MPI-Queue: Einfaches paralleles Rechnen mit MPI und Fortran

Martin Raifer

22. Januar 2011

MPI-Queue ist ein Grundgerüst und Programmbibliothek für eine möglichst einfache und sorgenfreie Implementation eines einfachen *parallelen* Warteschlangen-Algorithmus mit Hilfe von MPI in *modernem* Fortran.

Inhaltsverzeichnis

1 Funktionsweise 1

2 Ablauf 2

3 Verwendung 3

3.1 my_data_types 4

3.2 my_calculactions 5

4 Kompilieren und Ausführen 7

1 Funktionsweise

Das Programm basiert auf der Grundidee einer Warteschlange: Es gibt also eine Reihe von Aufgaben – *Tasks*, die unabhängig voneinander abgearbeitet werden können.

Jeder dieser Tasks besitzt einen für ihn spezifischen Datensatz, aus denen ein bestimmtes Ergebnis berechnet werden soll. Die Verteilung der Aufgaben auf die verschiedenen Prozesse übernimmt ein spezieller Prozess, der sogenannte *Master*-Prozess. Dieser empfängt außerdem die Ergebnisse, sobald diese fertig berechnet worden sind.

Die restlichen Prozesse (*Slaves*) warten auf einen Task, berechnen das Ergebnis und senden dieses an den Master zurück. Dies wird so lange wiederholt, bis vom Master ein Terminierungssignal empfangen wird.

2 Ablauf

Zur Erklärung des Programmablaufs sollen folgende zwei Diagramme stehen: Abbildung 1 zeigt den einfachsten Fall, nämlich die Ausführung mit nur einem Slave-Prozess. Deutlich zu sehen sind die verschiedenen Aufgaben der verschiedenen Prozesse: Der Master-Prozess übernimmt die Verwaltung und Verteilung der Tasks, die Slaves (hier nur ein einziger) führen die eigentlichen Berechnungen aus. Zur Speicherung bzw. Weiterverarbeitung der Resultate der Tasks wird im Master-Prozess für jedes empfangene Ergebnis eine Callback-Routine ausgeführt.

Abbildung 2 zeigt einen etwas komplexeren Fall, nämlich wenn zwei Slaves an der Abarbeitung der Warteschlange werken. Im Allgemeinen kann es vorkommen, dass die einzelnen Tasks nicht gleich viel Rechenzeit benötigen, die Ergebnisse also nicht in der gleichen Reihenfolge einlangen, wie die Tasks verschickt wurden.

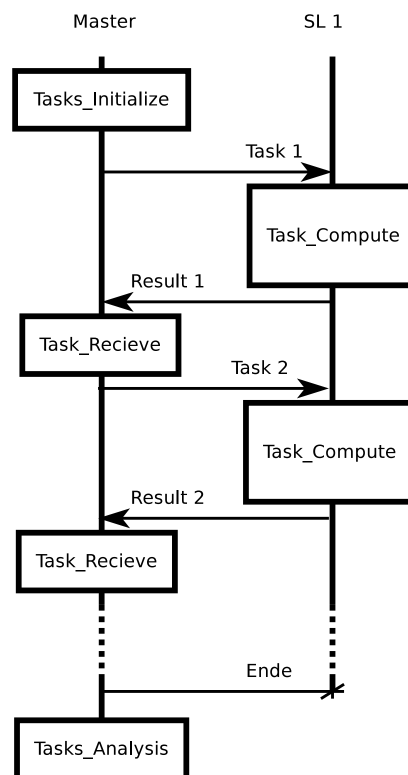

Abbildung 1 – Programmablauf bei zwei Prozessen.

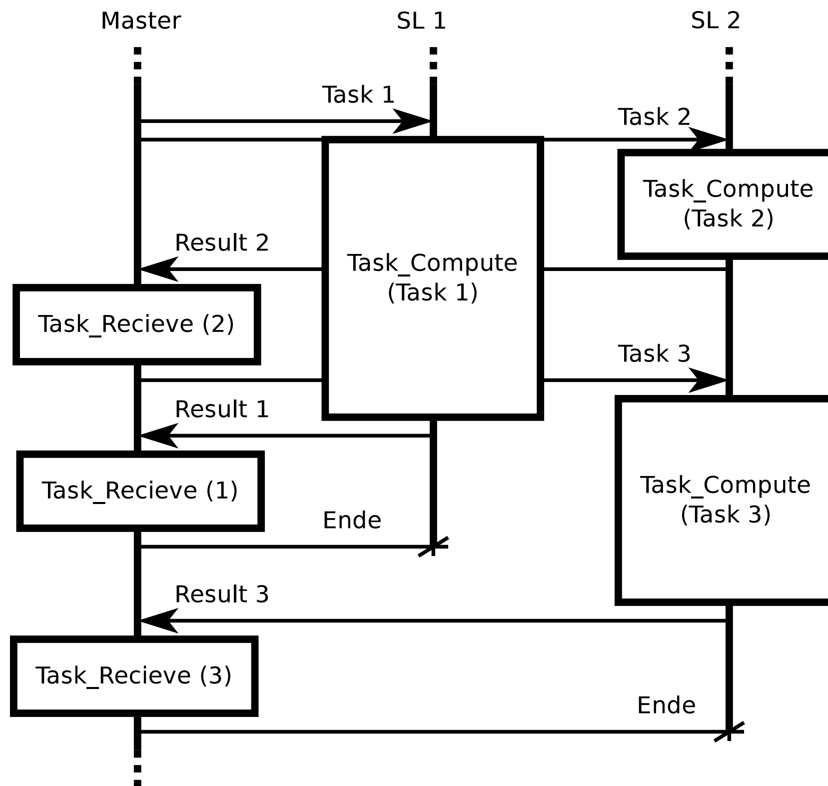

Abbildung 2 – Programmablauf bei drei Prozessen. Zu sehen ist, dass die Reihenfolge der empfangenen Ergebnisse nicht der Reihenfolge der gesendeten Tasks entsprechen muss.

3 Verwendung

Das Paket beinhaltet zwei Fortran-Module, die an die jeweilige konkrete Problemstellung angepasst werden müssen:

1. **module** my_data_types
2. **module** my_calculations

Ersteres beinhaltet Informationen zu den Datenstrukturen, die zwischen dem Master und den Slaves hin- und hergeschickt werden. Das Zweite definiert welche Tasks überhaupt ausgeführt werden sollen, welche Berechnungen für jeden Task gemacht werden müssen und was mit den empfangenen Ergebnissen geschehen soll.

Für diese beiden Module sind bereits zwei Vorlagen-Dateien bereitgestellt, die lediglich umbenannt und an die eigene Problemstellung angepasst werden müssen:

```
$ mv my_data_types.f90.template my_data_types.f90
$ mv my_calculations.f90.template my_calculactions.f90
```

3.1 my_data_types

Dieses Modul beschreibt die Datentypen, die für die Kommunikation zwischen dem Master- und den Slave-Prozessen verwendet werden. Diese drei Datentypen müssen definiert werden:

1. **type** global_data: Für globale Parameter, die jedem Slave einmalig ganz zu Beginn mitgeteilt werden.
2. **type** task_data: Daten, die für jeden Task unterschiedlich sind.
3. **type** task_result: Ergebnis(se) eines einzelnen Tasks.

Diese Datentypen sind als Fortran-Strukturen zu implementieren, die die gewünschten Variablen beinhalten. Wird ein Datentyp nicht benötigt (z. B. wenn keine globalen Parameter benötigt werden), kann einfach eine leere Struktur verwendet werden.

Ein Beispiel für **type** task_data:

```
integer, parameter :: N = ...
[...]
! This type defines the data which is used for a single task
! in the direction Master -> Slave
type task_data
  sequence ! important - do not remove this line
  integer, dimension(N) :: vect
  double precision :: x
  integer :: i
end type task_data
```

Sobald alle Datentypen definiert sind, muss die Struktur der verwendeten Datentypen dem MPI-Basissystem mitgeteilt werden (schlussendlich dafür, dass dieses berechnen kann, wie viele Bytes für die jeweiligen Daten hin- und hergeschickt werden müssen). Dafür sind folgende drei Funktionen vorgesehen:

1. **function** Define_MPI_Global_Data()

2. **function** Define_MPI_Task_Data()

3. **function** Define_MPI_Task_Result()

Diese erwarten ein Rückgabewert vom Typ integer, der am einfachsten mit Hilfe folgender Hilfsfunktionen aus dem Modul mpi_helpers bestimmt werden kann. Diese können zusammen mit den vordefinierten MPI-Basis-Datentypen wie z. B. MPI_INTEGER, MPI_DOUBLE_PRECISION, ... verwendet werden.[1]

1. **function** Define_MPI_Struct(): für Fortran-Strukturen.

2. **function** Define_MPI_Vector(): für Vektoren bzw. Arrays, auch für Matrizen.

3. **function** Define_MPI_Empty_Set(): zur Verwendung, wenn ein Datentyp nicht benötigt wird.

Diese Funktionen können auch beliebig verschachtelt verwendet werden (z. B. um ein Array einer Struktur oder eine Matrix als Array eines Arrays darzustellen). Auf das Beispiel von oben angewendet:

```
function Define_MPI_Task_Data() result(newtype)
  implicit none
  integer :: newtype
  integer :: vect_type

  vect_type = Define_MPI_Vector(MPI_INTEGER,N)
  newtype   = Define_MPI_Struct([vect_type, &
                                 & MPI_DOUBLE_PRECISION, &
                                 & MPI_INTEGER])
end function Define_MPI_Task_Data
```

3.2 my_calculactions

In diesem Modul werden die verschiedenen Aktionen definiert, wie sie in den Abbildungen 1 und 2 vorkommen. Das sind folgende:

1. **subroutine** Tasks_Initialize (): Definiert die Tasks, welche abgearbeitet werden sollen. Außerdem können die globale Daten hier gesetzt werden.

2. **subroutine** Task_Compute(): Die Berechnung eines Tasks.

[1] Eine Auflistung der MPI-Basis-Datentypen findet sich beispielsweise hier: `https://computing.llnl.gov/tutorials/mpi/#Derived_Data_Types`

3. **subroutine** Task_Recieve(): Aktionen nach dem Empfangen eines Task-Ergebnisses wie z. B. das Abspeichern der Ergebnisse.

4. **subroutine** Tasks_Analysis(): Wird ausgeführt nachdem alle Tasks abgearbeitet wurden.

Für die Subroutine Tasks_Initialize werden die Prozeduren Queue_Add_Task() und Queue_Set_Global() benötigt, um dem System mitzuteilen, welche Tasks überhaupt abgearbeitet werden sollen bzw. welche globalen Daten an alle Slaves geschickt werden sollen:

```
subroutine Tasks_Initialize()
  implicit none
  type(global_data) :: global
  type(task_data) :: task

  global = [...]
  call Queue_Set_Global(global)
  task = [...]
  call Queue_Add_Task(task,1)
  task = [...]
  call Queue_Add_Task(task,2)
  [...]
global
```

Das zweite Argument von Queue_Add_Task() ist außerdem eine beliebige (positive) Nummer, welche dem Task zur späteren Identifizierung mitgegeben werden muss (siehe Abbildung 2).

Die anderen Prozeduren (Task_Compute(),Task_Recieve() und Tasks_Analysis()) bedürfen weniger Erklärung:

```
subroutine Task_Compute(dat,global, res)
  implicit none
  type(task_data), intent(in) :: dat
  type(global_data), intent(in) :: global
  type(task_result), intent(out) :: res
  ! irgend eine Berechnung, die aus 'dat' und 'global' ein
     'res' berechnet
  res = [...]
end subroutine Task_Compute
```

```
subroutine Task_Recieve(res,task_id)
  implicit none
  type(task_result), intent(in) :: res
```

```
    integer, intent(in) :: task_id
    ! hier kann beispielsweise das Ergebnis der einzelnen Tasks
       fuer die spaetere Analyse abgespeichert werden.
  end subroutine Task_Recieve

  subroutine Tasks_Analysis()
    implicit none
    ! irgend eine Analyse nachdem alle Tasks berechet worden sind.
  end subroutine Tasks_Analysis
```

4 Kompilieren und Ausführen

Dem Paket liegt bereits ein Makefile bei, welches lauffähig sein sollte. Zu beachten ist, dass durch die Verwendung von Funktionszeigern, welche erst im Fortran 2003 Standard definiert sind ein möglichst aktueller Fortran-Compiler benötigt wird. Der beliebte GNU Fortran Compiler (gfortran) unterstützt diese Funktion beispielsweise erst ab Version 4.4.

Deshalb gibt es in dem Paket zwei Makefiles, eines (`makefile`) das die spezielle Version von gfortran 4.4.5 voraussetzt, und eines (`makefile.default`), welches den Standardcompiler verwendet. Wenn möglich sollte das Zweite verwendet werden.

```
$ make -f makefile.default
```

Ausgeführt wird das kompilierte Programm über den von MPI bereitgestellten Befehl mpiexec[2]. Diesem Befehl kann die Anzahl der Prozesse und ein sogenanntes *machinefile* übergeben werden, in welchem die Rechner angegeben sind, auf welchen die Prozesse parallel laufen sollen.[3]

```
$ mpirun -np 5 -machinefile machinefile.txt ./queue
```

Anmerkung: Wird das kompilierte Programm stattdessen direkt ausgeführt (oder wird über mpiexec bewusst nur ein einzelner Prozess erzeugt), dann bleibt das Programm trotzdem noch lauffähig! Der Programmablauf wird intern automatisch so angepasst, dass der alleinige Prozess abwechselnd die Aufgaben des Masters und eines Slaves übernimmt (siehe Abbildung 1).

[2] siehe `http://www.open-mpi.org/doc/v1.2/man1/mpiexec.1.php`

[3] Mehr dazu in den man-pages (man mpiexec).