Metodi Computazionali della Fisica

9 ottobre 2023

Parte I
Introduzione

1 Obbiettivi del corso

Fin dal primo anno di corso in Fisica , o persino dal liceo, ci si accorge che lo studio della fisica presenta delle equazioni per la descrizione di fenomeni naturali, che possono essere anche differenziali o differenziali alle derivate parziali, e che in molti casi non porgono soluzioni analitiche. Per soluzione analitica intendiamo una formula matematica immediatamente valutabile.

Un buon esempio è dato dall'integrazione delle equazioni del moto del pendolo semplice (vedi Fig. 1):

$$\frac{d^2\theta\left(t\right)}{dt^2}+\frac{g}{l}\sin\left(\theta\left(t\right)\right)=0 \tag{1}$$

con le condizioni iniziali :

$$\begin{cases}\theta(t=0)=\theta_0\\ \frac{d\theta}{dt}\left(t=0\right)=0\end{cases} \tag{2}$$

noi sappiamo risolvere analiticamente Eq. 1 solo nel caso limite di piccolo θ_0 in tal caso espandendo in serie di Taylor e approssimando:

$$\sin\left(\theta\left(t\right)\right)\approx\theta\left(t\right) \tag{3}$$

andiamo a trovare la nota formula del moto armonico:

$$\theta\left(t\right)=\theta_o\cos\left(\sqrt{\frac{g}{l}}t\right) \tag{4}$$

Anche se l'Eq.1 non ammette nel caso generale soluzioni analitiche possiamo calcolare *numericamente* soluzioni particolari come, ad esempio, il periodo di oscillazione o l'angolo ad un instante di tempo dato. In principio tali soluzioni numeriche posso essere ottenute con precisione arbitraria. In pratica la precisione sarà limitata dei mezzi di calcolo disponibili.

Altri esempi fra quelli probabilmente già ben noti ai partecipanti al corso sono l'integrazioni delle equazioni del moto per un sistema di $N\geqslant 3$ corpi interagenti fra di loro o il calcolo del campo elettrico per una distribuzione di carica non banale.

In questo corso impareremo a risolvere le equazioni della fisica usando metodi numerici. E' doveroso notare che tali metodi sono stati introdotti ed ampiamente usati ben prima dell'avvento dei computer. Un esempio può essere l'uso dell'espansione di serie di Taylor per valutare le funzioni trigonometriche. Oggi però tali metodi numerici vengono sempre *implementati* in un software e i calcoli vengono svolti dai computers. Pertanto è opportuno parlare di *metodi computazionali* piuttosto che di metodi numerici.

Si tenga presente che l'utilizzo dei computers nella fisica non è limitato al pur vasto problema della soluzione di equazioni ma riguarda anche il largo settore dei sistemi di acquisizioni dati in laboratorio, della loro analisi, trattamento e visualizzazione. Essendo questo un corso introduttivo esso sarà focalizzato quasi esclusivamente sulla prima parte.

2 Il computer

2.1 Numeri in virgola mobile

Anche se questo corso presuppone delle basi di informatica, è opportuno trattare in forma un po' dettaglita gli aspetti del computer che influenzano maggiormente l'implementazione e l'uso dei metodi computazionali per la fisica. Un computer

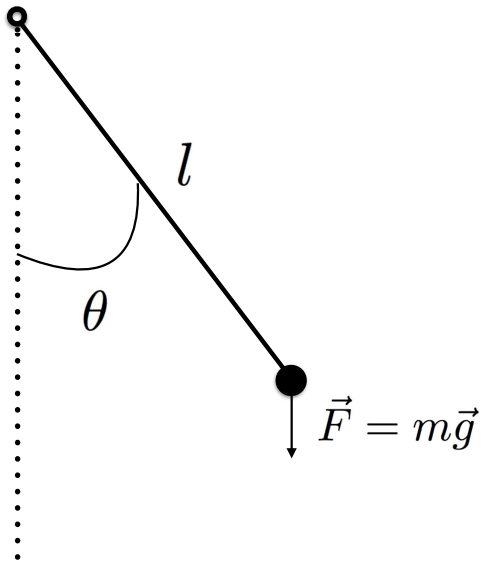

Figura 1: Il ben noto pendolo semplice

opera con dati espressi come serie di ***byte*** ogni *byte* è composto da 8 ***bit***. Un bit è una cifra in un sistema binario e pertanto può assumere solo due valori che vengono contrassegnati 0 e 1. Pertanto un byte può assumere $2^8 = 256$ valori diversi. Per rappresentare valori numerici i computers ricorrono alla rappresentazione in virgola mobile, come in:

$$f = \pm 0.c_1 c_2 c_3 c_4 c_5 \cdot 10^{\pm d_1 d_2} \tag{5}$$

dove le cifre $c_1 \ldots c_5$ sono cifre da 0 a 9 che formano la *mantissa* e le cifre $d_1\, d_2$ individuano l'esponente. Al giorno d'oggi la rappresentazione in virgola mobile è regolata dallo standard **IEEE754**. Esso prevede in particolare due formati per i quali le operazioni fondamentali come comparazione, somma, sottrazione, moltiplicazione, sono implementate a *livello hardware*. Essi sono la **singola** e la d**oppia precisione**. Un numero in virgola mobile in singola precisione viene immagazzinato con 4 bytes, 8 bytes invece occorrono per la doppia precisione.

Nel caso di un numero a singola precisione uno dei 32 bits viene usato per immagazzinare il segno (+ o -) 8 bits per l'esponente e i restanti 23 per la mantissa. Questo vuol dire che passando alla rappresentazione decimale la mantissa contiene da 6 a 9 cifre (significative) e, attenzione, l'esponente varia da -126 a +127. Questo perché -127 e +128 vengono usati per contrassegnare quando il risultato di un'operazione matematica è anomalo: **underflow** per un numero il cui valore assoluto è più piccolo del minimo valore rappresentabile, + o - **infinity** per un numero il cui valore assoluto è maggiore del massimo valore rappresentabile, **NaN (not a number)** per il risultato di operazioni come divisione per zero.

Un numero a doppia precisione riserva 1 dei sui 64 bits per il segno 11 bits per l'esponente e i restanti 52 bits per la mantissa. Ciò risulta in 15-17 cifre significative decimali e un esponente da -1022 a +1023 oltre alla trattazione dei casi particolari vista prima.

2.2 Schema di base di un computer

Vogliamo ora passare in rassegna gli elementi di un computer che rivestono particolare importanza nel calcolo numerico soprattutto in termini di velocità di calcolo. Escludendo memorie di massa (hard-disk,ssd) scheda grafica, scheda di rete, ci concentriamo sulla memoria e la **CPU (central processing unit)** delle quali riportiamo in Fig. 2 riportiamo uno schema molto semplificato. La CPU si occupa di eseguire le istruzioni (*control unit)* operando su appositi registri dove vengono immagazzinati i dati che vengono processati nella *ALU* (unità logico aritmetica) e nella *FPU (floating point unit)* per i dati in virgola mobile. La memoria *RAM* è connessa alla CPU tramite un *memory bus.* La CPU dispone anche di una *cache memory*, memoria RAM ad accesso/lettura e scrittura rapida.

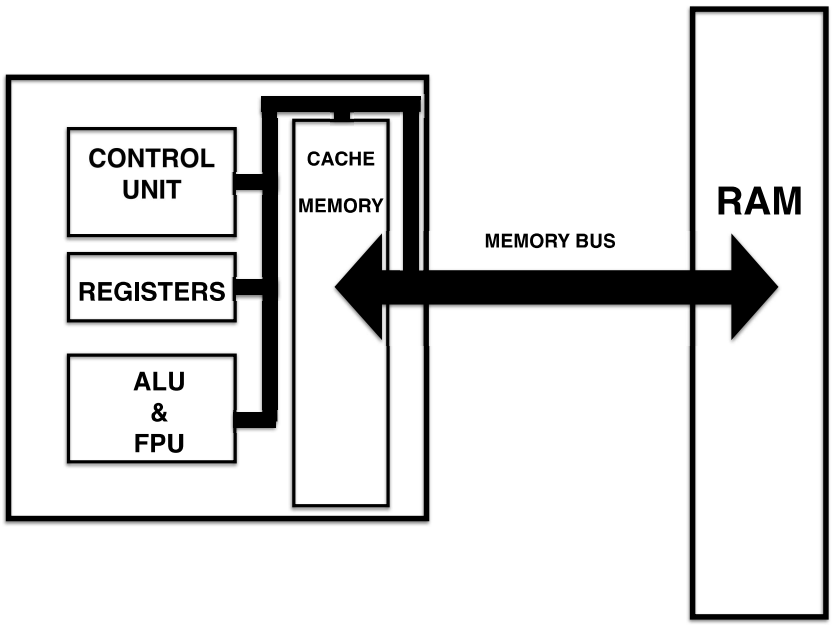

Figura 2: Schema delle connessioni tra CPU e RAM in computer

Una CPU moderna riesce a eseguire almeno un' operazione elementare su un numero a doppia precisione per ciclo di clock. Ciò significa che se un processore ha una «velocità» di 3GHz esso in teoria può eseguire almeno $3 \cdot 10^9$ operazioni in virgola mobile al secondo. La velocità di un computer può essere espressa in FLOPS ossia il numero di operazioni in virgola mobile per secondo. Quindi se ho un processore a 3 GHz mi aspetterei almeno 3 GFLOPS. Le situazione però non è sempre così favorevole, molto spesso i nostri calcoli richiedono il trasferimento di dati dalla RAM alla CPU. Si pensi ad esempio alla moltiplicazione di due matrici quadrate grandi. La velocità di tale trasferimento è limitata dalla velocità o larghezza di banda (**bandwidth)** del memory bus anch'essa è espressa in Hz ed una velocità , ad esempio, di 1.6 GHz significa che in un secondo vengono trasferiti $1.6 \cdot 10^9$ bytes, quindi in un secondo riusciamo a trasferire da memoria a CPU solo $1.6 \cdot 10^9/8 = 0.2 \cdot 10^9$ numeri in doppia precisione inoltre va considerato che nell'accedere a zone di RAM non attigue si verifica una latenza (tempo di accesso) dell'ordine dei ~ 10 ns. Questo determinerebbe un crollo notevole delle capacità di calcolo del nostro computer. Per rimediare a questo problema la CPU contiene una speciale memoria RAM , la cache memory, il cui contenuto è immediatamente utilizzabile (tipicamente in 1 ciclo di *clock)* dal resto della CPU. Quindi i nostri programmi per operare al meglio devono trasferire blocchi di dati dalla RAM alla cache per venire poi elaborati.

Da circa una decina d'anni ha preso piede la tecnologia a **molti core**: il singolo *microprocessore* contiene più di una CPU (*core*) , tipicamente da 2 a 8 per i personal computers/laptops, e fino a 64 o più per sistemi di calcolo avanzato. Come si vedi in Fig. 3 abbiamo una cache memory comune a tutti i cores più una o due cache memory interne a ciascun core. Lo sviluppo di tale tecnologia rende vantaggioso l'utilizzo di tecniche di programmazione i*n parallelo* in cui un solo programma è elaborato da più cores contemporaneamente.

Il calcolo parallelo è ormai il paradigma su si basano tutte le machine di calcolo ad alte prestazioni (*high performance computing*) . In esse più *nodi* ciascuno contente uno o più microprocessori a molti core e della memoria RAM sono interconnessi tra di loro.

Recentemente sono state sviluppate schede di calcolo esterne al processore derivate dalle schede grafiche. Tali schede sono denominate GP-GPU (general purpose graphical processing unit) o semplicemente GPU. La CPU scambia dati con la GPU attraverso una connessione (databus) ,solitamente di tipo PCI, che permette una larghezza di banda minore del memory bus. La GPU è strutturata come una CPU con moltissimi cores di calcolo (fino a qualche migliaia) che hanno accesso veloce ad una memoria RAM propria della scheda. La frequenza di clock dei cores della GPU non è, tipicamente, elevata come come quelli della CPU. Pertanto i programmi devono essere scritti in maniera da trasferire sulla GPU le parte che sono ben parallelizzabile.

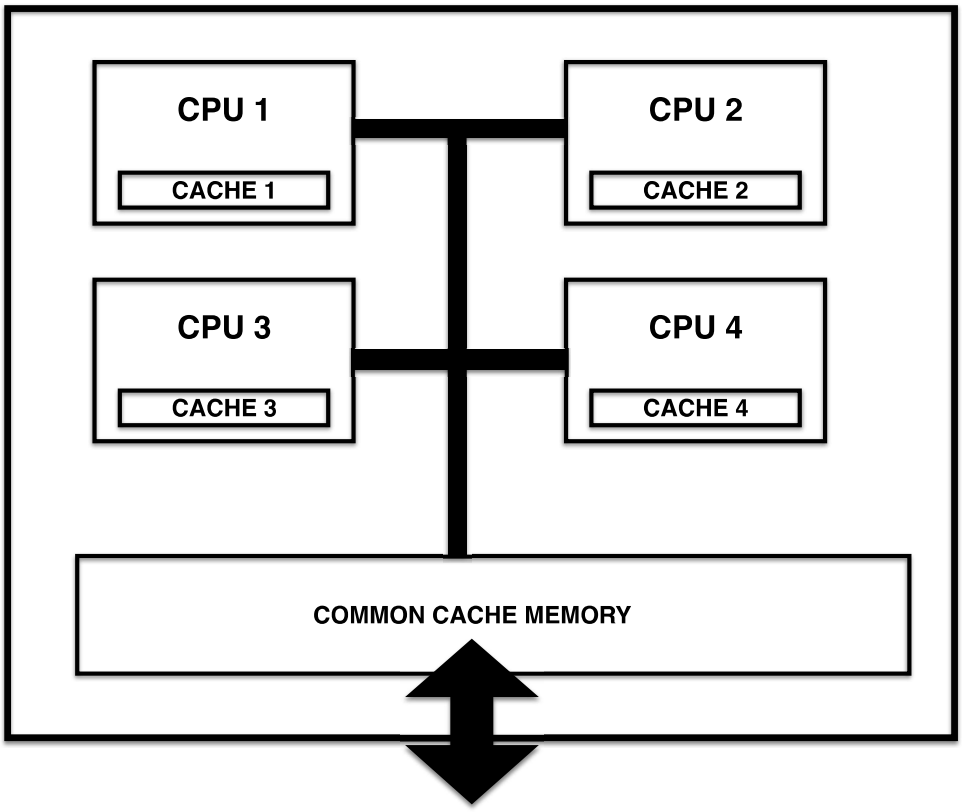

Figura 3: Schema di un microprocessore a 4 cores

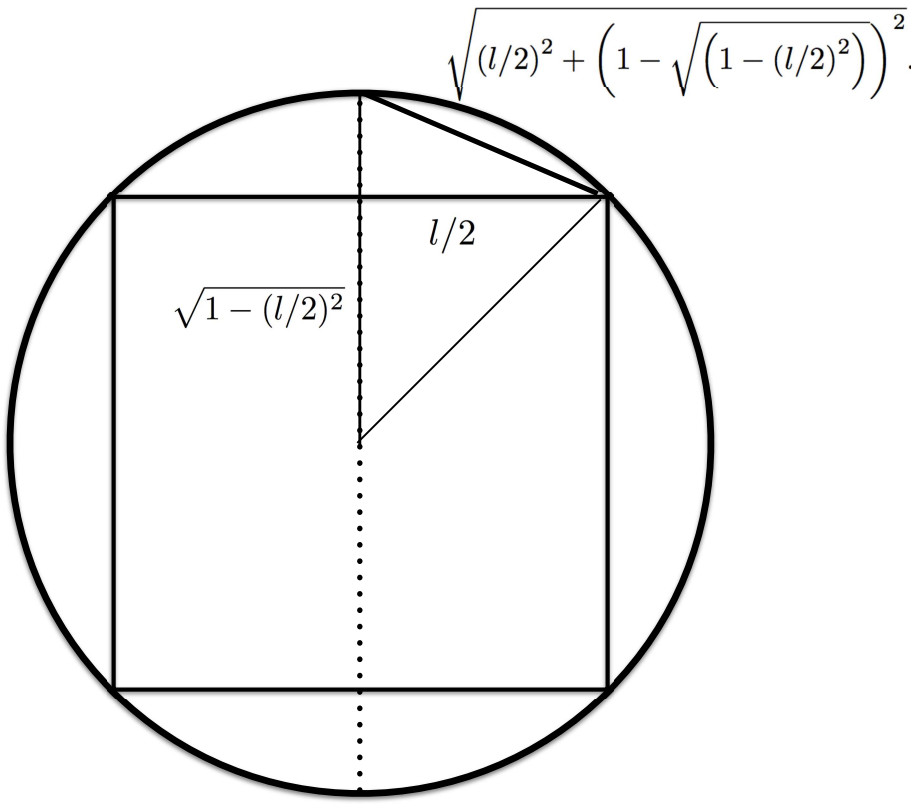

Figura 4: Calcolo del lato di un poligono regolare di numero di lati doppio al poligono regolare di lato l

Attualmente in Italia il sistema di calcolo più potente disponibile per la ricerca, Marconi-100 (www.cineca.it) consta di 980 nodi interconnessi tra di loro ciascuno con 32 cores, 256 GB di mermoria RAM e 4 schede GP-GPU NVIDIA A100.

2.3 Operazioni in virgola mobile: possibili problemi

Quando si lavora con numeri in virgola mobile possono presentarsi problemi (**roundoff** errors) dovuti al numero limitato di cifre significative disponibili. Il fatto di avere un limite di cifre significative implica che i nostri numeri sono definiti entro un *range* di valori possibili.

Quindi la sottrazione di due numeri molto vicini tra di loro (NB l'esponente è lo stesso) potrebbe essere calcolata con un notevole errore relativo.

Inoltre è importante considerare l'ordine con cui si eseguano le operazione matematiche, supponiamo di essere in doppia precisione e di calcolare:

$1) a = 1.0 + 1.0 \cdot 10^{-18} = 1.0$

$2) b = a - 1.0 = 0.0$

mentre se scambiamo l'ordine delle somme otteniamo:

$1) a = 1.0 - 1.0 = 0.0$

$2) b = a + 1.0 \cdot 10^{-18} = 1.0 \cdot 10^{-18}$

In taluni casi tali differenze possono portare ad esiti catastrofici. Tale è il caso del calcolo del valore di pigreco π come limite del perimetro di poligoni regolari inscritti in una circonferenza di raggio unitario.

Se la misura del lato di un poligono regolare di n lati è l, il teorema di Pitagora da per il lato l' del poligono di $2n$ lati: $l' = \sqrt{(l/2)^2 + \left(1 - \sqrt{\left(1 - (l/2)^2\right)}\right)^2}$. Vediamo cosa succede se implementiamo tale algoritmo usando:

a) formula CIOFECA $l' = \sqrt{2 - 2\sqrt{\left(1 - (l/2)^2\right)}}$ (dove abbiamo espanso il quadrato dentro la radice quadrata).

oppure

b)formula BUONA $l' = \sqrt{(l/2)^2 + \left(1 - \sqrt{\left(1 - (l/2)^2\right)}\right)^2}$

Usando la formula vediamo che per l molto piccolo la formula a) dà $l' = 0$ mentre la formula b) dà $l' = l/2$. Pertanto usando b) il calcolo del pigreco convergerà ad un valore vicino nel limite del roundoff error al valore esatto mentre a) darà risultati totalmente instabili. Ecco i risultati del metodo ciofeca in singola precisione:

```
1 3.06147
2 3.121444463729858398437 5
3 3.13654613494873046875
4 3.140333414077758789062 5
5 3.141285657882690429687 5
6 3.141518831253051757812 5
7 3.141207933425903320312 5
8 3.14245128631591796875
9 3.14245128631591796875
10 3.162277698516845703125
11 3.162277698516845703125
12 2.828427076339721679687 5
13 0
14 0
```

Questi invece sono i risultati del metodo buono ,

```
1 3.06146745892071869832307129399850965
2 3.12144515225805285751903284108266234
3 3.13654849054593976021010348631534725
4 3.14033115695475339990139218571130186
5 3.14127725093277332391039635695051402
6 3.14151380114430134327108135039452463
7 3.14157294036709178186583812930621207
8 3.14158772527716001476960627769585699
9 3.14159142151120018837673342204652727
10 3.14159234557011801669546002813149244
11 3.14159257658487289788240559573750943
12 3.14159263433856317249137646285817027
13 3.14159264877698607421052656718529761
14 3.14159265238659157759570916823577136
15 3.14159265328899328650891220604535192
16 3.14159265351459371373721296549774706
17 3.14159265357099393156659061787649989
18 3.14159265358509420806853995600249618
19 3.14159265358861894412711990298703313
20 3.14159265358950046120867227728012949
21 3.14159265358972117354596775840036571
22 3.14159265358977624060798916616477072
23 3.14159265358979000737349451810587198
24 3.14159265358979356008717331860680133
25 3.14159265358979444826559301873203367
26 3.14159265358979489235480286879464984
27 3.14159265358979489235480286879464984
```

dove le prime 15 cifre significative sono esatte: 3.14159265358979 come ci possiamo aspettare per un numero in doppia precisione. Nelle pagine moodle del corso trovate il semplice codice in C++ per ottenere tali risultati.

Nel resto del corso ci soffermeremo su tali problemi solo nel caso risultino rilevanti per il funzionamento degli algoritmi che svilupperemo.

2.4 Uso della memoria con C++: Stack e Heap

Durante l'esecuzione di un programma i dati vengono immagazzinati (allocati) nella memoria RAM. Distinguiamo due diversi tipi di allocazione: l'allocazione automatica nello stack e l'allocazione dinamica nello heap. Lo stack è un'area della RAM che viene assegnata ai dati temporanei allocatati in maniera automatica negli scopes di C/C++. Ad esempio:

```
{\\inizio scope
   double v;
   double a[100];
```

```
    double m[10,10];
}\\fine scope
```

La variabile m, l'array a e la matrice m vengono creati (allocati) riservando una zona dello stack. Quando lo scope finisce tale zona di memoria viene liberata automaticamente e i dati non sono più disponibili. Lo stack è basato sul principio di una pila (stack) di tipo LIFO (last in first out). Infatti man mano che si entra in scopes più interni la pila cresce mentre quando si esce dagli scopes la rispettiva parte di pila (ossia di memoria) viene rilasciata. Pertanto l'allocazione automatica nello stack risulta veloce anche se i dati sono disponibili solo all'interno dello scope in cui vengono definiti. La quantità di memoria massima riservata allo stack è determinata dal sistema operativo e di solito è relativamente limitata. Nei sistemi linux il comando *ulimit* permette di conoscere e in taluni casi di modificare la dimensione massima dello stack.

Lo heap, invece, è una zona della memoria riservata all'allocazione dinamica dei dati. I dati allocati in tale maniera sono disponibili anche al di fuori dello scope fintanto vengono liberati esplicitamente dal programma. In C++ possiamo allocare con *new* e deallocare con *delete*. Alternativamente possiamo usare come in C le funzioni *malloc* e *free*.

```
    double* v;
    double* a;

    v = (double*) new double;
    a = (double*) new double[100];

    delete [] v;
    delete [] a;
```

La dimensione massima dello heap è pari normalmente a tutta la RAM disponibile pertanto l'allocazione dinamica è consigliata per gestire grandi volumi di dati.

2.5 Array e Matrici in C++

Il C/C++ supporta l'allocazione automatica di vettori, chiamati arrays, e matrici. Basta scrivere semplicemente:

```
{\\inizio scope
    double v;
    double a[100];
    double m[10][10];
}\\fine scope
```

Inoltre dallo standard C++11 vengono supportati anche array e matrici di dimensione variabile ossia determinata durante l'esecuzione del programma:

```
{\\inizio scope
    int n;

    //qui viene determinato n

    double a[n];
    double m[n][n];
}\\fine scope
```

Purtroppo l'allocazione dinamica è limitata agli arrays:

```
{\\inizio scope
    int n;
    double* v;

    //qui viene determinato n

    v = (double*) new double[n];

    delete [] v;
}\\fine scope
```

Per allocare delle matrici riservando un unico blocco di RAM possiamo ricorrere alla strategia seguente

```
{\\inizio scope
   int n;
   double* bmat;
   double** b;

   //qui viene determinato n

   bmat = (double*) new double[n*m];
   b = (double**) new double*[n]

   for (int i=0; i<n; i++) {
        b[i]=&bmat[m*i];
   }


   delete [] v;
   delete [] mat;
   delete [] m;
}\\fine scope
```

In questo modo m si comporta esattamente come la matrice *double b[n][m]*. Inoltre ricordiamo che a differenza del Fortran in C/C++ gli le matrici sono immagazzinate in memoria riga dopo riga.

2.6 Importanza di un corretto uso della memoria

Mostriamo ora con un esempio quanto il modo in cui il nostro codice usa la memoria ed il trasferimento memoria/cache/-registri/fpu può influenzare il tempo di esecuzione. Consideriamo la moltiplicazione di due matrici quadrate di ordine (o dimensione) N, A e B:

$$C_{ij} = \sum_{k=1,N} A_{ik} B_{kj} \tag{6}$$

quindi per calcolare $C = A \cdot B$ dobbiamo compiere N^3 somme e addizioni per cui su un solo core ci aspettiamo un tempo di esecuzione paragonabile a $T_{ideal} = 2N^3/(N_{op} \cdot \text{clock})$, con clock la velocità di clock in cicli per secondo e N_{op} numero di operazioni in virgola mobile per ciclo di clock.

In questo corso gli esempi saranno dati in C/C++. Dato che N può essere un numero grande, le matrici A e B dovranno essere allocate in memoria (*heap*) sfruttando l'allocazione dinamica. In C però possiamo definire direttamente matrici sono nel caso di allocazione nello *stack*. Tipo:

```
double A[10][10];
```

per definire matrici di dimensioni arbitrarie in C possiamo utilizzare il seguente schema

```
....
long iN;
double *dA,*dB,*dC,*dD;
double **dAM,**dBM,**dCM;
.....
dA=(double*) malloc(iN*iN*sizeof(double));
dB=(double*) malloc(iN*iN*sizeof(double));
dC=(double*) malloc(iN*iN*sizeof(double));

dAM=(double**) malloc(iN*sizeof(double*));
dBM=(double**) malloc(iN*sizeof(double*));
dCM=(double**) malloc(iN*sizeof(double*));
........
 for(iI=0;iI<iN;iI++){
     dAM[iI]=&dA[iN*iI];
     dBM[iI]=&dB[iN*iI];
     dCM[iI]=&dC[iN*iI];
 }
........
```

ora possiamo scrivere l'operazione di moltiplicazione tra matrici come:

```
for(iI=0;iI<iN;iI++){
    for(iJ=0;iJ<iN;iJ++){
        for(iK=0;iK<iN;iK++){
            dCM[iI][iJ]+=dAM[iI][iK]*dBM[iK][iJ];
        }
    }
 }
```

Consideriamo $N = 1000$ e usiamo un MacBook Pro del 2014 con microprocessore Intel i5 (sandy bridge) che ha un clock=2.6 GHz e può eseguire fino a 8 somme e moltiplicazioni in virgola mobile per ciclo di clock. Ci aspettiamo un tempo di esecuzione ideale di $T_{ideale} = 2 \cdot 1000^3/(8 \cdot 2.6 \cdot 10^9) = 0.096$ s. Il codice sorgente è riportato in appendice. Se compiliamo senza attivare le ottimizzazioni:

```
g++ -lcblas   main.cpp
```

registriamo un tempo $T_{naive} = 6.8$ s ossia il nostro codice sta girando 68 volte più lentamente della velocità ideale. Ci accorgiamo che nel ciclo *for* più interno, quello su iK, compiamo l'operazione

```
dCM[iI][iJ]+=dAM[iI][iK]*dBM[iK][iJ];
```

quindi per iK e IK+1 andiamo ad usare due elementi di dBM che non sono contigui in memoria. Per usare solo elementi contigui basta considerare al posto di dBM la sua trasposta:

```
for(iI=0;iI<iN;iI++){
     for(iJ=0;iJ<iN;iJ++){
          dDM[iI][iJ]=dBM[iJ][iI];
     }
}
for(iI=0;iI<iN;iI++){
      for(iJ=0;iJ<iN;iJ++){
           for(iK=0;iK<iN;iK++){
                dCM[iI][iJ]+=dAM[iI][iK]*dDM[iJ][iK];
           }
      }
}
```

dove il calcolo della matrice trasposta dDM richiede N^2 operazioni di assegnazione. Ora per la moltiplicazione matriciale occorre un tempo $T_{trasposta} = 3.7$ s. Possiamo ora tentare di utilizzare al meglio le risorse del processore attivando una compilazione più ottimizzata:

```
g++ -lcblas   -Os main.cpp
```

questo consente di ridurre sia il tempo della strategia iniziale $T^*_{naive} = 2.8$ s sia di quella tramite trasposta $T^*_{trasposta} = 1.09$ s. Ci accorgiamo che ciascuna delle nostre matrici occupa uno spazio in memoria di $8 \cdot 1000^2 = 7.6$ MB più grande della cache memory (L3 di terzo livello). Per cercare di utilizzare al meglio la cache memory andiamo a scomporre l'operazione $C = A \cdot B$ nella moltiplicazione di matrici di dimensione ridotta:

```
for(iII=0;iII<iN;iII+=iNblock){
     for(iJJ=0;iJJ<iN;iJJ+=iNblock){
       for(iKK=0;iKK<iN;iKK+=iNblock){
                          //copy data
              for(iI=0;iI<iNblock;iI++){
                   pd1=&dAbl[iI*iNblock];
                   pd2=&dA[(iII+iI)*iN+iKK];
                        for(iJ=0;iJ<iNblock;iJ++){
                              pd1[iJ]=pd2[iJ];
                        }
               }
//of B take the transpose
               for(iI=0;iI<iNblock;iI++){
                   pd1=&dBbl[iI*iNblock];
                   pd2=&dB[(iKK+iI)*iN+iJJ];
                   for(iJ=0;iJ<iNblock;iJ++){
```

```
                        pd1[iJ]=pd2[iJ];
                    }
                }
                for(iI=0;iI<iNblock;iI++){
                    for(iJ=0;iJ<iNblock;iJ++){
                        dDbl[iI*iNblock+iJ]=dBbl[iJ*iNblock+iI];
                    }
                }
                for(iI=0;iI<iNblock*iNblock;iI++){
                    dCbl[iI]=0.;
                }
                for(iI=0;iI<iNblock;iI++){
                    for(iJ=0;iJ<iNblock;iJ++){
                        pd1=&dAbl[iI*iNblock];
                        pd2=&dDbl[iJ*iNblock];
                        pd3=&dCbl[iI*iNblock+iJ];
                        for(iK=0;iK<iNblock;iK++){
                            *pd3+=pd1[iK]*pd2[iK];
                        }
                    }
                }
            //sum up on global matrix
                for(iI=0;iI<iNblock;iI++){
                    pd1=&dC[(iII+iI)*iN+(iJJ)];
                    pd2=&dCbl[iI*iNblock];
                    for(iJ=0;iJ<iNblock;iJ++){
                        pd1[iJ]+=pd2[iJ];
                    }
                }
            }
        }
}
```

dove iNblock è la dimensione N_b della matrice ridotte. Il costo del calcolo dipende dalla scelta di N_b. Per $N_b = 25$ troviamo $T^*_{block} = 0.86$ s. Abbiamo ottenuto una diminuzione del tempo di esecuzione di circa il 20% ma siamo ancora lontani dal limite ideale. Inoltre il nostro codice è diventato molto più complesso.

Per sfruttare al meglio le capacità dei processori sono state create delle librerie matematiche standardizzate quali BLAS e LAPACK (www.netlib.org) che contengono routines per risolvere problemi di algebra lineare. La libreria BLAS provvede routines per prodotti e somme di matrici e vettori. LAPACK invece fornisce routines per operazioni quali la soluzione del problema agli autovalori e la diagonalizzazione di matrici. Queste librerie sono scritte in FORTRAN e possono contenere parti in scritte in ASSEMBLY. Per evitare le noie dovute al chiamare subroutines Fortran in un codice scritto in C/C++ possiamo usare le librerie CBLAS e CLAPACK che provvedono con delle interfacce. Proviamo allora ad usare la routine CBLAS: cblas_dgemm che calcola il il prodotto di due matrici. Essa usa gli stessi parametri della routine Fortran DGEMM descritti in http://www.netlib.org/lapack/explore-html/d7/d2b/dgemm_8f.html.Notiamo che usare cblas_dgemm richiede poca scrittura di codice:

```
cblas_dgemm(  CblasRowMajor,CblasNoTrans,CblasNoTrans,  iN,iN,iN,1.0,dA,iN,dB,iN,1.0,dC,iN);
```

inoltre non è richiesto alcun livello particolare di ottimizzazione durante la compilazione. Usando quest'ultima strategia troviamo $T_{dgemm} = 0.18$ s. Ben compatibile con la nostra stima per il limite ideale. I codici in C++ sono riportati sul moodle.

2.7 Moltiplicazione di matrici in python

Analizziamo ora il problema della moltiplicazione di due matrici usando python. Per definire delle matrici useremo la libreria *numpy*. Proviamo moltiplicare due matrici quadrate di ordine n nella seguente semplice maniera:

```
import numpy as np
import time
```

```
def main():
    n=input("Ordine della matrice :\n")
    n=int(n)

    ma=np.ones((n,n),dtype=np.float64)


    mb=np.ones((n,n),dtype=np.float64)

    mc=np.zeros((n,n),dtype=np.float64)

    print("Ora moltiplico")

    start_time = time.perf_counter()
    for i in range(n):
        for j in range (n):
            for k in range (n):
                mc[i,j]+=ma[i,k]*mb[k,j]

    end_time = time.perf_counter()

    elapsed_time = end_time - start_time

    print("Elapsed time: ", elapsed_time)


    print(str(mc[0,0])+ " "+str(mc[1,0])+"   "+ str(mc[2,0]))

if __name__ == "__main__":
    main()
```

Ho utilizzato il metodo *np.unos* per creare le matrici di tipo reale a doppia precisione (*dtype=np.float64*). Facciamo delle prove con un laptop che monta un processore Intel Core (11th Gen Intel(R) Core(TM) i7-1165G7) dotato di 4 cores. Ognuno di questi può svolgere fino a 10 operazioni in virgola mobile per ciclo di cloack. Per cui mi aspetto una performance teorica in doppia precisione di $4*8*4.8 = 192$ GFLOPS. Dove la velocità di cloack è stata posta a 4.8 GHz (valore di boost). Per matrici di ordine n ci mette 52.0 s invece dei, considerando $2*n^3$ operazioni in virgola mobile, $2*500^3/(192*10^9) = 0.0014$ s del tempo ideale. Questa grande differenza è dovuta sia al fatto che python è un linguaggio interpretato e non compilato, sia al cattivo utilizzo della memoria fatto dall'algoritmo usato per la moltiplicazione. Una semplice soluzione è usare il metodo *np.matmul* di *numpy* per moltiplicare due matrici. Tale metodo (nella maggior parte delle implementazioni) utilizza le librerie BLAS. Inoltre permetter di girare su più cori contemporaneamente tramite il protocollo OPENMP.

```
    import numpy as np
import time


def main():
    n=input("Ordine della matrice :\n")
    n=int(n)

    ma=np.ones((n,n),dtype=np.float64)
    mb=np.ones((n,n),dtype=np.float64)
    print("Ora moltiplico")

    start_time = time.perf_counter()
    mc=np.matmul(ma,mb)
    end_time = time.perf_counter()

    elapsed_time = end_time - start_time
```

```
print(str(mc[0,0])+ " "+str(mc[1,0])+"   "+ str(mc[2,0]))
print("Elapsed time: ", elapsed_time)
```

Ora ci vogliono solo 0.017 s. Proviamo ora un test più significativo scegliendo $n = 10000$. Ora *np.matmul* ci mette 12.5 s contro i $2(10^4)^3/(192 * 10^9) = 10.5$ s del tempo teorico ideale. Per cambiare il numero di *threads* impiegati, uso il comando linux, valido per la shell BASH :

```
export OMP_NUM_THREADS=4
```

in cui abbiamo impostato 4 threads corrispondenti a 4 cores fisici del processore.