VMC i Python og C++

Magnar K. Bugge

28. juli 2008

I forbindelse med min sommerjobb i CSE-prosjektet, har jeg implementert forskjellige varianter av min VMC-kode fra Prosjekt 4 i fjor for å utforske mulighetene for bruk av Python i FYS3150. Alle implementeringene bruker den enkleste prøvebølgefunksjonen for Helium

$$\psi_{\mathrm{T}} = \exp\left(-\alpha\left(r_1 + r_2\right)\right)$$

med analytisk uttrykk for den lokale energien. Alle parallelle kjøringer er gjort på Titan, mens de serielle kjøringene er gjort på `manybody.uio.no`. De forskjellige implementeringene er ordnet under et directory `VMC`. Python-C++-versjonen med pypar finnes for eksempel under `VMC/VMC-pypar-c++/`.

Implementeringene

C++

C++-implementeringen er tatt rett fra fjorårets prosjekt. Denne implementeringen brukte MPI, men jeg laget også en versjon hvor jeg kommenterte ut parallelliseringen for å kunne sammenlikne med ikke-parallelle versjoner av de andre variantene. Programmet mitt inneholdt en bit der en optimal steglengde ble funnet for hver alpha. Dette ble gjort ved en rekke kjøringer med få MC-sykler, og denne delen av koden er ikke parallellisert (med mange MC-sykler blir denne delen av programmet ubetydelig hva angår tid).

Python-C++

Siden Python er et interpretert språk, er det fleksibelt, lettskrevet og lettlest. C++ derimot, som er et kompilert språk, er mer tungvint å skrive, men mye raskere. En måte å prøve å ytnytte det beste fra begge verdener er å skrive et Python-program som bruker kompilert C++-kode, altså skrive en C++-extension til et Python-program. Ideen er da å flytte det mest tidkrevende

til C++ for å få et raskere program. I mitt VMC-eksempel, er det løkka over MC-syklene som er det tidkrevende, så denne biten ville jeg ha i C++.

Å kombinere Python og C++ innebærer wrapper-kode, som er veldig tungvint å skrive. Heldigvis finnes programvare som gjør dette for deg. Jeg brukte et program som heter SWIG. Det viste seg å fungere veldig greit til dette formålet. Fremgangsmåten er som følger. Du skriver en C++-fil med den funksjonen du ønsker å kunne kalle fra Python, og alle funksjoner denne igjen trenger å kunne kalle (alt trenger ikke være samlet i en fil, jeg brukte for eksempel `ran0` fra `lib.cpp` på vanlig måte). Deretter skriver du en SWIG interface-fil, der du har med filnavnet til kildekoden og deklarerer funksjoner og klasser som skal brukes fra Python.

MC-funksjonen skal returnere både en sum, en kvadratsum, antall sykler som ga statistisk bidrag (trekker fra termalisering og singulariteter) og antall aksepterte forflytninger. I tillegg må variabelen `idum` til randomgeneratoren returneres for at verdien av denne skal oppdateres. Siden C++ ikke har støtte for mer enn én returverdi, laget jeg en klasse data, slik at et data-objekt kunne holde alle disse variablene og returneres til Python (det skal også være mulig å gjøre "call by reference" fra Python til C++, som er en annen løsning).

Jeg brukte altså den samme C++-koden som før, bare med en liten ekstra klasse. Denne koden lå i fila `MC.cpp`. SWIG interface-fila mi, `MC.i`, så slik ut:

```
%module MC
%{
#include "MC.cpp"
%}

class data {

public:
  double sum;
  double squaresum;
  int N;
  int accepted;
  long idum;
};

long seed();

data runMC(int MCcycles, double delta, long idum, double alpha);
```

Funksjonen `seed` returnerer bare `time(NULL)` som brukes som initiell `idum`-verdi. Legg merke til at jeg kun trenger å inkludere (`#include`) selve kodefila.

Biblioteket `lib.cpp` inkluderes i et `#include`-statement i `MC.cpp`. Kompilering gjorde jeg med følgende linjer (som jeg hadde i et script):

```
swig -python -c++ MC.i
c++ -fPIC -c MC_wrap.cxx -I/usr/include/python2.4
c++ -fPIC -shared MC_wrap.o -o _MC.so
```

Flagget `-fPIC` er nødvendig for 64-bit-maskiner, og `/usr/include/python2.4` er området der `Python.h` og andre nødvendige filer finnes. Etter at kompilering var gjort, kunne jeg aksessere funksjonen og lagre resultatene i en variabel `x` med følgende Python-linjer:

```
import MC
x = MC.runMC(MCcycles,delta,idum,alpha)
```

Nå er de statistiske dataene fra kjøringen tilgjengelige ved `x.sum`, `x.squaresum` osv.

Python

Videre laget jeg en ren Python-implementering. Jeg brukte da den Python-koden jeg allerede hadde skrevet, og implementerte funksjonen `runMC` rett i Python. Jeg skrev koden svært lik C++-koden. Jeg brukte NumPy-arrayer, men så vidt jeg kunne se var det ikke noe særlig muligheter for å utnytte array-funksjonalitet spesielt effektivt (NumPy kan jo være veldig raskt når man gir "kollektive" kommandoer på store arrayer). Jeg forsøkte å bytte ut løkka som lagde `R_trial` med en enkelt NumPy-linje, men dette så ikke ut til å påvirke farten (antageligvis siden det er en såpass liten array). Jeg kunne også brukt `numpy.linalg.norm` til å regne ut lengden av noen vektorer, men tidligere erfaring har vist at dette går tregere (for arrayer av denne størrelsen).

Parallellisering med Pypar

Jeg parallelliserte både den rene Python-versjonen og Python-C++-versjonen med pakken Pypar, som er installert på Titan. Denne pakken er et frontend for MPI, og er i Pythons ånd veldig fleksibel og lett å bruke. Du trenger bare spesifisere hva du vil sende, og hvor du vil sende det, så ordner Pypar alt med spesifikasjon av datatyper osv. Dette fører selvsagt med seg litt tidsbruk, men dersom det ikke skal sendes så ofte har ikke dette så mye å si. Dessuten kan man spesifisere datatyper osv for at sending skal gå raskere, men da forsvinner jo noe av vitsen med Pypar (slik jeg ser det).

Følgende kode viser hvordan jeg sender statistikken fra slavene til master (som et tuppel), og legger denne til master sin statistikk:

```
if myid != 0:
    pypar.send((x.sum,x.squaresum,x.N,x.accepted),destination=0)

if myid == 0:
    for i in xrange(1,nprocs):
        isum,isquaresum,iN,iaccepted = pypar.receive(source=i)
        x.sum += isum
        x.squaresum += isquaresum
        x.N += iN
        x.accepted += iaccepted
```

For å få Python-C++-versjonen til å virke med Pypar, måtte jeg spesifisere i jobbskriptet til Titan hvor MC-modulen kunne finnes (heldigvis fikk jeg mye god hjelp av Jon til Titan-relaterte ting), se `VMC/VMC-pypar-c++/pypar-swig-job.sge`. For å få kompilert på Titan, måtte jeg endre include-pathen, se `VMC/VMC-pypar-c++/compile`.

De implementeringene som ble brukt til å sammenlikne kjøretider for C++ med MPI og Python-C++ med Pypar, var av den typen som splitter opp hvert MC-integral på alle nodene. Jeg laget også en parallell versjon med en master-slave-struktur, der master deler ut jobber (integraler) så fort slavene blir ferdige. (Denne implementeringen er basert på Kyrres Prosjekt 4, FYS3150.) Denne varianten bør kunne gi bedre resultater, siden termaliserings-delen bare gjøres én gang for hvert integral. I tillegg bør denne varianten være raskere enn den andre i tilfeller der man har mange integraler som skal regnes (mange α-verdier) og belastningen på de forskjellige nodene fra andre brukere er varierende. Dette er fordi trege noder ikke vil bli tildelt like mye jobb som de raske når man benytter denne implementeringen. Ulempen (slik som koden er nå) er at en av nodene brukes bare til administrasjon, og ikke gjør noe regning. Denne implementeringen finnes under `VMC/VMC-pypar-c++v2/`.

Resultater/vurdering

Kjøretider for de forskjellige implementeringene med forskjellige antall sykler osv er vist i tabeller 1-4. Av tabeller 1 og 2 ser vi at ren Python ikke kan sammenliknes med C++ når det gjelder hastighet. Så vidt jeg kan se, betyr dette at ren Python er uaktuelt for beregninger av denne typen i FYS3150.

Videre ser vi at Python-C++-versjonen går praktisk talt like fort som C++-versjonen, siden all den tunge regningen gjøres i den kompilerte modulen. Spørsmålet er om det egentlig er noen vits å bruke Python til det ytterste laget av programmet. Dette er ikke utenkelig, særlig når det gjelder

Python	11 min 50 sec
Python med pypar	3 min 30 sec
Python extended med C++	8.5 sec
Python med pypar, extended med C++	5.7 sec
C++	8.0 sec
C++ med MPI	4.7 sec

Tabell 1: Kjøring med 10^6 MC-sykler, 10^4 MC-sykler for bestemmelse av steglengde, 13 α-verdier, 10 noder for parallelle kjøringer.

Python	103 min 50 sec
Python med pypar	15 min 40 sec
Python extended med C++	1 min 11 sec
Python med pypar, extended med C++	14.7 sec
C++	1 min 10 sec
C++ med MPI	7.3 sec

Tabell 2: Kjøring med 10^7 MC-sykler, 10^4 MC-sykler for bestemmelse av steglengde, 13 α-verdier, 10 noder for parallelle kjøringer.

de studentene som har hatt Python i INF1100. Slik jeg ser det, må den største fordelen med å wrappe C++ i Python være at man da kan gjøre parallelliseringen i Python. Parallellisering med Pypar er mye lettere enn å bruke MPI-funksjonene i C++ direkte. Da kan kanskje den lille terskelen med bruk av SWIG være verdt bryet. SWIG var jo egentlig overraskende lett å bruke, og dersom man får de riktige kompileringskommandoene og et fungerende jobbskript for Titan servert, skulle det ikke by på noen problemer.

Vi ser at selv om Python-C++-versjonen kjører praktisk talt like fort som den rene C++-versjonen, går Python-C++-versjonen noe tregere enn den rene C++-versjonen når det kjøres parallelt (se tabeller 3 og 4). Dette må ha å gjøre med at Pypar er litt tregt. Som tidligere nevnt finnes det måter å få Pypar raskere på, men da blir det ikke like lett å bruke den.

Python extended med C++	2 min 4.7 sec
Python med pypar, extended med C++	14.7 sec
C++	2 min 4.6 sec
C++ med MPI	11.5 sec

Tabell 3: Kjøring med 10^7 MC-sykler, 10^4 MC-sykler for bestemmelse av steglengde, 23 α-verdier, 20 noder for parallelle kjøringer.

Python extended med C++	20 min 29 sec
Python med pypar, extended med C++	1 min 36 sec
C++	20 min 28 sec
C++ med MPI	53 sec

Tabell 4: Kjøring med 10^8 MC-sykler, 10^4 MC-sykler for bestemmelse av steglengde, 23 α-verdier, 20 noder for parallelle kjøringer.