Software Projects: Examples from Nuclear Physics

Gustav Baardsen

Centre of Mathematics for Applications
University of Oslo, NO-0316 Oslo, Norway

January 28, 2013

Outline

Background: Infinite nuclear matter

Outline

Background: Infinite nuclear matter

Case I: Code optimization and parallelization

Outline

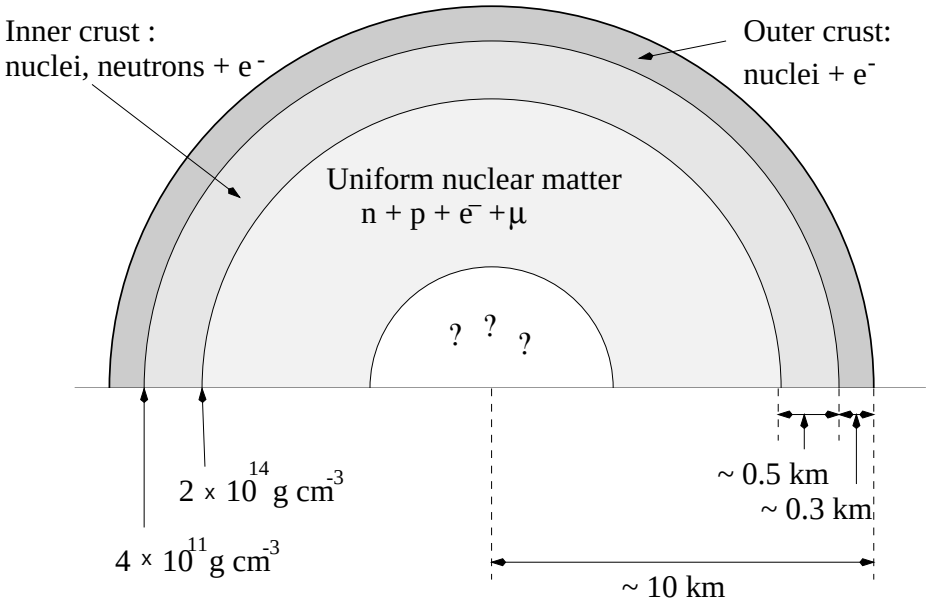
Background: Infinite nuclear matter

Case I: Code optimization and parallelization

Case II: Towards object-oriented Fortran code

Outline

We want to model a neutron star!



- A massive star burns up its fuel $\rightarrow$ Supernova explosion $\rightarrow$ A neutron star is formed
- Microscopic nuclear matter calculations give input to the neutron star Equation of State (EOS)

In mathematical terms...

We need to solve the eigenvalue problem

$$\hat{H}\Psi = E\Psi,$$

where E is the total energy and the Hamiltonian operator

$$\begin{aligned}\hat{H} &= \hat{T} + \hat{V} \\ &= -\frac{1}{2m}\sum_{i=1}^{A}\nabla_i^2 + \sum_{i<j}^{A}\hat{v}(\mathbf{r}_i, \mathbf{r}_j),\end{aligned}$$

$\hat{T}$ = kinetic energy operator,
$\hat{V}$ = two-particle interaction operator,

Outline

What needs to be coded

Expression for the correlation energy: $(\mathbf{k} = (k_x, k_y, k_z))$

$$\underbrace{\Delta E_{Correlation}}_{\text{The eigenvalue}} = \int d\mathbf{k}_i \int d\mathbf{k}_j \int d\mathbf{k}_a \int d\mathbf{k}_b \times \underbrace{\langle \mathbf{k}_i \mathbf{k}_j | v | \mathbf{k}_a \mathbf{k}_b \rangle}_{\substack{\text{Known,} \\ \text{not closed form}}} \underbrace{\langle \mathbf{k}_a \mathbf{k}_b | t | \mathbf{k}_i \mathbf{k}_j \rangle}_{\substack{\text{The unknown: A dense,} \\ \text{block-diagonal gigantic matrix}}} \quad (1)$$

$$\begin{aligned} 0 = & \langle \mathbf{k}_a \mathbf{k}_b | v | \mathbf{k}_i \mathbf{k}_j \rangle \\ & + \Delta\varepsilon(\mathbf{k}_i, \mathbf{k}_j, \mathbf{k}_a, \mathbf{k}_b) \langle \mathbf{k}_a \mathbf{k}_b | t | \mathbf{k}_i \mathbf{k}_j \rangle \\ & + \int d\mathbf{k}_c \int \mathbf{k}_d \langle \mathbf{k}_a \mathbf{k}_b | v | \mathbf{k}_c \mathbf{k}_d \rangle \langle \mathbf{k}_c \mathbf{k}_d | t | \mathbf{k}_i \mathbf{k}_j \rangle , \end{aligned} \quad (2)$$

- We use the Picard iteration method
- Conversion to quantum number basis (Fourier grid):
  $\mathbf{k} \rightarrow \{k(lS)\mathcal{J} m_{\mathcal{J}} M_T\}$

Challenges

The most naive implementation:

- Requires far too much RAM memory and CPU time

Code optimization I

Ways to optimize the code:

- Instead of loops, use matrix-matrix multiplications with optimized routines

Code optimization I

Ways to optimize the code:

- Instead of loops, use matrix-matrix multiplications with optimized routines
- Faster matrix operations with MKL BLAS than with standard BLAS

Code optimization I

Ways to optimize the code:

- Instead of loops, use matrix-matrix multiplications with optimized routines
- Faster matrix operations with MKL BLAS than with standard BLAS
- Utilize symmetries in physics (e.g. in total momentum and different quantum numbers):
  $\Rightarrow$ Matrices become block diagonal

Code optimization I

Ways to optimize the code:

- Instead of loops, use matrix-matrix multiplications with optimized routines
- Faster matrix operations with MKL BLAS than with standard BLAS
- Utilize symmetries in physics (e.g. in total momentum and different quantum numbers):
  $\Rightarrow$ Matrices become block diagonal
- I stored non-zero block diagonals into matrix lists

Code optimization II

- Limited cache memory $\Rightarrow$ it matters in which order matrix elements are obtained

Code optimization II

- Limited cache memory $\Rightarrow$ it matters in which order matrix elements are obtained
- Don't calculate the same thing many times:
  - Do as many matrix operations as possible before the Picard iterations
  - Analyze in what order it's wisest to write the loops

Code optimization II

- Limited cache memory $\Rightarrow$ it matters in which order matrix elements are obtained
- Don't calculate the same thing many times:
  - Do as many matrix operations as possible before the Picard iterations
  - Analyze in what order it's wisest to write the loops
- If tests consume much time, use sparingly

Parallelization

- The matrix lists were divided to different MPI processes
  $\Rightarrow$ available memory from many computing nodes
  $\Rightarrow$ many processors available

Parallelization

- The matrix lists were divided to different MPI processes
  $\Rightarrow$ available memory from many computing nodes
  $\Rightarrow$ many processors available
- Most of the CPU time was used to set up a big matrix list before the Picard iterations
  – The matrix list setup was parallelized on each node using OpenMP

Outline

From modules to objects

- The previous code was written in Fortran 90/95 using modules

From modules to objects

- The previous code was written in Fortran 90/95 using modules
- The Fortran 2003 standard fully supports object-oriented programming

From modules to objects

- The previous code was written in Fortran 90/95 using modules
- The Fortran 2003 standard fully supports object-oriented programming
- I have written a code for one-dimensional nuclear matter in (almost) object-oriented Fortran

- The problem was divided into classes

- The problem was divided into classes
- The code utilizes inheritance of an abstract class, as well as dynamic polymorphism (virtual methods)

Example of an abstract class

```
  TYPE, ABSTRACT :: CorrEnergy
     TYPE(Hamiltonian) :: hamilt                    ! Hamiltonian operator
     TYPE(SpPotential) :: spPot                     ! Single-particle potential
     INTEGER :: nkh,nkp,nkCM                        ! Number of momentum grid
                                                    !   points
     TYPE(QuadList) :: quads                        ! Quadratures for
                                                    !   correlatoion energy
     REAL(DP) :: eneCorr
   CONTAINS
     PROCEDURE(calcCorrEne), &
          DEFERRED :: calcCorrEnergy                ! Calculate correlation
                                                    !   energy
  END TYPE CorrEnergy

  ABSTRACT INTERFACE
     SUBROUTINE calcCorrEne(eneCorr)
       IMPORT CorrEnergy
       CLASS(CorrEnergy), TARGET, INTENT(INOUT) :: eneCorr
     END SUBROUTINE calcCorrEne
  END INTERFACE

CONTAINS

  SUBROUTINE printCorrEnergy(eneRef,eneCorrel)
    TYPE(RefEnergy), INTENT(IN) :: eneRef
    CLASS(CorrEnergy), POINTER, INTENT(IN) :: eneCorrel
    REAL(DP) :: eneTot
```

A routine that uses polymorphism

```
!
!     Routine to calculate the correlation energy in
!     second order perturbation theory (MBPT(2)).
!
SUBROUTINE calcPt2Energy(eneCorr)
  USE QuadratureMod
  CLASS(Pt2CorrEnergy), TARGET, INTENT(INOUT) :: eneCorr
  TYPE(Quadrature) :: relmomH,relmomP1,relmomP2,CMmom
  REAL(DP), ALLOCATABLE, DIMENSION(:) :: kpPoints,weightKp
  REAL(DP) :: PMax,interaction,kh,wkh,kp,wkp,P,wP
  REAL(DP) :: spPotK1,spPotK2,spPotK3,spPotK4
  REAL(DP) :: temp,khMax,kpMin,k1,k2,k3,k4
  INTEGER :: iP,ikh,ikp

  ALLOCATE(kpPoints(2*eneCorr%nkp),&
       weightKp(2*eneCorr%nkp))
  kpPoints = 0.d0
  weightKp = 0.d0

  !     Set up CM momentum quadrature
  PMax = 2.d0*eneCorr%hamilt%kF
  CMmom = gaussLegendre(-Pmax,Pmax,eneCorr%nkCM)

  !     Loop over CM momentum grid points
  temp = 0.d0
  DO iP=1, eneCorr%nkCM
     P = CMmom%xpoints(iP)
     wP = CMmom%weightx(iP)
```

Code structure when using objects

```
!
!     Set up single-particle potential mesh
!
SUBROUTINE setupSpPotMesh(spPot)
  TYPE(SpPotential), INTENT(INOUT) :: spPot
  TYPE(Quadrature) :: labmom
  REAL(DP) :: kmax,k,potential
  INTEGER :: ik

  kmax = spPot%hamilt%krelCutoff + spPot%hamilt%kF
  !     Set up a Gauss-Legendre quadrature
  labmom = gaussLegendre(-kmax,kmax,spPot%mesh%nMeshp)

  spPot%mesh%points = labmom%xpoints

  !     Loop over momentum grid points
  DO ik=1, spPot%mesh%nMeshp
     k = spPot%mesh%points(ik)
     !     Evaluate the single-particle potential
     !     with input momentum k
     potential = calcSpPotential(k,spPot)
     spPot%mesh%values(ik) = potential
  ENDDO


END SUBROUTINE setupSpPotMesh
```

Acknowledgements

- Professor Morten Hjorth-Jensen, UiO, main advisor
- Gaute Hagen, Oak Ridge National Laboratory, USA, advisor
- Andreas Ekström, Gustav Jansen, and Simen Kvaal for many useful discussions