UPPSALA
UNIVERSITET

IT 11 084
Examensarbete 15 hp
November 2011

Parallelizing a Software Framework for Radial Basis Function Methods

Linus Sunde

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Parallelizing a Software Framework for Radial Basis Function Methods

Linus Sunde

Numerical simulation of realistic problems requires parallel software, which is non-trivial to develop for modern multi-core architectures. Radial basis function approximations can be used when simulating and tasks when parallelizing.

In a previous master thesis a flexible framework for solving time-independent partial differential equations using radial basis function approximation was reworked. The aforementioned framework is written in Fortran 90 and is entirely serial. Parallelization of software is needed to acquire good performance on modern multi-core hardware.

To improve the performance of the framework, parts of it are parallelized using a task library being developed at the Division of Scientific Computing at Uppsala University. The task library is written in C++ so an interface to Fortran 90 was created to allow for its usage. Moreover, the framework is extended with two modules, the ”Workflow Manager” and the ”Data Manager”. They respectively handle parallelizing operations provided by the framework and managing data needed by the operations. These modules should allow for easy addition of new operations.

The performance and ease of addition of new operations is tested by adding a single parallel operation to the framework. Test results show that the unavoidable overhead is small and that a speedup is acquired, which however is sublinear when theoretically a linear speedup could have been expected.

Handledare: Martin Tillenius / Elisabeth Larsson
Ämnesgranskare: Michael Thuné
Examinator: Anders Jansson
IT 11 084
Tryckt av: Reprocentralen ITC

Contents

1 Introduction **1**

2 Task library **1**

2.1 Interfacing C++ and Fortran 90 . . . 1
2.2 C interface . . . 2
2.2.1 Name mangling . . . 3
2.2.2 Calling convention . . . 4
2.2.3 Initialization . . . 4
2.3 Fortran 90 interface . . . 4
2.3.1 The `tl` module . . . 4
2.3.2 Handles . . . 5
2.3.3 Tasks . . . 5
2.4 Using the task library . . . 7
2.4.1 Reentrancy . . . 7
2.4.2 Linking . . . 7
2.4.3 Implementation test . . . 7

3 Workflow Manager **9**

3.1 Matrix multiplication example . . . 9
3.2 Data manager class . . . 11
3.2.1 Mutex . . . 12
3.3 Operation class . . . 12
3.4 Workflow manager class . . . 13

4 Radial Basis Function Methods **13**

4.1 Assemble operation . . . 14
4.1.1 Approximation class . . . 14
4.1.2 Problem class . . . 14
4.1.3 Geometry class . . . 14
4.1.4 Expression class . . . 14
4.1.5 Implementation . . . 15
4.2 Implementation test . . . 15
4.2.1 Results . . . 16
4.2.2 Conclusions . . . 17

5 Future work **17**

A Runtimes **20**

1 Introduction

Much existing scientific computing code is written in Fortran. Specifically this is the case for a flexible software for solving time-independent partial differential equations using radial basis function (RBF) approximation which was reworked in a previous master thesis [2]. See [1] for more information about RBF approximation methods.

As multicore processors become more and more prevalent, writing parallel software becomes more and more important. Without parallelization we can not utilize the full potential of the hardware.

We seek to parallelize operations performed by the above mentioned software. The task model was chosen as it gives us, and other potential users of the framework, an easy way to divide larger portions of work into smaller pieces.

To facilitate this parallelization, we will use a task library currently being developed at the Division of Scientific Computing at Uppsala University [3]. It provides functionality such as task creation and barriers and it allows you to specify how a task depends on different variables. That is to say, you can use a number of handles to specify which variables the task reads from, writes to, and adds to. A handle is an abstract construct that can be thought of as simply representing a variable.

This task library is written in C++. We want to create an interface so that the task library is usable from Fortran 90 without performance loss or increased memory usage. The interface should also be constructed so that the Fortran 90 code using the task library is kept as clean and straightforward as possible. Preferably it should also be relatively easy to change existing serial code into parallel code using the task library. Another important aspect is 64-bit memory architecture compatibility. The task library will likely be used on high performance machines with large amounts of memory available. Having access to all of this is a must.

Using this task library we want to continue the design of two of the framework's Fortran 90 modules described in [2]. The first module is called the ”workflow manager”. It provides a set of operations for working with radial basis function approximations. Given a list of operations the intent is then for the module to schedule the operations in an efficient way using the task library to parallelize operations if possible. The second module is called the ”data manager” and is used by the workflow manager to store partial results between operations and final results after all the operations.

In an effort to solve these problems we

- examine different ways to interface the C++ and Fortran 90 codes in an attempt to find a way to fulfill our wishes for a clean and straightforward interface that preferably is easily implemented in existing code,
- identify problems and possible pitfalls with using Fortran 90 codes in a parallel fashion,
- construct the workflow manager and data manager as well as implementing a number of operations.

2 Task library

2.1 Interfacing C++ and Fortran 90

There are at least two different ways of making C++ code accessible from Fortran 90. Both of these approaches require a flat C-interface to the C++ code.

The first approach is to define the functions in the flat C-interface in such a way that they are directly usable from Fortran 90. This includes naming the C-interface functions in such a way that they can be seen from Fortran 90 and using argument types that exists in both languages.

The second approach is to use the intrinsic module `ISO_C_BINDING` [7]. This module allows you to ”bind” a Fortran 90 function or subroutine to a C function. It also provides types that represent

existing C types. This module, even though it exists in many Fortran 90 compilers, is not part of the Fortran 90 standard but was added to the standard in Fortran 2003.

We chose to go for the first approach as described in [4] with a few exceptions such as storing void pointers to C++ objects instead of using opaque pointers and allowing some functions in the C-interface to be called directly from Fortran 90 code without going through a Fortran 90 interface. Reasons as to why we chose this approach are given in Sections 2.2 and 2.3.

2.2 C interface

The details for implementing a C++/Fortran 90 interface using the approach we chose are described in great detail in [4]. We bring up a few key points that influenced our design choices.

To understand why we chose this approach we need to know a bit about the functionality the task library provides. The primary function of the task library is to create and schedule tasks. A task is a piece of code that can be executed independently of the rest of the program. Possible data dependencies are explicitly provided as a set of handles. Each handle represents some data. The dependencies can be of either read, write or, add to type. If possible, with respect to the provided dependencies, tasks are run in parallel.

Tasks are created using the C-interface functions `tl_add_task_safe()` and `tl_add_task_unsafe()`. The only difference between these two is that safe version does some extra checks to verify that the input is correct. This is something you might want when debugging code but not while actually running it as it results in loss of performance. We will simply refer to these functions as `tl_add_task()`.

The C interface function `tl_add_task()` needs the following arguments:

- The code to run. This is passed as a function pointer. When called from Fortran 90 this will be a pointer to a subroutine.
- The arguments with which to run the passed function. This is passed as a void pointer to a piece of memory that the function will have to cast to the appropriate type. Since the Fortran 90 subroutines have a defined argument type the function will know how to interpret this piece of memory. In most cases this will probably be a Fortran 90 derived type.
- The size in bytes of the previous argument. We need this since the task library needs to make a local copy of the memory to work with.
- Three vectors with handles and the number of elements in each. Each one defining read, write, or add to dependencies of the task.

We decided that the functions passed should have the return type void. That is they should be Fortran 90 subroutines. This removes the need for the C code to handle any return values which could be of some Fortran 90 type not existing in C. If the subroutine is to change or generate data pointers to where to store this should be passed as part of the derived type argument. This means the only thing C will do is receive a subroutine to run and the arguments with which to run it and then, when appropriate, call this subroutine with the arguments. The other two most interesting C-interface functions are `tl_create_handles()` and `tl_destroy_handles()`. These two functions allows you to create the above mentioned dependency handles.

`tl_create_handles()` takes a preallocated handle vector and a number and fills the vector with that many new constructed handles. `tl_destroy_handles()` takes a vector filled with handles using `tl_create_handles()` and a number and destroys that many handles.

The handles are in reality just stored as pointers to the memory they occupy. The reason these functions are the most interesting is because they as arguments use user defined data types that do

not exist in both C and Fortran 90. There are several more functions in the C-interface but these use data types that exists, albeit with different names, in both C and Fortran 90.

Full list of interface functions:

- `tl_init()`
- `tl_destroy()`
- `tl_barrier()`
- `tl_add_task_safe()`
- `tl_add_task_unsafe()`
- `tl_create_handles()`
- `tl_destroy_handles()`

2.2.1 Name mangling

The first problem with accessing the C-interface code from Fortran 90 is name mangling. Fortran 90 and C++ both to some extent rename procedures internally. In the C-interface we can use the extern "C" construct. This tells the compiler that the code should be compiled in C style. This prevents most name mangling.

The external procedure names are still mangled in Fortran 90. The specific compiler we use (Oracle Solaris Studio 12.2) append an underscore to external procedure names. We append an underscore to each of the function names in the C-interface to make them accessible from Fortran 90. The function names are defined in a single header file making it relatively easy to change them to suit the used compiler's name mangling scheme.

Listing 1: tl.h

```
...
#define tl_init             tl_init_
#define tl_destroy          tl_destroy_
#define tl_barrier          tl_barrier_
#define tl_add_task_safe    tl_add_task_safe_
#define tl_add_task_unsafe  tl_add_task_unsafe_
#define tl_create_handles   tl_create_handles_
#define tl_create_handle    tl_create_handle_
#define tl_destroy_handles  tl_destroy_handles_
#define tl_destroy_handle   tl_destroy_handle_
...
```

If we had instead used the `ISO_C_BINDING` module the name mangling problem could have been avoided. This module allows you to create a Fortran 90 interface with subroutines or functions that are implemented in C. You can specify which C function they are to be bound to using the `BIND(C, name="<name of implementing C function>")` attribute.

We chose our approach since it should work on any compiler. Existing code that wants to use the task library might be compiler dependent. Generating the C-interface function names could be done automatically if the Fortran 90 compiler mangling conventions are known.

2.2.2 Calling convention

Unlike C/C++, Fortran 90 uses call by reference instead of call by value as default [4]. This means that the types in the C interface must be defined appropriately. The `tl_init()` function for example which needs an integer will actually be passed a pointer to an integer when called from Fortran 90. This is easily solved by changing appropriate types to pointers instead and then dereferencing them as needed. Arrays are passed from Fortran 90 as a pointer to the first element [6]. This results in both Fortran 90 integers and integer arrays being passed as what appears to C as int pointers.

The pass by reference calling convention is also a possible pitfall when sending arguments to a task. If you have a loop over an integer n from 1 to 10 creating a task in each iteration, then sending the value of n or an reference to n as the argument might have totally different results. Since you don't know exactly when the tasks will be run n might have changed before the task is run if you send a reference.

2.2.3 Initialization

As described in [4] initialization of static and const C++ variables is not guaranteed with a Fortran 90 main. Their recommended approach for solving this problem is using a shared-library that automatically initializes itself correctly.

We chose to make sure the `tl_init()` function, which needs to be called before any other task library function, initializes all the necessary variables.

2.3 Fortran 90 interface

2.3.1 The `tl` module

Now when we have made the C interface available from Fortran 90 we can begin constructing the Fortran 90 side of the interface. The interface is constructed in a module we call "tl" short for task library. The functions `tl_init()`, `tl_destroy()` and `tl_barrier()` are relatively straightforward to implement. Only `tl_init()` has an argument at all and it is a C int. This is equivalent to a Fortran 90 `INTEGER` [6]. The tl module is created with an interface declaring the corresponding functions in the C interface.

Listing 2: The `tl` module

```
module tl

    implicit none

    interface

    subroutine tl_init(num_threads)
        integer, intent(in) :: num_threads
    end subroutine tl_init

    subroutine tl_destroy()
    end subroutine tl_destroy

    subroutine tl_barrier()
    end subroutine tl_barrier

    end interface

end module tl
```

The remaining functions are slightly more complex. They are discussed in 2.3.2 Handles and 2.3.3 Tasks.

2.3.2 Handles

To encapsulate and hide the inner workings of handles on the Fortran 90 side we adopt the object based approach [5] and create a module called `tl_handle_class`. The `tl_handle_class` contains a type called the `tl_handle`. This type represents a handle on the Fortran 90 side and contains enough memory to store one pointer to a handle. This can be done by having it contain an integer pointer which results in the type having the correct size on both 32-bit and 64-bit memory architectures.

Listing 3: `type(tl_handle)`

```
type, public :: tl_handle
    private
    integer, pointer :: address
end type tl_handle
```

We then use this type to define the arguments to `tl_create_handles()` and `tl_destroy_handles()` on the Fortran 90 side. Since the elements of the type are private we ensure that the only way to give them values is by using the mentioned functions. The interface in the tl module is expanded:

Listing 4: The expanded `tl` module

```
module tl

    use tl_handle_class

    implicit none

    interface

    ...

    subroutine tl_create_handles(num_handles, handles)
        use tl_handle_class
        integer, intent(in) :: num_handles
        type(tl_handle), dimension(:), intent(out) :: handles
    end subroutine tl_create_handles

    subroutine tl_destroy_handles(handles, num_handles)
        use tl_handle_class
        integer, intent(in) :: num_handles
        type(tl_handle), dimension(:), intent(out) :: handles
    end subroutine tl_destroy_handles

    end interface

end module tl
```

2.3.3 Tasks

At last we want to add the task creating subroutines to the interface. These are the most troublesome and we finally decided not to add them to the interface and instead call them directly from the Fortran 90 code. This means we lose type safety. While this might sound bad it is actually one of the reasons we do it this way.

The first argument to the `tl_add_task()` functions is a code pointer to the subroutine to run. Fortran 90 allows you to pass subroutines and functions as parameters. Usually you have to state the argument

types and return type in case of a function. This can however be circumvented by defining the subroutine or function passed as external in the interface.

Listing 5: Using `external` to pass arbitrary functions or subroutines as arguments

```
interface
    subroutine tl_add_task(task, ...)
    external task
    ...
    end subroutine tl_add_task
end interface
```

Then you could pass `tl_add_task()` an arbitrary subroutine as long as the subroutine is external, a module procedure or declared in an interface block which are the requirements to be able to use a subroutines as an argument. If we used the `ISO_C_BINDING` module we could utilize the derived type `type(c_funptr)` it provides. This data type represents a C null function pointer. This would however require that we first define the subroutine we want to create a task from with the `BIND(C)` attribute and then cast the subroutine to a `type(c_funloc)` using the `c_funloc()` function. We want to avoid this as it makes the code messier and forces the user of the task library to do more work.

The first problem arises with the second argument which is a void pointer to the struct (or in the case of Fortran 90 a derived type) containing the arguments to be passed. Fortran 90 does not have a type equivalent to a C void pointer. In the interface we would have to define the argument's type. Since this is a derived type containing the input and output arguments to the task subroutine its actual type is unknown. Once again if we used the `ISO_C_BINDING` module it provides a void pointer equivalent type `type(c_ptr)` but would require explicit casting every time `tl_add_task()` was used. By not defining `tl_add_task()` in the interface we can simply pass it an arbitrary derived type which is by Fortran 90 standards passed by reference and hence the C interface receives an pointer to the derived type.

The third argument also gives us some problems. It is supposed to be the size in bytes of the the derived type passed as the second argument. The Fortran 90 standard does not provide an equivalent of C's `sizeof()`. Since the size is needed for the task library to work, a way to calculate the size of a derived type is needed. Luckily most compilers provide `sizeof()` as a compiler dependent extension which then can be used by the user.

In the case `sizeof()` is not available there are some other workarounds. The most general one is probably something similar to the solution proposed by James Van Buskirk in (the usenet group) comp.lang.fortran [8] which utilizes the intrinsic functions `size()`, `transfer()`, `bit_size()` and `selected_int_kind()`.

- `transfer(source, mold)` allows you to convert the bits representing source into the type of mold.
- `size(array)` returns the number of elements in the array array.
- `bit_size(i)` returns the number of bits used by the integer type of i
- `selected_int_kind(i)` returns the kind for an integer data type that can represent all integers n such that $-10^i < n < 10^i$.

This can be combined into the following where bytes at the end will contain the size of the variable `derived_type` in bytes.

Listing 6: Calculating variable size in bytes using intrinsic functions

```
integer :: bytes
integer(kind=selected_int_kind(2)), parameter :: byte = 0
bytes = size(transfer(derived_type, (/byte/)))
```

What we do is we first set `byte` to be an integer that should be able to hold -100 to 100. This requires at least one byte. We then cast the derived type into an array with elements of the same type as `byte`. By counting the number of elements in the array `bytes` we get the size of `derived_type` in bytes. To be sure that `byte` actually has a size of 1 byte (8 bits) we can insert a one time check using `bit_size()`. If it is not 8 bits we can use the information from `bit_size()` to scale properly.

2.4 Using the task library

2.4.1 Reentrancy

Some Fortran 90 compilers, including Oracle Solaris Studio 12.2 used by us, defaults to using the `save` attribute for subroutine and function local variables. Variables with the `save` attribute will keep their value between calls. In other words the variable in question is stored in static memory instead of being allocated on the stack each call. This does not work well with the task library since subroutines used by tasks need to be reentrant. Let us consider a subroutine that takes two arguments which are two integers it is supposed to iterate between. Two tasks are created: Task one is supposed to iterate 1-10 and task two 11-20. Task one starts running setting the subroutine's local variable used to iterate the loop to 1. It then starts iterating and reaches 5. At this point task two is started. It sets the subroutine's loop variable to 11 overwriting task one's progress.

To prevent this problem local variables used in subroutines that are used by several tasks simultaneously need to be forced to be allocated on the stack. Fortran 90 introduced the `recursive` keyword to allow for `recursive` subroutines. For `recursive` subroutines to work local variables need to be allocated at function entry or previous results are lost. This allows us to make specific subroutines reentrant.

Another possible solution is to use compiler specific flags. The compiler used by us provides the -stackvar flag which forces all local variables in the compiled file to be allocated on the stack.

Allocating local variables on the stack could cause problems with existing scientific computing code as it often works with large arrays or matrices. If the existing program uses large local arrays this could cause the stack to overflow.

2.4.2 Linking

If we use a Fortran 90 compiler to link the Fortran 90 object files using the task library to the object file produced when compiling the task library we will need to specify a number of extra libraries. These are the libraries used by the task library that the C++ compiler linked by default. When we compiled using the Oracle Solaris Studio 12.2 CC compiler the flag -v allowed us to examine exactly what was linked. The libraries linked was libCstd which is Sun's implementation of the C++ standard library and libCrun which is their C++ runtime library. These would have to be linked when linking with a Fortran 90 compiler as they would probably not be linked by default.

2.4.3 Implementation test

To test the design and implementation of the task library interface with respect to the requirements listed in Section 1 a small piece of existing code was parallelized. The code consists of a loop that, with some slight modification, is perfectly parallel. Since the loop is perfectly parallel we will not need to use handles.

Listing 7: The modified serial loop

```
do k=1,size(phi,1)
    sol(k) = new_sol(phi(k),A(k),rhs,ep,epRad(k),lam=lam)
end do
```

We replace the call to the `new_sol()` function with a call to a new wrapper subroutine named `new_sol_task()`. This new subroutine takes all the arguments and packs them into a derived type. This is necessary since we, as mentioned earlier, only can pass a single argument when creating tasks. We also send the variable to which the result is assigned since the task will need to know where to store the output. It then creates a task which will run another new wrapper subroutine named `new_sol_unpack()` which unpacks the arguments from the derived type and calls the original function.

Listing 8: The parallel loop

```
do k=1,size(phi,1)
    call new_sol_task(sol(k),phi(k),A(k),rhs,ep,epRad(k),lam)
end do
```

Listing 9: The wrapper functions and the derived type

```
type new_sol_args
    type(solution),pointer  :: sol
    character(len=80)       :: phi
    type(operator)          :: op
    type(field)             :: rhs
    type(epsilon)           :: ep
    real(kind=rfp)          :: epRad
    logical                 :: lam
end type new_sol_args

subroutine new_sol_task(sol,phi,op,rhs,ep,epRad,lam)
    type(solution), target, intent(out) :: sol
    character(len=*), intent(in)        :: phi
    type(operator), intent(inout)       :: op
    type(field), intent(in)             :: rhs
    type(epsilon), intent(in)           :: ep
    real(kind=rfp), optional            :: epRad
    logical, optional                   :: lam

    type(new_sol_args)                  :: args

    args%sol   => sol
    args%phi   =  phi
    args%op    =  op
    args%rhs   =  rhs
    args%ep    =  ep
    args%epRad =  epRad
    args%lam   =  lam

    call tl_add_task_unsafe(new_sol_unpack, args, sizeof(args), 0, 0, 0, 0, 0, 0)

end subroutine new_sol_task

subroutine new_sol_unpack(args)
    type(new_sol_args), intent(inout) :: args

    args%sol = new_sol(args%phi, args%op, args%rhs, args%ep, args%epRad, args%lam)

end subroutine new_sol_unpack
```

Since some of the existing code is now used inside tasks we need to make sure those parts are reentrant. This requires a bit of work since we need to ascertain which parts of the existing code that can be reached from inside tasks. The simplest solution is to compile all the Fortran 90 code with the -stackvar flag or equivalent. If we do not want to compile all the files containing Fortran 90 code using the -stackvar flag we can instead trace dependencies from the initial call to existing code. If we know

which files that needs to be reentrant we can simply compile only these using -stackvar. If we do a more thorough analysis and find out exactly which subroutines or functions that need to be reentrant we can make them so by using the `recursive` keyword.

We note that changing existing code to use the task library was relatively straightforward. Making sure the right parts are reentrant is probably be the hardest part. We did not need to change much of the existing code. Instead we wrote two simple wrapper functions that pack and unpack the arguments and uses the existing code to do any calculations. The existing code was initially written for a 64-bit memory architecture but was tested using both 32-bit and 64-bit memory architecture.

3 Workflow Manager

Above we created a Fortran 90 interface to the task library with the intent to use it to construct the workflow manager. So what is the workflow manager? Starting from and continuing the design from [2] the workflow manager is supposed to provide users with a set of basic building blocks for working with radial basis function approximations. The user can combine these building blocks into a scheme which describes what he or she wants to do and then tell the workflow manager to execute the scheme. In other words, you give the workflow manager a list of predefined operations which it then performs in an efficient way. Efficient includes finding and scheduling potential parallel operations which is where the task library comes into the picture. It also includes knowing when partial results can be thrown away which, as described in Sections 3.1 and 3.2, can be solved using handles. The partial results need to be accessible by the different tasks the operations will be split into. This will be handled by a central data storage called the data manager which allows you to access data by the means of human readable names.

3.1 Matrix multiplication example

To get started with the implementation of the workflow manager we defined two simple operations. Operation one, ”Create Matrix”, which creates a matrix with some dummy values and operation two, ”Matrix Multiplication” which multiplies two matrices storing the product. We want to use these operations in the following way illustrated in Listing 10.

Listing 10: Matrix multiplication using the workflow manager

```
program main

    use wfm_class

    implicit none

    type(wfm) :: w
    type(op), dimension(4) :: ops

    call tl_init(4)

    call op_create_matrix(ops(1), "A")
    call op_create_matrix(ops(2), "B")
    call op_mult_matrix(ops(3), "A", "B", "C")
    call op_mult_matrix(ops(4), "A", "C", "Result")

    call wfm_new(w, 4)
    call wfm_add(w, ops(1))
    call wfm_add(w, ops(2))
    call wfm_add(w, ops(3))
    call wfm_add(w, ops(4))
```

```
    call wfm_execute(w, (/ "Result" /))
    call wfm_print_dm(w)

end program main
```

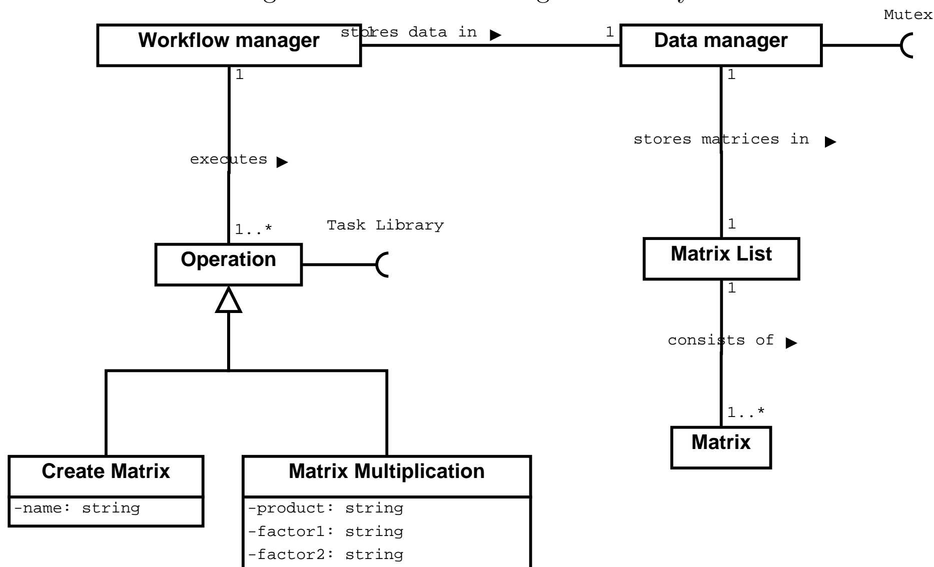
We first declare a single workflow manager variable and an array of operation variables. The desired operations are then created. In this case we first create two "Create Matrix" operations objects which will create the matrices A and B. We then create two "Matrix Multiplication" operations objects of which the first will multiply A and B storing the result in C and the second multiply A and C storing the result in Result. We then create a workflow manager with space for 4 operations and add our operations to it after which we tell the workflow manager to execute storing Result.

The data we work with in this case are the matrices A, B, C, and Result. These will be stored in the data manager allowing the workflow manager to receive references to them by using their human readable names. The last row of code prints the contents of the data manager. What do we expect to be in the data manager at this point? The answer is: only Result. We only specified that we want to store Result when we executed the workflow manager. This means that the other matrices should be automatically removed, preferably as soon as they are not needed anymore.

From this we extract the design presented in Figure 1. The workflow manager class contains an instance of the data manager class which is used to store the data. In the case above we need to store matrices so the data manager needs to be at least able to store those. We settle for a list of matrices for our simple example but for any serious attempt this should be at least a tree structure allowing for lookup of names in $\mathcal{O}(\log_2(n))$ or preferably some kind of hash table allowing average $\mathcal{O}(1)$. For each matrix we also need to store a handle which represents that matrix. Moreover, since tasks will be adding and removing data from the data manager we will also need access to mutexes to make this task safe.

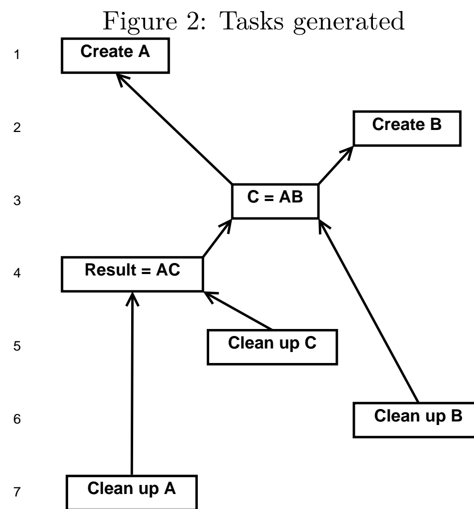
The workflow manager also contains a list of operations which can be of either of the two kinds mentioned above. When we execute the workflow manager these operations are translated into tasks dependent on appropriate handles. This means operations need access to the task library.

Figure 1: Workflow manager initial layout



In this example each operation will be translated into a corresponding task when executing the workflow manager. As we will see in Section 4.1.5 this is not necessarily the case. The data manager will also generate a number of clean up tasks as we want to remove matrices as they become obsolete. After generating all the operation tasks we generate for each variable not defined as ”to be saved” in the execute call a task which removes it from the data manager. These tasks will be write dependent on the variable’s associated handle and since it was generated after the other tasks the task library will ensure it is the last task run in that chain of tasks dependent on that handle. Figure 2 shows the order the tasks are generated and which previous tasks that need to have finished for a task to be allowed to run. The clean up tasks happens to be generated in reverse creation order in our list implementation.

Figure 2: Tasks generated



3.2 Data manager class

As we have discussed above the data manager’s main purpose is to keep track of existing data and given a human readable name be able to produce a reference to the associated data. Having existing data also means we need a method of declaring what we want to create. Since Fortran 90 is a strongly typed language this means we will need a data structure, a declaration function, and a get reference function for for each kind of data type that the data manager is supposed to be able to contain. The declaration functions takes a human readable name and declares a variable with the given name in the data manager. The type of the variable depends on which declaration function that was called, as we said we will need one for every type of data that we can store in the data manager. After the variable has been declared you can get a reference to it by using the appropriatly typed version of get reference function. When you have the reference, which is nullified at declaration, you can allocate it and fill with the data you want. In our example, we limit ourselves to two dimensional matrices. We also need to be able to get the handle associated with a specific variable given its human readable name so that we can create the dependencies when generating tasks.

Furthermore, we need a function which generates the clean up tasks. Since we want to be able to save specified variables we let the function take an array of human readable names as argument and generate tasks that will remove all other variables. To be able to iterate over the variables in the data manager these need to have been declared. This means since there is no guarantee that any tasks have been run yet declaration of variables in the data manager should not be done in tasks but preferably

instead just before the tasks needing that variable is created. That is, the variables are declared in serial while the actual allocation can be done in parallel in different tasks. This guarantees that all variables that needs to be removed have been declared when the clean up tasks are to be generated since this is done after translating operations into tasks.

- `dm_type_add(dm, name)` declare a variable of type `type` with name `name` in the datamanager `dm`
- `dm_type_get(dm, name)` get a reference to a variable of type `type` with name `name` from the datamanager `dm`
- `dm_handle_get(dm, name)` get the handle of the variable with name `name` from the datamanager `dm`
- `dm_cleanup(dm, save)` generate tasks removing all variables in the datamanager `dm` except those listed in `save`

3.2.1 Mutex

Since the data manager will be used by several tasks running concurrently it needs to be task safe. More precisely its functions that modifies internal data structures needs to ensure that those are kept in a consistent state. One way to solve this problem is to use mutexes.

To give us access to mutexes we created a Fortran 90 interface to Pthreads. This is basically a subset of what is presented in [9]. Since the task library already uses Pthreads in Unix environments this does not add any dependencies. We defined a Fortran 90 mutex derived type and an interface consisting of the following functions.

- `pthreads_mutex_init(mutex)` initialized a mutex variable
- `pthreads_mutex_destroy(mutex)` destroys a mutex variable
- `pthreads_mutex_lock(mutex)` locks a mutex variable
- `pthreads_mutex_unlock(mutex)` unlocks a mutex variable

3.3 Operation class

The operation class represents a single operation that the workflow manager can be told to do. It contains the needed information to be able to perform that operation. In the case of a matrix multiplication this would be the name of the factors as well as the result.

To make the process of setting up a workflow manager easy we want a single function for adding operations to the workflow manager. This is solved by having the superclass operation. Different subclasses represent different operations. Since Fortran 90 is not a fully object-oriented language this is solved by having the superclass contain a pointer of each type of subclass [5]. In a specific operation object each of these pointers except one will be nullified with the one not nullified being the actual object. Function calls on the superclass are dispatched to the appropriate subclass implementation.

To be able to use the operation class we need to be able to create operations and to run them, that is translate them into a set of tasks. The subclass implements its translation into tasks in a subroutine which can be called from the superclass. Hence when translating an operation into tasks you only need to tell the operation to translate into tasks, and the call will as described above be dispatched to the appropriate implementation. Exactly how the operation is divided into tasks is up to the implementer. In our simple test case matrix multiplication was translated into a single task but this could just as easily have been some block algorithm generating several tasks.

3.4 Workflow manager class

Given the data manager and operation classes above we construct the workflow manager class in accordance with the layout presented in Figure 1. The workflow manager class contains the operations it is to perform and a data manager to facilitate the usage of data between tasks.

- `wfm_add(wfm, op)` Adds the operation `op` to the workflow manager `wfm`
- `wfm_execute(wfm, save)` Executes the workflow manager `wfm` saving the variables in the list of names `save`

4 Radial Basis Function Methods

As we mentioned in Section 3 the ultimate goal of the workflow manager is to provide predefined operations for working with radial basis function (RBF) approximations. So far we have only implemented a simple matrix multiplication test case but now we are ready to start the implementation of actual RBF operations.

The following method is described in detail in [10]. An RBF is a function with a value which only depends on the distance from a center point $\underline{x}_j$. We will use the Gaussian RBF $\phi(\|\underline{x} - \underline{x}_j\|, \epsilon) = e^{-(\epsilon\|\underline{x}-\underline{x}_j\|)^2}$ which also accepts a shape parameter ϵ. Given a set of N points $\underline{x}_j$ the function value in a point $\underline{x}$ is approximated as $s(\underline{x}, \epsilon) = \sum_{j=1}^{N} \lambda_j \phi(\|\underline{x} - \underline{x}_j\|, \epsilon)$. The coefficients λ_j can be calculated by collocation with given function values.

As a case study, we apply RBF to solve a Poisson problem which is a kind of partial differential equation (PDE):

$$\begin{aligned} u(\underline{x}) &= g(\underline{x}) \qquad \text{on } \partial\Omega, \\ \Delta u(\underline{x}) &= f(\underline{x}) \qquad \text{in } \Omega. \end{aligned} \tag{1}$$

Here, Ω is a d-dimensional domain, that is $\Omega \subset \mathbb{R}^d$, and $\partial\Omega$ is the boundary of Ω and Δ is the Laplace operator. We use a number of points $(\underline{x}_j, j = 1, \ldots, N_B)$ from the boundary $\partial\Omega$ and a number of points $(\underline{x}_j, j = N_B + 1, \ldots, N_B + N_I = N)$ from the domain Ω for collocation. Using straight collocation the RBF approximation of $u(\underline{x})$ is

$$s(\underline{x}, \epsilon) = \sum_{j=1}^{N} \lambda_j \phi(\|\underline{x} - \underline{x}_j\|, \epsilon).$$

Using the aforementioned points for collocation we end up with the following equations to satisfy.

$$\begin{aligned} s(\underline{x}_i, \epsilon) &\equiv \sum_{j=1}^{N} \lambda_j \phi(\|\underline{x}_i - \underline{x}_j\|, \epsilon) = g(\underline{x}_i) && i = 1, \ldots, N_B, \\ \Delta s(\underline{x}_i, \epsilon) &\equiv \sum_{j=1}^{N} \lambda_j \Delta\phi(\|\underline{x}_i - \underline{x}_j\|, \epsilon) = f(\underline{x}_i) && i = N_B + 1, \ldots, N. \end{aligned}$$

Which gives us a system of equations on the form

$$\left[\frac{\phi}{\Delta\phi}\right] [\lambda] = \left[\frac{g}{f}\right].$$

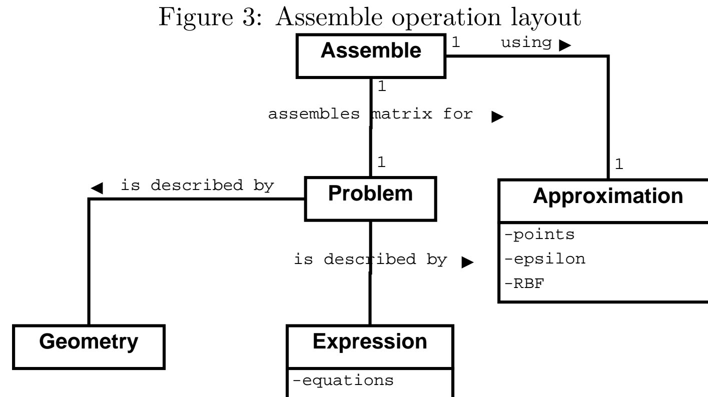
We want to calculate $[\lambda]$ so that we may use the approximation for arbitrary points $\underline{x}$. This means that we first have to calculate the matrix $\left[\frac{\phi}{\Delta\phi}\right]$ so that we then can solve $[\lambda] = \left[\frac{\phi}{\Delta\phi}\right]^{-1}\left[\frac{g}{f}\right]$.

An already written function `dphi()` allows us to calculate either $\phi(\|\underline{x}_i - \underline{x}_j\|, \epsilon)$ or $\Delta\phi(\|\underline{x}_i - \underline{x}_j\|, \epsilon)$ for each element of a given distance matrix containing the values of $\|\underline{x}_i - \underline{x}_j\|$.

4.1 Assemble operation

We add an assemble operation to the workflow manager for producing the $\left[\frac{\phi}{\Delta\phi}\right]$ matrix. The assemble operation will accept three input parameters: a problem object, an approximation object, and a name for the resulting matrix. The problem and approximation objects, and the objects they contain, are detailed in Sections 4.1.1, 4.1.2, 4.1.3, and 4.1.4.

Figure 3: Assemble operation layout



4.1.1 Approximation class

The approximation class contains information about the approximation method. In our case it contains the information that we use the Gaussian RBF, an epsilon object with our chosen shape parameter, and a point set used for collocation divided into several subsets. The subsets in our case are either interior or boundary.

4.1.2 Problem class

The problem class contains information about the problem. It consists of a geometry object and an expression object.

4.1.3 Geometry class

The geometry object contains information about how many subsets the problem is divided into and which equation each of these uses. In our case they use either of two equations either the interior or the boundary. This allows us to map a subset of points from the approximation to the equation they use.

4.1.4 Expression class

The expression class represents the PDE. It contains several named equations describing the PDE, in our case the interior and boundary equations. Given a subset of points we can from the geometry

object get the equation it uses and then from the expression get the operator, that is either the identity or Laplace operator, to use in the call to `dphi()`. A generic expression class was constructed in [2], but in the implementation of this case study we use a stub expression class which can only represent equation 1.

4.1.5 Implementation

The point set in the approximation is as mentioned above divided into several subsets. For each pair of subsets we need to first calculate the distance matrix and then using `dphi()` calculate a block of the final $\left[\frac{\phi}{\Delta\phi}\right]$ matrix. To allow us to test the task library's handles we decided to only calculate the distance matrix for the subset pair (B_i, B_j) and then use the transpose for (B_j, B_i). Handles to blocks of a matrix can be represented by declaring a matrix in the data manager and using its handle but never actually allocating it. This means that when we translate an assemble operation into tasks two types of tasks will be created. First $\frac{n(n+1)}{2}$, where n is the number of subsets, tasks that calculate distance matrices will be created. Secondly a total of n^2 tasks dependent on the distance matrices will calculate blocks of the final $\left[\frac{\phi}{\Delta\phi}\right]$ matrix. Finally, a single synchronization task is created which is used to output the result to a file for comparison to other known working implementations.

4.2 Implementation test

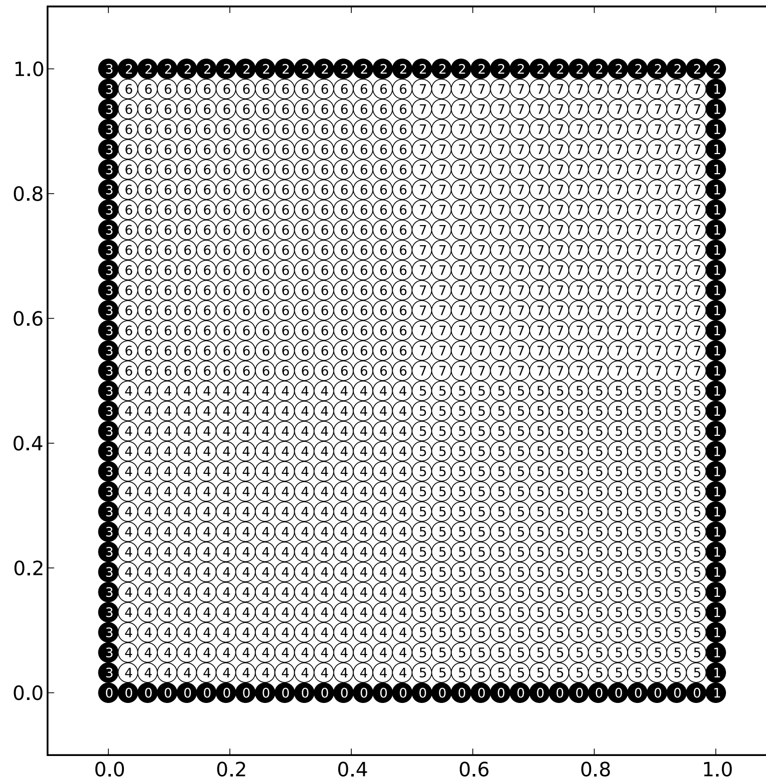
We test the implementation on the geometry illustrated in Figure 4 which is stored in the file "boxgeom1024.dat". As you can see the points are divided into eight subsets: four interior subsets with 225 points each; and four boundary subsets with 31 points each. The program code is shown in Listing 11. A serial version using the same method of calculating and transposing distance matrices as well as using the `dphi()` subroutine was constructed for comparison.

Listing 11: Using the assemble operation

```
program main

    use wfm_class

    use problem_class
    use approximation_class
    use class_point_set
    use fp

    implicit none

    type(wfm) :: w
    type(op), dimension(1)  :: ops

    type(problem)           :: prob
    type(geometry)          :: geom
    type(expression)        :: pde

    type(approximation)     :: approx
    type(epsilon)           :: eps

    call tl_init(4)

    ! 1: Subset uses the interior equation
    ! 0: Subset uses the boundary equation
    call geometry_new(geom, (/0,0,0,0,1,1,1,1/))

    call expression_new(pde)
```

```
    call problem_new(prob, geom, pde)

    eps = new_eps(.false., (/2.0_rfp, 2.0_rfp/), 1)
    call approximation_new(approx, "gauss", eps, "input/boxgeom1024.dat")

    call op_assemble(ops(1), prob, approx, "Result")

    call wfm_new(w, 1)
    call wfm_add(w, ops(1))
    call wfm_execute(w, (/ "Result" /))

    call tl_destroy()

    write(*,*) "Done."

end program main
```

Figure 4: Geometry



4.2.1 Results

The programs were run on the Uppsala University x86 server linne.it.uu.se. Since this server uses timesharing and is accessible by all students with a Unix account, the best of five runs was used to minimize disturbance from other users.

Runtime was measured for the program using the serial version as well as the task version using 1, 2, 4, and 8 threads (See A). Given the time to run the serial version T_{serial} and the time to run the task version on n threads relative speedup was calculated as $\frac{T_{\text{serial}}}{T_n}$ (See Figure 5).

Using functionality built into the task library, schedules of how tasks were run was also generated (See Figures 6, 7, and 8). The tasks doing distance calculations are colored pink while the tasks calculating blocks of the final matrix are colored blue.

4.2.2 Conclusions

Reasoning logically and ignoring hardware we should be able to reach linear speedup as the matrix is divided into blocks which can be calculated independently.

Looking at Figure 5 we can see that although speedup increases as the number of threads increases it is not as prominent as we would want to. While it is a first step towards an efficient working parallelization, in the future we will want to investigate in more detail how the speedup can be improved. Looking at Figure 6 and 7 we can see that the tasks are relatively tightly packed with only a small amount of dead time at the end so the load balance in these cases seems fine. In the case of eight threads (see Figure 8) we can see that the load balancing is getting worse and that we get some dead time on some of the threads in the beginning.

Comparing the runtime of the 16 largest tasks (corresponding to calculating `dphi()` for the larger interior domain pairs) in each schedule we can see that they take longer to complete as the number of threads increase. This is a probable cause of the nonlinear speedup and should be investigated further.

We can conclude that the unavoidable overhead when writing systems such as this seems to be small in the case of comparing the serial version to the one thread task version. Increasing overhead with increasing number of tasks might be another reason for the acquired speedup.

These tests were run using double precision. When we ran a few tests with quad precision we got notably worse speedup. This calls for further investigation.

5 Future work

Using the assemble operation we can construct the first matrix, $\left[\frac{\phi}{\Delta\phi}\right]$, needed in the effort to calculate λ. What remains to do, except improving the performance, is to construct operations for the remaining steps in calculating λ. This includes an operation for constructing the $\left[\frac{g}{f}\right]$ matrix as well as calculating $\lambda = \left[\frac{\phi}{\Delta\phi}\right]^{-1}\left[\frac{g}{f}\right]$. Other operations might also be interesting to add to the workflow manager.

Moreover, currently a stub expression class is used. A generic expression class was constructed in [2] which could be incorporated in the workflow manager system. The stub was used since the current generic expression class did not provide the necessary functionality and might have to be rewritten.

Performance is a third area that still needs work. As noted in Section 4.2.2 the speedup is of primary concern. We also noted in 3.1 that we settled for a list of matrices in the data manager. This should also be changed to a more efficient data structure. Moreover the performance degradation when using quad precision should be investigated.

Our goal is, as mentioned in [2], to construct a flexible and reusable framework for working with RBF approximations.

Figure 5: Relative Speedup

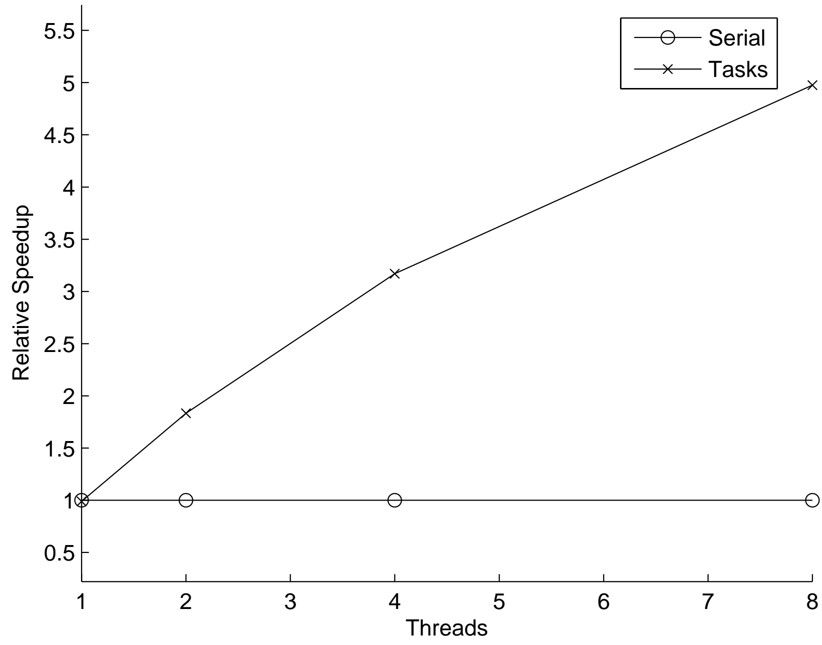


Figure 6: Task Schedule, 2 Threads

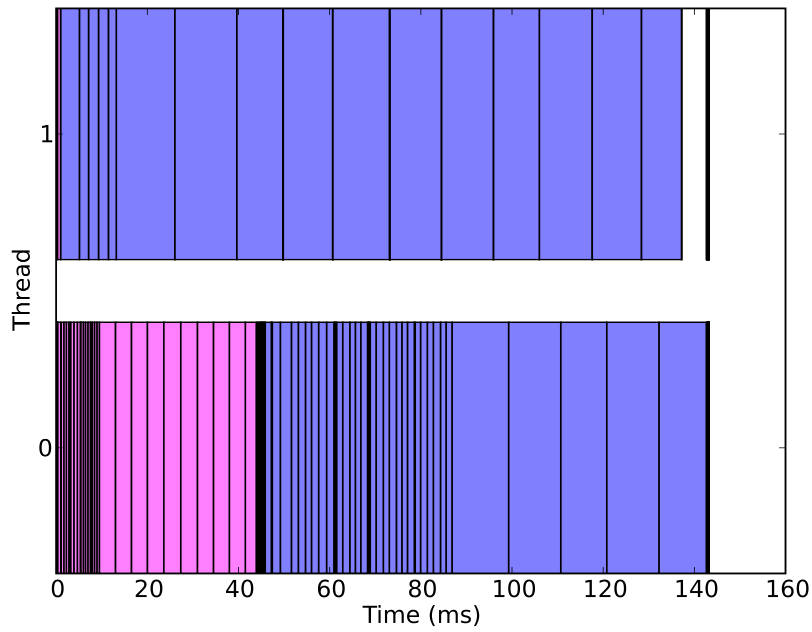


Figure 7: Task Schedule, 4 Threads

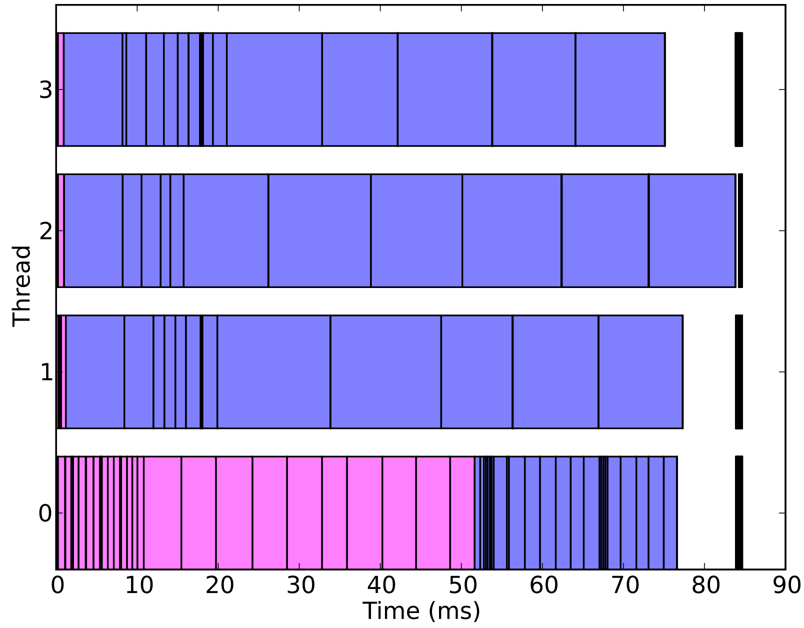
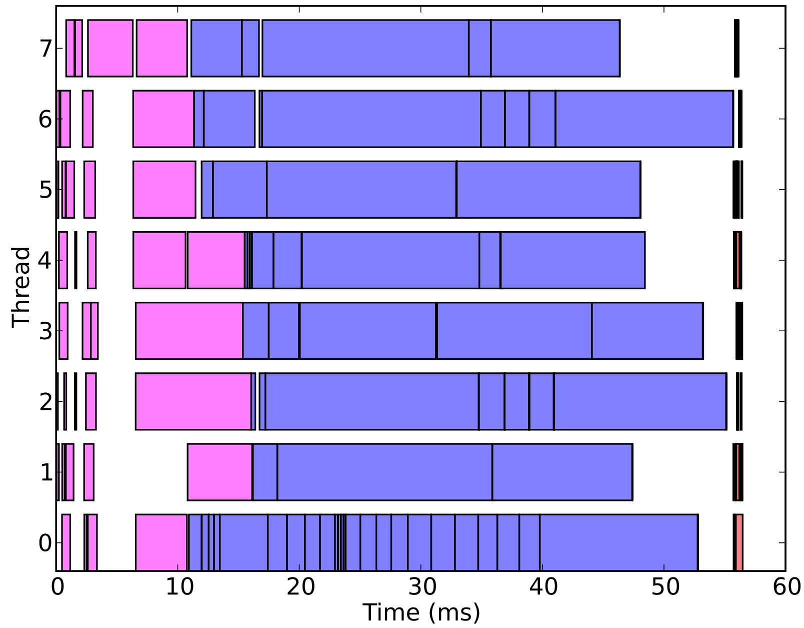


Figure 8: Task Schedule, 8 Threads



A Runtimes

Program	Run 1	Run 2	Run 3	Run 4	Run 5
Serial	266	266	266	266	266
1 Thread	269	269	269	269	269
2 Threads	145	157	157	157	158
4 Threads	85	86	84	85	86
8 Threads	55	60	53	54	56

Table 1: Runtimes in milliseconds

References

[1] G. F. Fasshauer. Meshfree Approximation Methods with MATLAB. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2007.

[2] Danhua Xiang. Designing a Flexible Software Tool for RBF Approximations Applied to PDEs. Student thesis (Master Programme in Computational Science), supervisor: Elisabeth Larsson, Martin Tillenius, examiner: Michael Thuné, Anders Jansson, IT nr 10 058, 2010.

[3] Martin Tillenius and Elisabeth Larsson. An efficient task-based approach for solving the n-body problem on multicore architectures. pages 74:1–4. University of Iceland, 2010.

[4] M. Gray, R. Roberts, and T. Evans. Shadow-object interfacing between Fortran 95 and C++. Computers in science and engineering, 1:63-70, 1999.

[5] M. Gray, and R. Roberts. Object-based programming in Fortran 90. Computers in Physics, 11:355-361, 1997.

[6] Sun Microsystems, Inc. Fortran 90 User's Guide. SunSoft 1995. Part No.: 801-5492-10

[7] Malcolm Cohen. Standard intrinsic module ISO_C_BINDING. Nihon Numerical Algorithms Group KK, Tokyo, Japan. http://www.nag.co.uk/nagware/np/r51_doc/iso_c_binding.html

[8] James Van Buskirk. http://groups.google.com/group/comp.lang.fortran/msg/875b87ae9f35c428

[9] V. Ganesh. Using Pthreads in Fortran. University of Pune, India.

[10] E. Larsson, and B. Fornberg. A Numerical Study of Some Radial Basis Function Based Solution Methods for Elliptic PDEs. Computers and Mathematics with Applications, 46:891-902, 2003.