Generalising a parallel code into a reusable framework

1 Introduction

Based on the 2D geometric decomposition of the serial code in practical one, we are going to look at extending this 2D code into more of a reusable framework than the code you were initially given. The idea is that, regardless of the solver the programmer wishes to use, much of the existing code such as the work you have done on parallelisation and decomposition can be reused.

2 Function pointers in C

C allows you to define a variable which points to the address of a function. Such a variable is termed a *function pointer*. The syntax is a bit different from normal variables, and best described using an example.

```
1.  int fn1(float a, int b) {
2.    return (int) a+b;
3.  }
4.
5.  int main() {
6.    int (*fn_pointer)(float, int);
7.
8.    fn_pointer=&fn1;
9.
10.   int result = (*fn_pointer)(2.5, 8);
11.   return 0;
12. }
```

Lines 1-3 define a function which takes in a float and int and returns an int which is the sum of these numbers. At line 6 we define a variable ***fn_pointer*** to hold a reference to this function and at line 8 we assign a reference to the ***fn1*** function to ***fn_pointer***. Line 10 illustrates how one calls a function through its function pointer. A really common use of function pointers is in passing functions to other functions.

```
1.  int fn1(float a, int b) {
2.    return (int) a+b;
3.  }
4.
5.  int fn2(float a, int b) {
6.    return (int) a-b;
7.  }
8.
9.  int invoke(int (*fn_pointer)(float, int), float a, int b) {
10.  return (*fn_pointer)(a, b);
11. }
12.
13. int main() {
14.   int result = invoke(fn1, 2.5, 8);
15.   int result2 = invoke(fn2, 2.5, 8);
16.   return 0;
17. }
```

In lines 1-3 and 5-7 we define two functions, ***fn1*** and ***fn2*** which both take in a float and int and return an int but have different behaviours (one performs an addition and the other a subtraction.) The ***invoke*** function takes in a function pointer (to a function which returns an int takes a float and int as arguments) and basically will invoke the function pointer with the additional float and int which has been passed to ***invoke***, then returning the resulting int. Lines 14 and 15 illustrate calling ***invoke*** with ***fn1*** and ***fn2***, as these functions have the same signature then it is irrelevant to the ***invoke*** function what they actually do and this is an example of the caller providing the actual behaviour kernel. From this you should be able to see that it would be trivial to define another function with the same signature as ***fn1*** and ***fn2*** and call ***invoke*** with this.

3 Procedure pointers in Fortran

The concepts of procedure pointers in Fortran are very similar to those in C, but obviously the syntax is a bit different. In the example below we have an interface block in lines 4-9 which contains the ***fn_pointer*** function definition. This signature is identical to the C example and it takes in a floating point (real), integer and returns an integer. Now that this is defined we can use the type ***procedure(fn_pointer)*** as per line 27, which is saying that the caller will provide a procedure called ***provided_function*** that has a signature defined in ***procedure(fn_pointer)***. In Fortran you can use a procedure variable as like any other procedure and this is illustrated at line 31 where the function is actually called. In lines 40 and 41 we are calling ***invoke*** with ***fn1*** and ***fn2***, the ***invoke*** function really doesn't care about what is actually being done inside the provided procedure pointer - just as long as it returns an integer takes in a real and integer as per the type definition. From this example it should be obvious how you could add another ***fn3*** function to do some different behaviour and provide this to the ***invoke*** function.

```
1.  module example_mod
2.   implicit none
3.
4.   interface
5.    integer function fn_pointer(a,b)
5.      real, intent(in) :: a
7.      integer, intent(in) :: b
8.    end function fn_pointer
9.   end interface
10. contains
11.
12.  integer function fn1(a,b)
13.    real, intent(in) :: a
14.    integer, intent(in) :: b
15.
16.    fn1=a+b
17.  end function fn1
18.
19.  integer function fn2(a,b)
20.    real, intent(in) :: a
21.    integer, intent(in) :: b
22.
23.    fn2=a-b
24.  end function fn2
25.
26.  integer function invoke(provided_function, a, b)
27.    procedure(fn_pointer) :: provided_function
28.    real, intent(in) :: a
29.    integer, intent(in) :: b
30.
31.    invoke=provided_function(a,b)
32.    end function invoke
```

```
 end module example_mod

program example
 use example_mod
 implicit none

 integer :: result, result2
 result= invoke(fn1, 2.5, 8)
 result2=invoke(fn2, 2.5, 8)
  print *, result, result2
end program example
```

4 Adding in another solver

The Jacobi iterative method is slow and inefficient, one way in which we can speed it up is via over relaxation. Basically, this works on the assumption that at each iteration the solution is moving closer to the final answer - we can speed this up by artificially moving it a bit further each iteration than the equation computes. The extra amount by which we move the solution each iteration, called *W* is critical and lies between 1 and 2. At some point the solver is likely to go a bit too far and overshoot the answer but given a sensible value for *W* it will correct itself. If the value of *W* is too large then the system will fail to converge on the answer (called divergence) so initially it is best to run with *W* equal to 1.1 and increase this experimentally. As you increase *W* you should see the number of iterations decrease until this divergence point.

Your task is to extend the parallelised 2D code to allow for additional solvers to be ``plugged in''. I suggest starting with the code for this practical I provide on Learn, which is the non-blocking parallel version and I have popped in these two solvers (Jacobi and over relaxation) as separate functions (procedures in Fortran) and called from within the iteration of the code. The general idea is that a programmer should be able to select their own solver and use all your existing parallelisation, iteration structure and residual calculation.

I provide a submission script for Cirrus, you will need to submit providing your budget code, for instance ***sbatch --account=[budget] subpractical.srun*** where *[budget]* is your personal budget code*.*

If you want to run on ARCHER2 then that's also fine, I provide a makefile for ARCHER2 (makefile_archer2) which you can use with ***make -f makefile_archer2*** and a submission script (submit_archer2.srun).

1. Download the code for this practical four (you can find these linked from the timetable page of the PDP Learn site or on the practical page). I provide both C (pollution.c) and Fortran (pollution.F90) versions, so use which ever you feel most comfortable with.
2. Move the majority of the existing code into its own function (it is OK to keep the MPI initialisation/finalisation and local domain size calculation in main if you wish to) and call this from the program entry point.
3. Take this opportunity to do some re-factoring on the code, you can probably split the global norm calculation out into a separate function, which can be called for both the initial residual and solution residual each iteration with ***u_k*** . You can probably also split out your halo swapping code into its own function too.
4. Configure your code so that the function with the iteration in takes in a function pointer to the actual solver to use (pointing to the specific solver function – see next point).
5. The Jacobi and Jacobi over relaxation solvers are already provided as two separate functions (***jacobi_solver*** and ***jacobi_sor_*solver**) in the code for this practical. You will see that we have also defined ***W*** as a pre-processor directive. Note that effectively the ***u_kp1[i]*** in the RHS of

the calculation for the Jacobi SOR is referencing the value held at ***i*** two iterations ago (before it is overwritten.)

6. Hook it all up - you can either select the solver to use as a pre-processor directive or some runtime check from the main function.

Once you have done this, have a play with the value of *W* and see how this changes the overall number of iterations to convergence. You might want to add a check that the system has not diverged (due to *W* being too high) - an easy way of doing this is to check that the value of ***norm*** has not exceeded 1 at each iteration. This is a bit of a contrived example, and over relaxation is not normally applied to Jacobi - but we are lucky in this situation that our equation will converge with it.

5 Advanced exercises

This section is for those who have completed the Jacobi relaxation exercise and wish to further explore the example. Don't worry if you don't get onto doing this, the most important thing is that you get the basic exercise completed.

There is another type of iterative method called Gauss-Seidel. This is very similar to Jacobi but, instead of updating ***u_kp1*** each iteration with the values in ***u_k***, it will instead work directly updating ***u_k***. By doing this, not only does it save considerable memory (as we no longer need ***u_kp1***) but also by directly operating on the latest values at all times this speeds up the convergence quite considerably. The downside of this is that the order of calculation does matter therefore, unlike Jacobi, the convergence behaviour of Gauss-Seidel does change with program decomposition (i.e. it will take a different number of iterations to converge on different numbers of processors.)

1. Extend your framework code to support the Gauss-Seidel solver. At its simplest this involves replacing ***u_kp1*** on the left hand side of the solver assignment (e.g. in the Jacobi or Jacobi SOR functions) with ***u_k*** . However, now we no longer need ***u_kp1*** or the ***temp*** variable when swapping values around. Therefore, extend the framework to so that each solver will determine the memory that it requires. You might want to add some form of initialisation and finalisation calls for Jacobi and Gauss-Seidel (via additional function pointers.) To make it easier you can represent these variables using global variables if you wish, but a better (and more complex) design might be to use some common structure which encapsulates the specifics of them.
2. Extend the framework to support the Successive Over Relaxation (SOR) solver, this is Gauss-Seidel with some extra relaxation *W*. Unlike our Jacobi with over relaxation above, SOR is commonly used. This involves combining the ***jacobi_sor_solver*** and your new Gauss-Seidel solver, basically doing the over relaxation writing into ***u_k*** rather than ***u_kp1***. As with Jacobi relaxation you will also need to define ***W*** and I would suggest setting this to 1.1 initially (the over relaxation behaviour will be different than it was with Jacobi.)