UAIC
Fall Semester 2022

Teacher: Cristian Vidraşcu
Student: Gheorghiţă Mutu

Master: SAI I
SISTEME DE OPERARE DISTRIBUITE

—

Tema 2: Calculul paralel a produsului a două matrici, folosind OpenMP şi un calculator SMP

Introducere

I) Calculul paralel a produsului a două matrici
II) Studiul experimental al performanţei programelor paralele realizate la pct.I)

I) **Implementare**: scrieţi un program, utilizând OpenMP, în care să implementaţi diversele variante posibile de paralelizare a codului secvenţial ce calculează produsul a două matrici.

Se dau două matrici (bi-dimensionale): A[n,m] , B[m,r] şi se cere să se calculeze produsul lor: C[n,r] . Calculul secvenţial al produsului C = A*B arată astfel:

```
for (i = 0; i < n; i++)
  for (j = 0; j < r; j++)
    for (k = 0; k < m; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Variante posibile de paralelizare a produsului ar fi:
- paralelizarea (doar a) buclei interioare, i.e. for k ...
- paralelizarea (doar a) buclei 'din mijloc', i.e. for j ...
- paralelizarea (doar a) buclei exterioare, i.e. for i ...

Alte variante de paralelizare:
- fie folosind sintaxa #pragma omp parallel for ... collapse (N), și astfel obținem încă 3 variante de paralelizare (i.e., primul for cu al doilea, al doilea for cu al treilea, și respectiv toate cele 3 bucle for);
- fie folosind 'nested parallelism', și astfel obținem alte 4 variante de paralelizare (i.e., primul for 'nested' cu al doilea, al doilea for 'nested' cu al treilea, toate cele 3 bucle for 'nested', și respectiv primul for 'nested' cu al treilea !!!).

Implementați cele 10 variante de paralelizare descrise mai sus și calculați timpii de execuție, doar pentru calculele propriu-zise, fără duratele necesare pentru generat datele de intrare A,B și pentru verificat corectitudinea calculului efectuat al matricii rezultat (prin compararea cu rezultatul obținut cu algoritmul secvențial).

P.S. Și totuși, mai sunt cumva posibile și alte variante de paralelizare decât cele 10 de mai sus?
Show / Hide the answer

Da, mai sunt posibile două variante de paralelizare, ce se obțin prin combinarea paralelizării cu 'collapse' și cu 'nested', în felul următor:

- (primul for 'collapsed' cu al doilea for) și amândouă 'nested' cu al treilea for;

- primul for 'nested' cu (al doilea for 'collapsed' cu al treilea for

II) **Studiul experimental** al performanței programelor paralele realizate la pct.I)

Studiu experimental: să se studieze eficiența (i.e., performanța exprimată prin prisma timpului de execuție necesar pentru calculul produsului matricilor) pentru programul cu versiunile paralele de calcul implementate la pct.I), pe baza diverșilor factori ce influențează timpul de execuție a calculelor.

Experimente:
Care dintre cele 10 variante paralele (+1, cea secvențială) este mai eficientă, și în ce condiții?
Studiați cum variază timpul de execuție a calculului C = A*B, în funcție de următorii parametri:
- dimensiunile n,m,r ale matricilor A și B;
- numărul de thread-uri alocat pentru fiecare dintre for-urile paralele (respectiv, de observat limitarea dată de numărul de core-uri HW);
- modul de alegere a 'schedule'-ului (i.e., "împărțirea iterațiilor în chunk-uri") pentru fiecare dintre for-urile paralele.

Alți factori/parametri ce ar putea influența timpul de execuție și pe care i-ați putea studia:

- influența platformei pe care se rulează programul (numărul de core-uri HW, cu HT sau nu; rulat direct pe mașina gazdă sau într-o mașină virtuală, etc.);
- influența gestiunii memoriei virtuale (i.e. rata erorilor de pagină). *Recomandare*: pentru minimizarea lor, matricea B ar putea fi stocată în memorie "pe coloane" (i.e., memorată transpusa lui B și adaptat codul în mod corespunzător);
- influența folosirii de containere Docker (și orchestratoare pentru acestea) pentru un control mai fin al resurselor gazdei alocate pentru execuția programului;
- alți factori de influență... ???

Pentru a realiza acest studiu, culegeți date experimentale despre timpii de execuție și realizați reprezentarea grafică a performanței pe baza timpilor și a parametrilor (i.e., dimensiunile n,m,r ale matricilor, numărul de thread-uri folosit la rulare, etc.), extrăgând la final și niște *concluzii* despre valorile optime ale acestor parametri.

Recomandare: executați mai multe (e.g., 5) rulări pentru un același set de date de intrare (i.e., matricile A,B, plus ceilalți parametrii: numărul de thread-uri, tipul 'schedule'-ului, etc.), și luați media aritmetică a timpilor de execuție pentru acel set (A,B,...), pentru a minimiza impactul factorilor aleatori, e.g. "încărcarea" (load-ul) sistemului de calcul.

Fișiere suplimentare

Proiect C++ folosind CMake

- CMakeList.txt Top Level (global configuration)
- CMakePresets.json (cross-platofrm build template file pentru Visual Studio)
- CMakeList.txt (inner level - construieste .obj-urile)
- Generator (.h & .cpp - generează/randomizează matricile de input)
- Solver (.h & .cpp - conține 17 versiuni pentru a înmulți două matrici)
- main.cpp (executable entry point)
- .vs folder (cu taskurile necesare pentru build și lansare în Visual Studio)

Date și statistici

- log.out (fișierul generat de executabilul proiectului)
- output.csv (postprocesarea fișierului log.out)
- output.txt (cea mai bună versiune pentru fiecare caz extrasă pe baza output.csv)
- Anexă - Date, grafice și statistici.pdf (toate datele, graficele și statisticile extrase din output.csv)

- Anexă - Date, grafice și statistici.ipynb (fișierul în format Jupyter Notebook / Python ce pot fi încărcat și modificat - pe baza lui a fost genereat PDFul anexă)
- SOD Homework 02.pdf (fișierul curent)

Caracteristicile sistemului folosit

De ce Windows? Prima opțiune a fost Windows (MSVC) & OpenMP. Din păcate, MSVC suportă doar OpenMP 2.x (2002) care nu oferă suportul de `collapse` al buclelor for. Alternativ am folosit compilerul CLang pe Windows sub Visual Studio. Pe Ubuntu 22.04 default este gcc 11 care a cărei versiune de OpenMP este 4.5.

Compilers (prin CMake sau bash):

clang-cl.exe:
[CMake] -- Found OpenMP_C: -Xclang -fopenmp (found version "5.0")
[CMake] -- Found OpenMP_CXX: -Xclang -fopenmp (found version "5.0")
[CMake] -- Found OpenMP: TRUE (found version "5.0")

cl.exe
[CMake] -- Found OpenMP_CXX: -openmp (found version "2.0")
[CMake] -- Found OpenMP: TRUE (found version "2.0")

gcc
echo _OPENMP | gcc -fopenmp -E -x c - | tail -1
201511 => Nov 2015 OpenMP 4.5 (https://www.openmp.org/news/press-releases)

Windows, MSVC, CLang & OpenMP 5

Spec	Value
WindowsBuildLabEx	19041.1.amd64fre.vb_release.191206-1406
WindowsCurrentVersion	6.3
WindowsEditionId	Professional
WindowsInstallationType	Client

WindowsProductName	Windows 10 Pro
WindowsVersion	2009
CsHypervisorPresent	True
CsManufacturer	Dell Inc.
CsModel	Latitude 5411
CsNumberOfLogicalProcessors	12
CsNumberOfProcessors	1
CsProcessors	{Intel(R) Core(TM) i7-10850H CPU @ 2.70GHz}
CsSystemFamily	Latitude
CsSystemType	x64-based PC
CsTotalPhysicalMemory	51122126848
CsPhyicallyInstalledMemory	50331648
OsName	Microsoft Windows 10 Pro
OsType	WINNT
OsVersion	10.0.19045
OsBuildNumber	19045
OsBuildType	Multiprocessor Free
OsArchitecture	64-bit

HyperVisorPresent	True

Legendă / Parametri folosiți

Name	Seminification
Series	De câte ori s-au rulat operațiile pe matrice pentru a face o medie
Threads	Numărul de threaduri pe care s-au făcut operațiile
V0	Execuție secvențiala
V1	Execuție paralelă: `#pragma omp for` pe prima buclă
V2	Execuție paralelă: `#pragma omp for` pe a doua buclă
V3	Execuție paralelă: `#pragma omp for` pe a treia buclă
V4	Execuție paralelă: `collapse` pe cele 3 bucle
V5	Execuție paralelă: `collapse` pe primele două bucle
V6	Execuție paralelă: `collapse` pe ultimele două bucle
V7	Execuție paralelă: `#pragma omp for` pe prima buclă și `parallel for` pe a doua buclă
V8	Execuție paralelă: `#pragma omp for` pe prima buclă și `parallel for` pe a treia buclă
V9	Execuție paralelă: `#pragma omp for` pe a doua buclă și `parallel for` pe a treia buclă
V10	Execuție paralelă: `#pragma omp for` pe prima buclă și `parallel for` pe a doua și a treia buclă
V11	Execuție paralelă: `collapse` pe primele două bucle și `parallel for`

	pe ultima buclă
V12	Execuție paralelă: `#pragma omp for` pe prima buclă și `collapse` pe ultimele două bucle
V13	reduction V #03 în locul secțiuni critice
V14	reduction V #08 în locul secțiuni critice
V15	reduction V #09 în locul secțiuni critice
V16	reduction V #10 în locul secțiuni critice
V17	reduction V #11 în locul secțiuni critice
Average	Media timpilor de execuție (series)
Worst	Cel mai rău timp de execuție
Best	Cel mai bun timp de execuție

Date raw

Permutări generate

Folosind Python:

from itertools import product
n = [1, 2, 3] result = product(n, repeat=3)
print(list(result))

Generează lista (27) de tuple {n, m, r} (din cate doar 21 am testat din cauza mărimii inputului):

Case number	N	M	R
#0	10	10	10
#1	10	10	100

#2	10	10	1000
#3	10	100	10
#4	10	100	100
#5	10	100	1000
#6	10	1000	10
#7	10	1000	100
#8	10	1000	1000
#9	100	10	10
#10	100	10	100
#11	100	10	1000
#12	100	100	10
#13	100	100	100
#14	100	100	1000
#15	100	1000	10
#16	100	1000	100
#17	100	1000	1000
#18	1000	10	10
#19	1000	10	100
#20	1000	10	1000
#21	1000	100	10
#22	1000	100	100
#23	1000	100	1000
#24	1000	1000	10
#25	1000	1000	100
#26	1000	1000	1000

Exemplu output rulare (timpii sunt în nanosecunde)

CASE #8
Generated matrix A[10][1000], B[1000][1000] and C[10][1000].

SERIES #10 THREADS #1
Processing RUN #0
Processing RUN #1
Processing RUN #2
Processing RUN #3
Processing RUN #4
Processing RUN #5
Processing RUN #6
Processing RUN #7
Processing RUN #8
Processing RUN #9

V #0 Best: 87571300 Average: 98314660 Worst: 130715300
Runs: [87571300, 90378000, 94684900, 90417300, 97788100, 111297100, 130715300, 93294200, 94896800, 92103600,]

V #1 Best: 18562300 Average: 22273840 Worst: 30158200
Runs: [20592100, 18562300, 19361400, 21201800, 18838500, 30158200, 29478100, 26127900, 18903400, 19514700,]

V #2 Best: 16222300 Average: 21103860 Worst: 27078000
Runs: [20454900, 16321500, 24247200, 16791500, 16222300, 25158200, 24260800, 20638900, 27078000, 19865300,]

V #3 Best: 3602165900 Average: 4699176260 Worst: 8226040300
Runs: [4062692700, 3902109400, 3882941000, 4668235600, 3602165900, 4135386600, 5082599500, 4256544600, 5173047000, 8226040300,]

V #4 Best: 3412209300 Average: 4349182360 Worst: 9540707800
Runs: [9540707800, 4115171300, 3619386400, 4283918100, 3412209300, 3442629300, 3915507800, 3817643000, 3648883000, 3695767600,]

V #5 Best: 15813800 Average: 18111120 Worst: 24175000
Runs: [16508500, 16255100, 24175000, 16292000, 15813800, 15958800, 17444600, 16493600, 20261200, 21908600,]

V #6 Best: 3557644100 Average: 4549229890 Worst: 8394523800
Runs: [4132595800, 4182622700, 3865284100, 3933979700, 8394523800, 5633502300, 3884669000, 3699862100, 3557644100, 4207615300,]

V #7 Best: 19072500 Average: 19828140 Worst: 20893000
Runs: [19263400, 20530000, 19508500, 19072500, 19963900, 19148800, 19726900, 19758800, 20415600, 20893000,]

V #8 Best: 21930900 Average: 24987920 Worst: 42713600
Runs: [22637700, 42713600, 23091800, 23464000, 22673200, 24361900, 22507000, 21930900, 24258200, 22240900,]

V #9 Best: 18977200 Average: 25802910 Worst: 37405100
Runs: [20389600, 29580000, 23392700, 26197500, 33654000, 37405100, 25390200, 18977200, 23807200, 19235600,]

V #10 Best: 20892000 Average: 32963060 Worst: 114678700
Runs: [22653300, 32399400, 23415000, 24301300, 20892000, 114678700, 21386000, 24030300, 23009800, 22864800,]

V #11 Best: 18644900 Average: 24410440 Worst: 46466500
Runs: [19517600, 31480800, 19107300, 20804600, 24940400, 46466500, 19560100, 22658000, 20924200, 18644900,]

V #12 Best: 22438000 Average: 26918210 Worst: 40276800
Runs: [30889000, 22535200, 25524800, 32523000, 25388200, 40276800, 22742900, 22885700, 22438000, 23978500,]

Concluzii

Cea mai eficientă variantă?

Depinde. Depinde de dimensiunile matricii și de existența sau nu a unei secțiuni critice în paralelizare.

Astfel, versiunile 3, 4, 6, 8, 9, 10, 11, 12 unde există o astfel de secțiune sunt cele mai puțin performante indiferent de input.

Metodele ce folosesc reduction sunt mult mai eficiente decat cele cu o secțiune critică.

Metoda secvențială este cea mai eficientă atunci când inputul este mic iar calculul întrece overheadul generat de managementul poolului firelor de execuție de către OpenMP.

Topul variantelor overall

Caz	Versiune
#0 (n=10 m=10 r=10)	0: 8, 5: 7, 1: 5
#1 (n=10 m=10 r=100)	5: 8, 1: 4, 2: 3, 0: 1, 7: 1
#2 (n=10 m=10 r=1000)	7: 4, 2: 4, 5: 2, 0: 1, 1: 1
#3 (n=10 m=100 r=10)	17: 10, 5: 5, 14: 5, 15: 5, 16: 4, 1: 2, 0: 1
#4 (n=10 m=100 r=100)	5: 4, 16: 3, 14: 2, 0: 1, 7: 1, 2: 1, 1: 1
#5 (n=10 m=100 r=1000)	5: 4, 7: 3, 16: 2, 0: 1, 2: 1, 14: 1
#6 (n=10 m=1000 r=10)	17: 6, 16: 3, 14: 2, 13: 1, 15: 1
#7 (n=10 m=1000 r=100)	16: 8, 15: 2, 13: 1, 17: 1
#8 (n=10 m=1000 r=1000)	17: 7, 15: 3, 16: 2
#9 (n=100 m=10 r=10)	1: 8, 5: 2, 0: 1, 2: 1
#10 (n=100 m=10 r=100)	5: 5, 7: 3, 1: 3, 0: 1
#11 (n=100 m=10 r=1000)	7: 5, 5: 5, 0: 1, 2: 1
#12 (n=100 m=100 r=10)	14: 6, 5: 4, 17: 2, 0: 1, 15: 1
#13 (n=100 m=100 r=100)	5: 9, 0: 1, 2: 1, 7: 1
#14 (n=100 m=100 r=1000)	5: 5, 2: 3, 1: 3, 0: 1
#15 (n=100 m=1000 r=10)	14: 5, 17: 4, 16: 2, 13: 1
#16 (n=100 m=1000 r=100)	16: 5, 14: 4, 13: 1, 15: 1, 17: 1
#17 (n=1000 m=10 r=10)	5: 7, 1: 6, 0: 1
#18 (n=1000 m=10 r=100)	5: 5, 7: 4, 1: 3, 0: 1
#19 (n=1000 m=10 r=1000)	2: 4, 1: 4, 7: 3, 0: 1
#20 (n=1000 m=100 r=10)	5: 7, 7: 2, 0: 1, 14: 1, 2: 1
#21 (n=1000 m=100 r=100)	5: 5, 7: 3, 2: 2, 0: 1, 14: 1

Topul variantelor:
V #0 => 1
V #1 => 1
V #2 => 2
V #5 => 10
V #7 => 3
V #14 => 2
V #17 => 3
V #16 => 2

Din cele 21 de variante de input testate, V #5 (schedule(runtime) collapse(2) first 2) ocupă primul loc în 10 teste.

Varianta constă în:

```
#pragma omp parallel shared(c) private(i, j, k)
{
#pragma omp for schedule(runtime) collapse(2) nowait
	for (i = 0; i < n; i++)
	{
		for (j = 0; j < r; j++)
		{
			for (k = 0; k < m; k++)
			{
				c.at(i).at(j) += a.at(i).at(k) * b.at(k).at(j);
			}
		}
	}
}
```