Summary of the preliminary works on the parallelization of ZKA project

Igor Merkulow

merkulow@rz.rwth-aachen.de

Center for Computing and Communication
RWTH Aachen University

Agenda

- Introduction
  - What is ZKA
  - Benefits of parallelization
- First attempt
  - Difficulties
  - Possible solutions
- Second attempt: serial tuning and restructuring
  - Modularization
  - Error handling
  - Data I/O
- Summary and Perspectives

Introduction

- What is ZKA
  - Tool for simulation and computation of gear wheels
  - At the moment just simulating the rotation without pressure
  - Next version should include pressure/load
  - Computation amount would increase significantly
- Benefits of parallelization
  - Low-cost speeding up calculations
  - Using existing networks
  - More detailed or more memory-intensive computations

First attempt

- Straight-forward MPI parallelization
  - Distributing iterations over the processes, depending on their ID
  - Ignoring the output for the first tests
- This attempt was a dead end
  - Program seemed to fail before the calculations start
  - Assuming problems in the initialization part

First attempt - difficulties

- 2 “generations” of source code
  - New computation libraries
  - Old I/O and initialization code
- “Old” code is problematic with respect to parallelization
  - Many “common block” variables
  - Direct disc access leading to lock errors with shared files (on Windows)
  - Some routines seem to be faulty

First attempt - solutions

- First – we need to guarantee that every step before the parallelization is error-free
- Second – we should separate input, calculations and output
- Third – we should eliminate “common blocks” and pass all the data we need in a subroutine as parameter (and use interfaces to ensure their correctness)
- These considerations led to the shift of the working focus from parallelization to the tuning and restructuring of the serial program

Second attempt: serial tuning and restructuring

- The second part was divided in 3 sections
  - Implementing consistent error handling
  - Encapsulating file I/O
  - Program modularization

Serial Tuning – Error handling (1)

- "Old" code used different error handling approaches:
  - `SUBROUTINE allocFORMORG (allokieren)`
    - No error handling
  - `SUBROUTINE ABLPRUEF (FEHLER)`
    - Only a logical value
  - `SUBROUTINE ABLEIT (FKTNR, FEHLER, IFEHL)`
    - Logical and integer values
  - `SUBROUTINE CLOSEF (CLUNIT, CLSTAT, FHLNR)`
    - Only an integer value (sometimes used parallel to IFEHL but containing different values)
  - `SUBROUTINE EINLWK (ZEILE, IKV, IEFHL, IFA, IKEFHL)`
    - Two integers
  - `SUBROUTINE Get_File (file_id, file_handler, error)`
    - New message type, but optional and rarely used

Serial Tuning – Error handling (2)

- "New" approach:
  - "All-in-one" package containing all the necessary information
  - Additional information for parallelization
  - Should be passed to every subroutine (non-optional)

- Error data structure:

```
type exception
     logical              :: lOccurred=.false.
                                  ! just to know if an error occurred
     integer              :: iMPIProcessID=0
                                  ! process id (if necessary)
     integer              :: iMPINumProcs=1
                                  ! number of processes (if necessary)
     integer              :: iErrorID=0, iErrorID2=0
                                  ! identifier, second id if needed
     integer              :: iPriorityID=0    ! importance of this error
     character(char_30)   :: cFilename, cSummary, cFunction
                 ! source file name, short description, function/subroutine
     character(char_100)  :: cErrorText
                                 ! full description or additional information
     character(LENPAR), dimension(ANZPAR) :: cParam
                                  ! replacement for old PARAM-array
end type
```

Serial Tuning – Error handling (3)

- This results in consistent and recognizable calls and error handling:

```
Call mySub (param1, param2, error)
If (error%lOccurred) ...do something...
```

- Inside of a subroutine / function exception should be set:
  - Current version:

```
If (data == 0) then
  error%lOccurred = .true.
  error%iErrorID = 1
  error%iSummary = "data is equal to zero"
End if
```

  - Next version:

```
If (data == 0) call setException(error, ZERO_DATA_ERROR)
```

Serial Tuning – File I/O (1)

- "Old" disk access was made using hardcoded unit-numbers and file names, leading to
  - Complicated maintainability
  - File locking errors
  - Portability problems

- "New" approach encapsulates all file-related information in a "file data object" and manages all file accesses internally
  - Easy-to-use alias names
  - Configurable without recompilation
  - Defined interfaces
  - Less error-prone

Serial Tuning – File I/O (2)

- File data structure:

```
type :: file
    character (len=LEN_NAME)    :: cAlias   = ""
                    ! smth like "control_file", "log_file"
    character (len=LEN_FLAGS)   :: cPath    = ""
                    ! file path, e.g. c:/projects/myproject1
    character (len=LEN_NAME)    :: cName    = ""
                    ! real file name, e.g. mylogfile
    character (len=LEN_SHORT)   :: cExt     = ""    ! file extension
    integer                     :: iUnitnr  = 0
                    ! according unit number
    character (len=1)           :: cAccess  = ""
                    ! access mode: S = sequential, D = direct
    character (len=LEN_FLAGS)   :: cFlags   = ""
                    ! additional flags (position, etc)
end type
```

- Defined subroutines / functions

openFile / closeFile / *closeAll*
readStringFromFile / writeStringToFile
getFileStatus / *setPosition* / *createFile*
(Subroutines written in *italic* will be implemented in the next version)

Serial Tuning – File I/O (3)

- Configuration files:
  - Using existing INI-Files as a prototype
  - Improved functionality (chapters, variables)
  - Flexible use (configuration, messages, porting, …)
  - Easy handling, defined data passing interface

- Data structure (Associative array / Key-Value-Array)

```
type KeyValueArray
   character(len=char_30)   :: id        ! module name
   character(len=char_100)  :: files     ! files containing module data
   integer                  :: size      ! size (lines) of data
   character(len=char_100), dimension(:,:), allocatable    :: moddata
                ! array (1:size, 1:size) of strings (key-value-table)
end type
```

Serial Tuning – File I/O (4)

- Only main configuration file name is hardcoded in the program
- Other file names and locations can be edited without recompiling the program
- User configuration files can be used for testing with some different values without changing the main configuration
- By moving all literal constants to external files the program can be made portable and language-independent
- Changing to other file format (e.g. XML) require modifications in just one source file (m_Parser)

Serial Tuning – Modularization (1)

- The last topic leads us to the next main issue: Modularization
  - "New" routines are arranged into modules according to their functionality
  - Most of the "old" routines are not sorted and have no interfaces / modules
- Extending the modularization principle of the new routines, modules were combined into libraries
  - Reusable program component
  - None or strict defined dependencies
  - Allows separate development
  - Easy-to-use dependencies (version X.X of program A depends on version Y.Y of library B and Z.Z of library C)

Serial Tuning – Modularization (2)

- Splitting the source code into 5 libraries
  (any other arrangement also possible)

- Data type library (wzl_datatypes)
  - Defines new data types, e.g. String or DateTime
  - Includes corresponding functions, e.g. UpCase or StringSplit
  - Should include ToString- / Serialize-Routines for printing and sending with MPI

- System access library (wzl_system)
  - Implements routines for accessing file system, program configuration, logging and error handling
  - Depends on the data type library

Serial Tuning – Modularization (3)

- Mathematical library (wzl_math)
  - Contains all the mathematical routines used elsewhere
  - Should have no dependencies
- Kegelspan library (wzl_kegelspan)
  - Implements the entire Kegelspan calculation
  - Depends on mathematical and system libraries
- ZKA library (wzl_zako)
  - Implements the entire ZKA simulation
  - Depends on mathematical, system (and eventually Kegelspan) libraries

Summary

- Calculation routines have been transformed to modern F90 previously
- My work was primary focused on cleaning up the remaining parts of “old” code with regard to parallelization
- Following program modifications were suggested:
  - Splitting into libraries to give the program more structure
  - Unifying program configuration, making the program more flexible and portable
  - Encapsulating file system access, making the program more independent of underlying operating system and making the parallelization easier
  - Harmonizing error handling to get the most complete information about program status and to make development easier

Perspectives

- If the proposals mentioned above will be accepted and consequently implemented, next step can be the parallelization
- If it works, then we can move over to the optimization of parallel performance

Thank you

for your

attention!