UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Scientific Computing (COC)

Lab 1: Introduction to PETSc

Year 2021-2022

Contents

Introduction

This introductory lab exercise has a duration of 1 session. The objective is to get familiar with the work environment.

The lab work is done using the `kahan` cluster, so it is convenient to have the information describing its use handy.

To carry out this lab session we will use the following files:

Session 1	Introduction	`$PETSC_DIR/src/sys/tutorials/ex1.c`
		`$PETSC_DIR/src/sys/tutorials/ex2.c`
		`prac1a.c`

1 Work environment

In `kahan`, the PETSc library is located at `/opt/petsc-x.x`, where `x.x` indicates the installed version. If there are several available, we recommend using the most recent one.

It is advisable to dedicate a few minutes to explore the PETSc directory tree, in particular:

- There are several "architecture" directories named `arch-*`.
- In `include` we can find the `*.h` files for each of the classes.
- In `src` we can find the source code for each of the classes. It also includes examples for the different classes, in subdirectories named `tutorials`. For instance, in `src/mat/tutorials/` we have the examples for class `Mat`.
- The documentation in HTML format might be included. However, it is more convenient to check this documentation online at the address `https://petsc.org/release/docs/`

To work with PETSc it is necessary to set variables `PETSC_DIR` and `PETSC_ARCH`, for example:

```
$ export PETSC_DIR=/opt/petsc-3.16
$ export PETSC_ARCH=arch-linux-gnu-c-debug
```

The value of `PETSC_ARCH` must coincide with any of the `arch-*` directories present in the installation. Each `arch-*` represents a different configuration, for example if it has been configured with complex scalars, or if optimized compilation has been activated or not. A `debug` configuration must be used for development and an `opt` configuration must be used whenever we want to measure performance of the codes.

```
default: ex1

include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules

ex1: ex1.o
        -${CLINKER} -o ex1 ex1.o ${PETSC_KSP_LIB}
        ${RM} ex1.o
```

Figure 1: Example of a `makefile` to compile a PETSc program.

To compile programs it is convenient to use a `makefile` as the one shown in Figure 1. (Note: the white space at the beginning of the two last lines must be a tab character). Try it for example with program `$PETSC_DIR/src/sys/tutorials/ex1.c` (Note: the example must be compiled in a subdirectory under your `$HOME` directory, so copy the file there and add the makefile). The last three lines of the `makefile` can be repeated for other examples, replacing e.g. `ex1` with `ex2`. The instruction

```
$ make ex1
```

creates the executable file for the indicated example. You will see that `mpicc` is being used to compile and link. With

```
$ ldd ex1
```

you can see all dynamic libraries being used by the program.

PETSc programs are MPI programs, so to run them one has to take this into account. For short sequential executions (with a single MPI process) or in parallel with only a few processes, it can be executed interactively (in the *front-end* of `kahan`), for example with

```
$ mpiexec -n 2 ./ex1 [options]
```

For longer executions, especially in case that we want to measure execution times, the queue system must be used, in particular the `coc` queue as is shown in the example of Figure 2 (for more details check the documentation of `kahan`).

2 Basic PETSc programs

2.1 Hello world

We start with the typical "hello world" program, that simply prints a message in the console. This is what is done by example `ex1.c` in `$PETSC_DIR/src/sys/tutorials`.

Compile and run the program, for different number of processes. Observe the source code in detail to understand the behavior of the executions. Check the online documentation for the PETSc functions

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --ntasks=4
#SBATCH --time=5:00
#SBATCH --partition=coc

mpiexec ./ex1 [options]
```

Figure 2: Script to run in the queue system with 4 MPI processes in different nodes.

(`PetscInitialize()` and `PetscPrintf()`). Be aware that `PetscPrintf()` works in one way or another depending on which communicator is passed (there are two predefined communicators: `PETSC_COMM_WORLD` and `PETSC_COMM_SELF`).

The documentation for `PetscInitialize()` shows several command-line options (*Options Database Keys*). These options also appear when running:

```
$ ./ex1 -help
```

Try to execute with some of these options, for example `-info`, `-get_total_flops` or `-malloc_info`. The `-log_view` option is very useful and we will use it later on.

Both `PetscInitialize()` and `PetscFinalize()` are compulsory in any PETSc program. Try commenting out the line with `PetscFinalize()`. What happens? Even in the case that no error is produced, some options such as `-log_view` no longer work.

2.2 Synchronized printf

Example `ex2.c` in the same directory is similar to the previous one, but performs the `printf` operations in a synchronized way, that is, the lines appear in order because the processes coordinate with each other. Run it with e.g. 4 processes. What happens if the call to `PetscSynchronizedFlush()` is commented out or moved to another place?

The documentation states that the `PetscSynchronizedFlush()` operation is **collective**. Try enclosing the call between `if (rank>0) { ... }`. What happens? Do you get the same behavior if you do this with one of the calls to `PetscSynchronizedPrintf()`?

2.3 Ordered list of random numbers

Example `prac1a.c` generates n random numbers and then sorts them. The parameter n can be specified in the command line:

```
$ ./ex1 -n 100000
```

Observe how the `-n` option is managed in the source code with `PetscOptionsGetInt()`. The program has another argument `-view_values` (boolean) to indicate whether the values must be printed or not. Add a new option `-alpha` that gets a real value to be added to the generated random values (instead of the `2.0` in the original code).

This example creates a `PetscRandom` object that is destroyed at the end. It also allocates dynamic memory to store the list of random numbers, by means of `PetscMalloc1()`, and deallocates it at the end with `PetscFree()`. A correct PETSc program should free all memory that it has allocated; to check this, one can run with option `-malloc_dump`, that prints information about unfreed memory. Try commenting out the call to `PetscFree()` or to `PetscRandomDestroy()`.

Try the `-log_view` option with this program, running it with a large value of `n` so that the execution time is significant. Observe the provided information, in subsequent lab sessions we will be able to analyze this report in more detail.

Another alternative to measure execution times is to insert a few calls directly in the code to measure time, with `MPI_Wtime()` or with `PetscTime()`[1]. Try it, printing the time with `PetscPrintf()`, and compare it with the one shown by `-log_view`.

We have seen that all calls to PETSc are followed by an error-checking macro, `CHKERRQ`, for example:

```
ierr = PetscRandomGetValue(rnd,&value);CHKERRQ(ierr);
```

If the error code `ierr` is different from zero, an exception is thrown and by default `CHKERRQ` aborts the execution and shows information about the error and the point of the program where it has been generated. To try it, force an error artificially, for example commenting out the call to `PetscRandomCreate()`. Observe the printed information. Another error is generated if the line `n++;` is added before the call to `PetscSortInt()`. Which is the reason of the error in this case?

[1] To use this function it is necessary to include `petsctime.h`