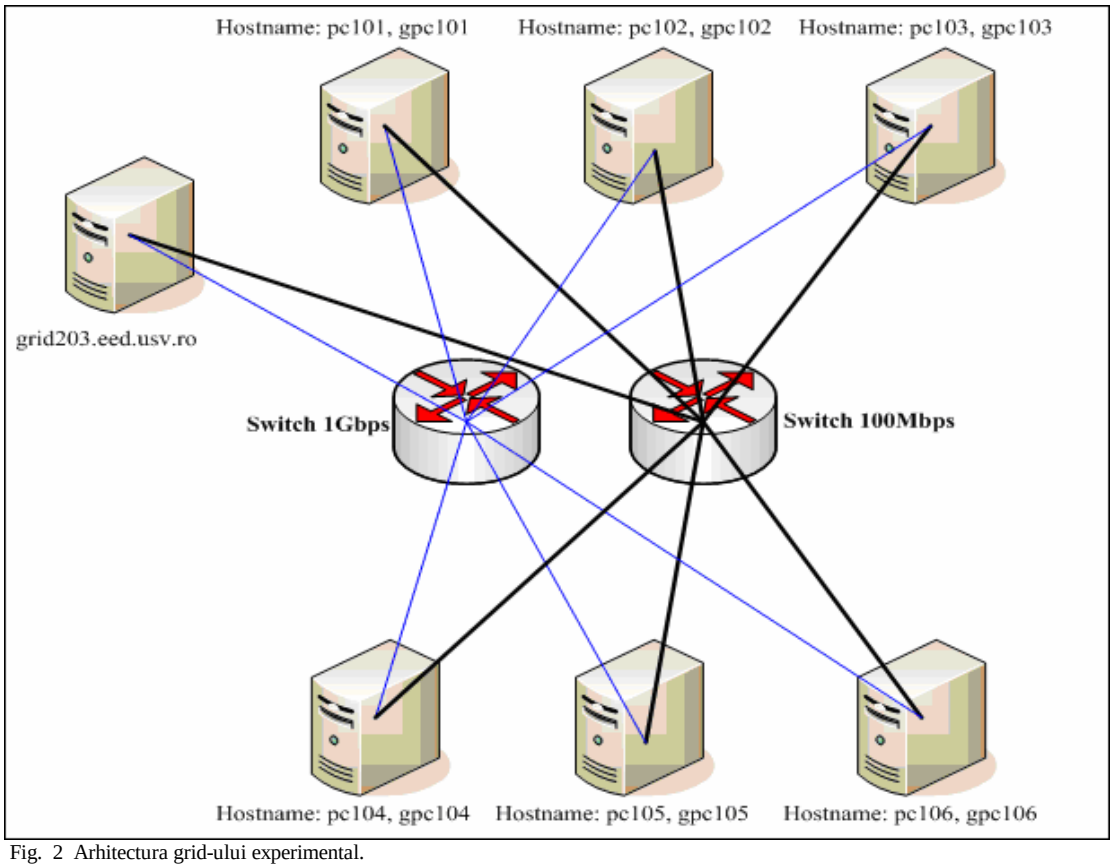

Fig. 2 Arhitectura grid-ului experimental.

În plus pe server mai este instalat şi un server de gestiune al bazelor de date ProgresSQL. Acest server este cerut de Globus Toolkit pentru a salva în baza de date toate operaţiile pe care efectuează pachetul de unelte Globus.

În Fig. 2 este prezentată arhitectura GRID-ului experimental. GRID-ul conţine 2 switch-uri (unul de 100Mbps şi unul de 1Gpbs). La aceste două switch-uri sunt conectate toate nodurile GRID-ului. În acest mod se formează 2 reţele, una se poate folosi pentru administrare şi una pentru execuţia programelor MPI. Fiecare nod are 2 host-uri (de exemplu pc101 şi gpc101) care corespund celor două reţele. Nodurile folosesc clase de IP-uri private iar serverul are IP real şi are hostname-ul grid203.eed.usv.ro.

Configuraţia hardware a calculatoarelor folosite pentru realizarea GRID-ului experimental este următoarea:

- Procesor: AMD Athlon(tm) XP 2000+ , 1.66 Ghz;
- RAM: 256MB din care 64MB estre folosit de placa video integrată.

IV. Modul de măsurare a timpului într-o aplicaţie MPI

Pentru a se putea măsura timpul de execuţie într-o aplicaţie MPI se foloseşte funcţia MPI_Wtime. Această funcţie returnează o valoare double, valoare care reprezintă timpul trecut de la un moment arbitrar din trecut. Dacă atributul MPI_WTIME_IS_GLOBAL este setat cu valoarea true atunci valoarea returnată de această funcţie este sincronizată pentru toate procesele din MPI_COMM_WORLD.

Pentru a se evidenţia (şi testa) modul de funcţionare a acestei funcţii s-a realizat o aplicaţie MPI care măsoară performanţa funcţiei memcpy pe un nod. Aplicaţia generează un tabel cu 3 coloane: prima coloană reprezintă numărul de octeţi transferaţi, a doua coloană reprezintă timpul măsurat iar a treia coloană reprezintă rata de transfer (exprimată în MegaBytes pe secundă). Timpul este măsurat prin apelarea funcţiei MPI_Wtime înainte şi după apelarea funcţiei memcpy şi efectuarea diferenţei dintre cele 2 valori. În continuare este prezentat rezultatul afişat de această aplicaţie pe un nod. Trebuie remarcat că în cazul în care aplicaţia creează mai multe taskuri pe acelaşi nod performanţele care sunt afişate de fiecare task nu scad în mod semnificativ faţă de cazul în care aplicaţia creează un singur task. În continuare este prezentată ieşirea afişată de aplicaţia de test.

```
[bacon@pc101 memtest]$ mpirun -np 1 memcpy
Lungime (octeti) Timp (sec)     Viteza (MB/sec)
4           0.000000        221.985293
8           0.000000        380.609866
16          0.000000        578.969406
32          0.000000        795.288467
64          0.000000        968.785700
128         0.000000        829.865592
256         0.000000        856.236139
512         0.000000        1204.846616
1024        0.000001        1225.320201
2048        0.000002        885.592730
4096        0.000005        885.004637
8192        0.000009        881.882907
16384       0.000018        885.801841
32768       0.000041        804.272242
65536       0.000106        620.305692
131072      0.000454        288.719231
262144      0.001643        159.560309
524288      0.003593        145.918895
1048576     0.007304        143.561747
2097152     0.014419        145.443659
```

V. Determinarea vitezei de procesare

În această secţiune se doreşte determinarea vitezei de procesoare prin implementarea iteraţiilor *Jacobi[6]* pentru aproximarea soluţiei unui sistem liniar de ecuaţii. Se va rezolva ecuaţia Laplace în două dimensiuni cu diferenţă finită.

Orice analiză numerică va demonstra că iteraţiile:

```
while (not converged) {
  for (i,j)
    xnew[i][j] = (x[i+1][j] + x[i-1][j] + x[i][j+1] +
x[i][j-1])/4;
  for (i,j)
    x[i][j] = xnew[i][j];
  }
```

vor realiza o aproximaţie pentru soluţia ecuaţiei Laplace. Există un ultim detaliu: înlocuirea valorii xnew cu media valorilor din jurul lui este aplicată doar pentru valorile din interior, valorile de la margine sunt lăsate nemodificate. În practică acest lucru înseamnă că dacă grila are dimensiunile n x n, atunci valorile:

```
x[0][j]
x[n-1][j]
x[i][0]
x[i][n-1]
```

rămân nemodificate. Deoarece valorile sunt înlocuite cu media valorilor din jurul lui această tehnică este denumită şi metoda relaxată.

Se va dori calcularea acestei aproximări în paralel. Pentru condiţia de terminare a calculului de aproximare (variabila booleană converged din iteraţia prezentată) se va calcula mai întâi valoarea diffnorm folosind următoarea secvenţă de cod:

```
diffnorm = 0;
for (i,j)
   diffnorm += (xnew[i][j] - x[i][j]) * (xnew[i][j] -
```

x[i][j]);
diffnorm = sqrt(diffnorm);

Se va stabili o valoare de referinţă (de exemplu 10e-2) iar în momentul în care valoarea diffnorm este mai mică decât valoarea de referinţă se poate spune că aproximarea s-a încheiat (converged = true).

Pentru simplitate vom considera un exemplu cu o grilă de 12 x 12 pe 4 procesoare.

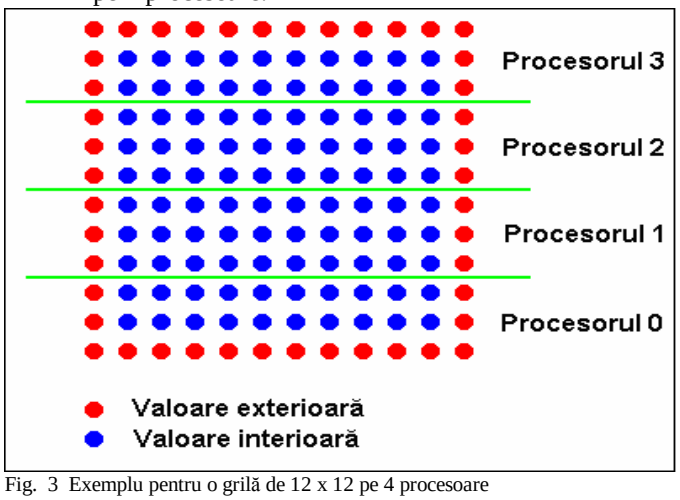

Fig. 3 Exemplu pentru o grilă de 12 x 12 pe 4 procesoare

Pentru acest exemplu se poate folosi următoarele valori iniţiale:

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3
2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

Această metodă este foarte slabă dar este folosită datorită simplităţii ei. În continuare s-au realizat câteva aplicaţii care implementează această metodă prin folosirea de tehnici diferite pentru comunicaţia dintre task-uri (tehnic care sunt definite în standardul MPI).

A. Metoda Jacobi folosind operaţiile MPI_Send/ MPI_Recv

În acest exemplu s-a realizat implementarea metodei Jacobi pentru o problemă cu dimensiunea problemei de 8 x 8. Pentru realizarea comunicaţiei s-au folosit funcţiile MPI_Send şi MPI_Recv. De asemenea soluţia este dată pentru exact 100 de iteraţii.

În continuare sunt date rezultatele (exprimate în MFlops) pentru cazul în care se creează 2,3….6 task-uri pe acelaşi nod şi pe 6 noduri diferite. Se poate observa că dacă se creşte numărul de noduri performanţa nu creşte în mod semnificativ deoarece aplicaţia realizează foarte multe operaţii de comunicaţie, operaţii care se execută mult mai lent decât calculele pe care trebuie să le execute fiecare task.

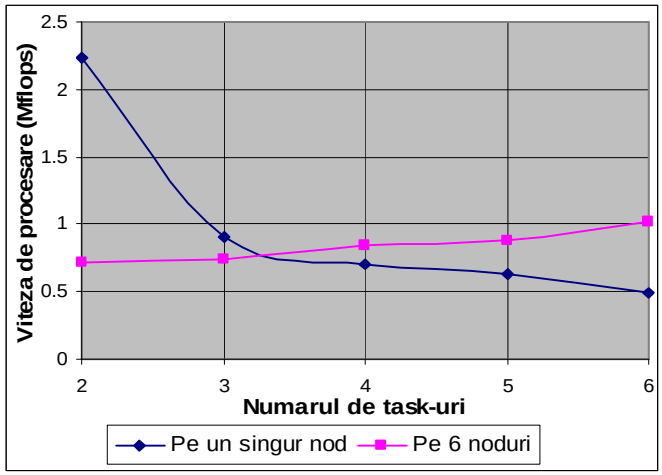

Fig. 4 Metoda Jacobi folosind operaţiile MPI_Send/ MPI_Recv

B. Metoda Jacobi folosind operaţia MPI_Sendrecv

În această secţiune sunt prezentate rezultatele obţinute de o implementare a metodei Jacobi în care se foloseşte funcţia MPI_Sendrecv. Această funcţie realizează un schimb de date între 2 task-uri. Task-ul care apelează această funcţie rămâne blocat până când se transmite un mesaj şi se recepţionează un mesaj.

În continuare sunt date rezultatele (exprimate în MFlops) pentru cazul în care se creează 2,3….6 task-uri pe acelaşi nod şi pe 6 noduri diferite. Se poate observa că nu sunt diferenţe foarte mari de performanţă faţă de cazul în care se folosesc funcţiile MPI_Send şi MPI_Recv pentru realizarea comunicaţiei între task-uri.

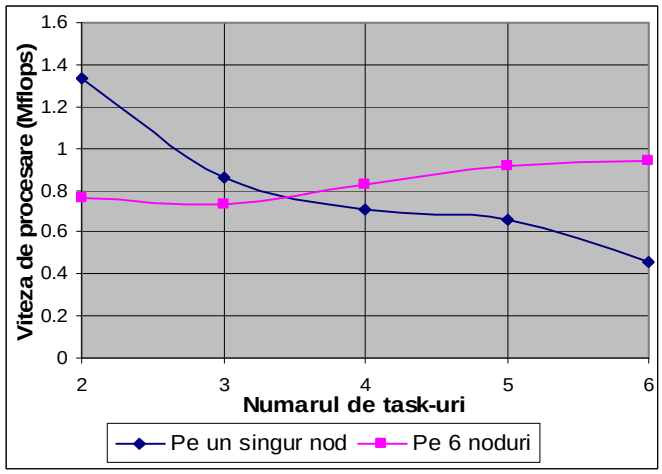

Fig. 5 Metoda Jacobi folosind operaţia MPI_Sendrecv

C. Metoda Jacobi folosind comunicaţie ”head to head”

În această secţiune sunt prezentate rezultatele obţinute de o

implementare a metodei Jacobi în care se foloseşte funcţia MPI_Sendrecv pentru a realiza o comunicaţie „head to head”.

În continuare sunt date rezultatele (exprimate în MFlops) pentru cazul în care se creează 2,3….6 task-uri pe acelaşi nod şi pe 6 noduri diferite. Se poate observa ca performanţele sunt mai mari faţă de implementarea anterioară.

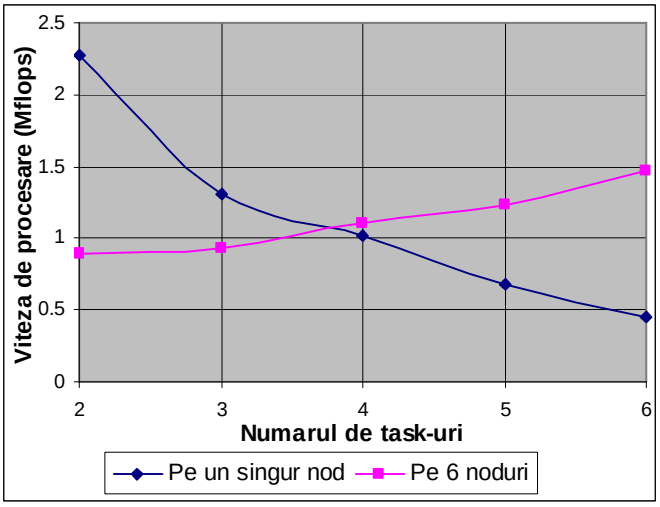

Fig. 6 Metoda Jacobi folosind comunicaţie ”head to head”

D. Metoda Jacobi folosind operaţii de tip nonblocking - versiunea 1

Această versiune foloseşte operaţii de comunicaţie de tip nonblocking pentru trimiterea şi recepţionarea mesajelor (acest lucru este realizat pentru a manipula mărimea bufferelor).

În continuare sunt date rezultatele (exprimate în MFlops) pentru cazul în care se creează 2,3….6 task-uri pe acelaşi nod şi pe 6 noduri diferite. Se poate observa că în această variantă a crescut semnificativ performanţă fată de cazul în care se foloseau operaţii de comunicaţie de tip blocking (MPI_Sendrecv).

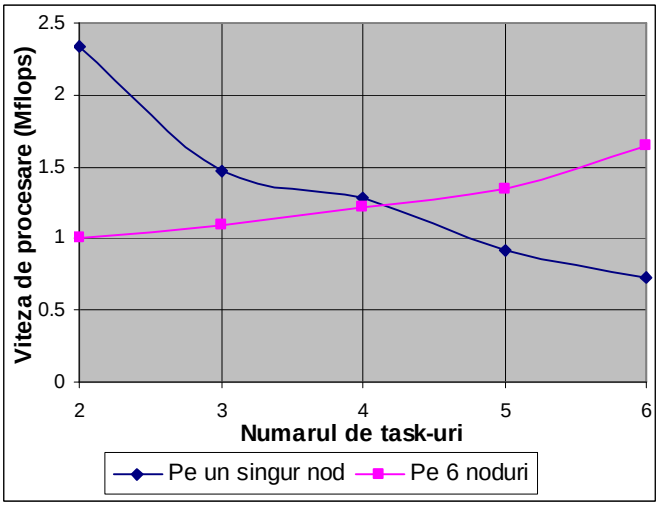

Fig. 7 Metoda Jacobi folosind operaţii de tip nonblocking -versiunea 1

E. Metoda Jacobi folosind operaţii de tip nonblocking – versiunea 2

Această versiune foloseşte operaţii de comunicaţie de tip nonblocking pentru trimiterea şi recepţionarea mesajelor (acest lucru este realizat pentru a manipula mărimea bufferelor). În această variantă se anunţă trimiterea prima dată permiţând folosirea protocoalelor de rendezvous pentru primirea datelor de task-ul destinatar minimizându-se costurile de sincronizare (acest lucru nu poate fi garantat)-

În continuare sunt date rezultatele (exprimate în MFlops) pentru cazul în care se creează 2,3….6 task-uri pe acelaşi nod şi pe 6 noduri diferite. Se poate observa că în această variantă performanţele sunt comparabile cu cele prezentate pentru prima variantă de folosire a operaţiilor de tip nonblocking.

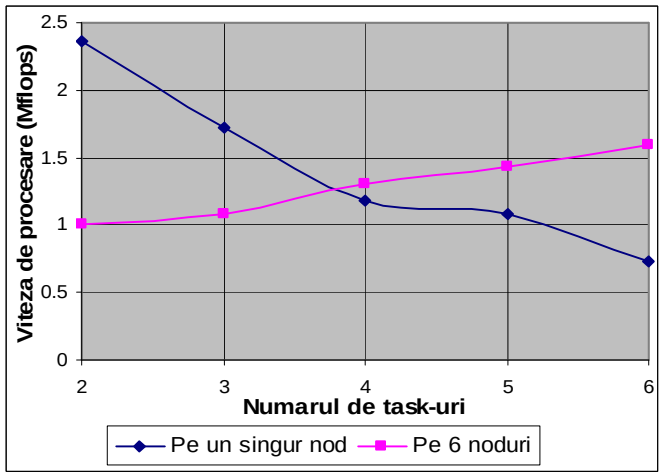

Fig. 8 Metoda Jacobi folosind operaţii de tip nonblocking -versiunea 1

F. Metoda Jacobi folosind transmitere sincronizată

Operaţiile de transmitere sincronizate permite transmiterea fără realizarea buffer-ării. Acest lucru se poate realiza folosind funcţia MPI_Rsend. Exemplul anterior a fost modificat să folosească această tehnică de transmitere.

În continuare sunt date rezultatele (exprimate în MFlops) pentru cazul în care se creează 2,3….6 task-uri pe acelaşi nod şi pe 6 noduri diferite.

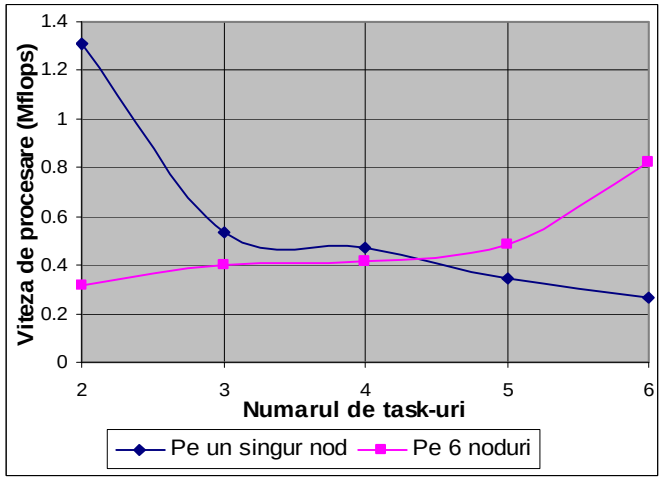

Fig. 9 Metoda Jacobi folosind transmitere sincronizată

G. *Metoda Jacobi folosind funcţia MPI_Rsend*

Într-o buclă simplă Jacobi la sfârşitul fiecărei iteraţii se poate folosii funcţia MPI_Allreduce pentru a testa convergenta soluţiei. Acest lucru furnizează un punct de sincronizare în program. În exemplul curent se foloseşte această sincronizare pentru a permite folosirea rutinei MPI_Rsend. În acest caz recepţia (MPI_Isend) este postată înaintea punctului de sincronizare iar transmisia (MPI_Rsend) este postată după punctul de sincronizare.

În continuare sunt date rezultatele (exprimate în MFlops) pentru cazul în care se creează 2,3….6 task-uri pe acelaşi nod şi pe 6 noduri diferite.

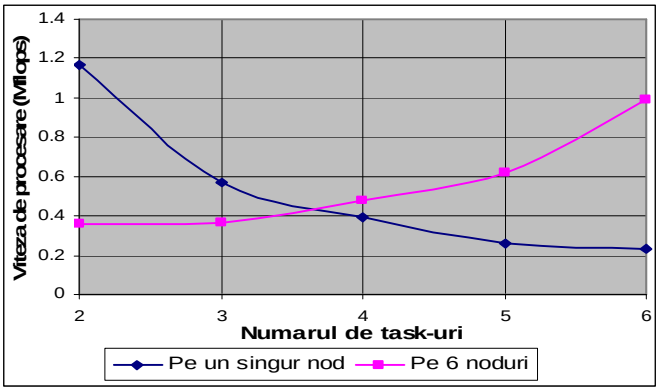

Fig. 10 G. Metoda Jacobi folosind funcţia MPI_Rsend

H. *Metoda Jacobi folosind funcţia MPI_Irsend*

Acest exemplu este asemănător cu cel anterior cu singura diferenţă că în locul funcţie MPI_Rsend se foloseşte funcţia MPI_Irsend (această funcţie implementează o operaţie de transmisie de tip nonblocking).

În continuare sunt date rezultatele (exprimate în MFlops) pentru cazul în care se creează 2,3…6 task-uri pe acelaşi nod şi pe 6 noduri diferite.

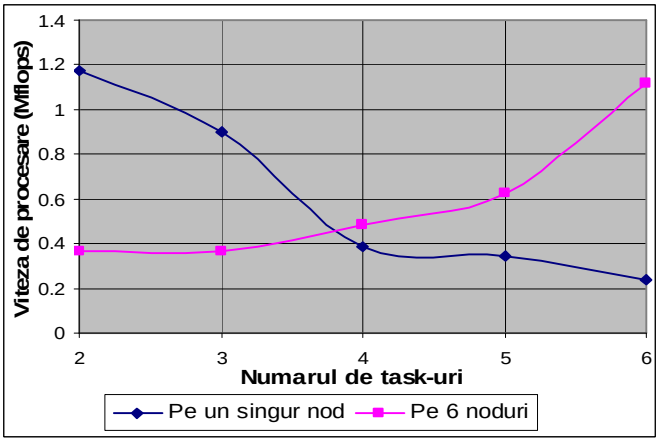

Fig. 11 Metoda Jacobi folosind funcţia MPI_Irsend

I. *Metoda Jacobi folosind suprapunerea comunicaţiei cu calculele*

In acest exemplu calculele sunt împărţite in două părţi: o parte care are nevoie de date de la alte procesoare şi o parte care nu are nevoie. Partea care este independentă de alte task-uri reprezintă interiorul domeniului (relativ la fiecare task) şi partea care are nevoie de date externe reprezintă graniţa. Comunicaţia este pornită, se realizează comunicaţie pentru datele din interior, se termină comunicaţia şi se realizează calculele pentru datele de la graniţă. În acest fel se realizează o suprapunere a calculelor şi a comunicaţie.

În continuare sunt date rezultatele (exprimate în MFlops) pentru cazul în care se creează 2,3….6 task-uri pe acelaşi nod şi pe 6 noduri diferite. Se observă o creştere a performanţei dacă se foloseşte această tehnică.

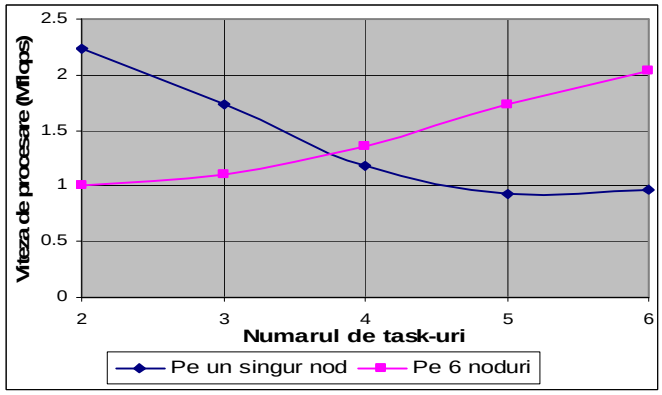

Fig. 12 Metoda Jacobi folosind suprapunerea comunicaţiei cu calculele

VI. Concluzii

În urma testelor efectuate se pot observa diferenţele performanţelor programelor paralele în funcţie de tehnicile de comunicaţie care sunt folosite. În cadrul testelor care au fost realizate s-a subliniat importanţa comunicării în cadrul programelor paralele prin evidenţierea efectelor pe care o are asupra performanţelor. Aceste teste evidenţiază valabilitatea legii lui Amdalh (prin dublarea numărului de noduri nu se dublează performanţa).

References

[1] I. Foster: Designing and Building Parallel Programs, Addison-Wesley 1995
[2] V. Niculescu: Modele de elaborare a algoritmilor paraleli, PhD. Thesis, Univ. Babes-Bolyai, 2002
[3] Open MPI: Open Source High Performance Computing, www.open-mpi.org
[4] Rohit Chandra, Leo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, Ramesh Menon, Parallel Programming in OpenMP, 2007
[5] The Message Passing Interface (MPI) standard, [http://www-unix.mcs.anl.gov/mpi/], 2000.
[6] Jacobi method, http://en.wikipedia.org/wiki/Jacobi_method
[7] Ian Foster, Carl Kesselman, The Grid - Blueprint for a new computing infrastructure, Morgan Kaufmann, 1999
[8] SKILLICORN, D.: Foundations of Parallel Programming, Cambridge International Series on Parallel Computations, 1994.
[9] Reevaluating Amdahl's Law (1988). Communications of the ACM 31(5), pp. 532–33
[10] Globus Toolkit, www.globus.org