Coding
IN
Fortran

Hongyang Zhou

$\mathcal{HY}$

September 30, 2020

Contents

Chapter 1

Overview

General reminder:

- The difference between theory and practice is less in theory than it is in practice.
- Don't reinvent the wheel. Use reliable softwares and libraries, if possible.
- Too many dependencies would kill your work.

1.1 Signature Features

`where` statement:

- arrays must have the same shape
- code block can contain

  Array assignments

  other where constructs

  forall constructs

Example: Stencil Update $A_i = 0.5 * (A_{i-1} + A_{i+1})$

```
1 v(2:n-1) = 0.5 * (v(1:n-2) + v(3:n))
```

Listing 1.1: Stencil update 1D

- Traditional scheme requires scalar variables
- Array syntax: Evaluate RHS, then copy the result

Example: Stencil Update $A_{i,j} = 0.25 * (A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1})$

```
1 a(2:n-1,2:n-1) = 0.25*(a(1:n-2,2:n) + a(3:n,2:n) + a(2:n,1:n-2) + a(2:n,3:n))
```

Listing 1.2: Stencil update 2D version 1

Or using forall statement:

```
1 forall (i=2:n-1, j=2:n-1) &
2 a(i,j) = 0.25*(a(i-1,j) + a(i+1,j) + a(i,j-1) + a(i,j+1))
```

Listing 1.3: Stencil update 2D version 2

- Fortran statement looks exactly like the original formula.

1.2 What's Falling Behind

1.2.1 Preprocessor

Conditional compilation has never really caught on in Fortran and there isn't a standard preprocessor. This limits the usage of many features, e.g., assertion.

1.2.2 MPI Standard Library

Can you imagine that there are three officially supported MPI library in Fortran? It is just because Fortran is old, and no one wants to touch the old code; so even if new features are added, like the argument checking in `mpi_f08`, very few codes are actually using it.

1.2.3 I/O

The general trend is that it is becoming more and more C-like. After Fortran 2003, you can even find the concept of stream!

From the benchmark done on the Julia Website, Fortran is 6 times slower in terms of writing to file.

1.2.4 Interfering with Command Line

Can you imagine that a simple task like mkdir is not still not available in Fortran? Fortran compiler vendors now provide a general interface to the command line, `execute_command_line`, to handle these kinds of problems.

1.2.5 Coroutines

It may once be a good attempt, but end in no where, because no one is actually using this built-in parallel feature.

1.3 Variables

There are several things you need to pay attention with regard to variables.

1.3.1 Variable Scope

Module variables have the intrinsic `save` attribute, which means that they will be allocated only once and stay there until the end of your program. In some programming languages, it is equivalent to `static`. A clear usage of `use` statement helps you organizing the available scopes. It is nice to add `ONLY` to limit your data scope.

Things get a little bit complicated with OpenMP included, but still, there are rules to follow.

1.3.2 Allocations

Local variables within functions are allocated on the stack (first in, first out), while dynamic variables are allocated on the heap.

Potential issues with variable allocation on the heap:

- The size of the stack is severely limited. (Check the specific default!)
- Remedies are problematic (Intel: -mcmodel=medium -intel-shared)

1.3.3 Garbage Collection

Modern compilers all have automatic garbage collections, so in general we as programmers don't need to worry about deallocating arrays. Explicit deallocating user-defined large arrays that won't be used later on is always not bad. However, there are cases in which you don't want to deallocate intermediate arrays that will be used again in a loop. Mark them with `save` and allocate them only once.

1.4 Modules

Modules stay in the key concept of Fortran90+. After Fortran 2008: Modules may contain Submodules. Modules have to be compiled first, with each results in a `.mod` file.

At linking time, the module is known through the `.mod` file.

What to put in a Module:

1. Constants (parameters)
2. Derived type declarations

   avoid repeating parameter and derived type definitions. Sometimes physical constants are put in an include file. This should be done using a module.
3. Global variables
4. Related functions and subroutines.

   move on by using the public, private and protected attributes
5. Write Object-Oriented code without performance penalty
6. Use Inheritance and Polymorphism with care

1.4.1 Compiling Modules

The module name need not be the file name; doing that is strongly recommended, though.

Now you compile it , but don't link it:

```
nagfor -C=all -c mymod.f90
```

It will create files like *mymod.mod* and *mymod.o*. They contain the interface and the code. The interfaces are like compiled headers. You dont do anything with these, explicitly: the compiler will do find them and use them.

To add another search path , use -I< *directory* >.

Compile all of the modules in a dependency order. If A contains USE B , compile B first. Then add a *.o for every module when linking:

```
nagfor -C=all -o main main.f90 mod_a.o mod_b.o mod_c.o
```

1.4.2 Use Statements

Proposed syntax

```
USE [[, module-nature] ::] module-name [AS use-name] [, ONLY:only-list]
```

Without AS use-name it is a regular USE statement. However, when the AS use-name is used, then the entities defined in module module-name are accessed only with use-name%entity-name syntax.

1.5 Miscellaneous

Benefits:

- Allows consistency check by the compiler
- Assume-shape arrays, optional parameters, etc.

intent: (in/out, inout) maintainability

optional

any, all

Using generic interfaces for functions

User-defined operators

Elemental function used with *all* and *where*: similar to bitwise operations in Matlab

Inquiry functions (see slides)

Pointers: useful for creating "linked lists"; avoid copying data

Command line arguments

get environment variable inside Fortran (e.g. home directory)

Pure function is used to prevent any input arguments being modified, regardless of the use of `intent` statement.

Elemental function is used to create procedures that operate parameters of arbitrary dimension.

1.6 Naming Standard

> There are only two hard things in Computer Science: cache invalidation and naming things. — Phil Karlton

Good variable names are crucial, especially in dynamically typed languages, where you don't have types defined to help you make sense of the variables. However, even statically typed languages will have the benefit of better readability when good naming conventions are used.

1.6.1 Arrays

Arrays are an iterable list of items, usually of the same type. Since they will hold multiple values, pluralizing the variable name makes sense.

```
// bad
const fruit = ['apple', 'banana', 'cucumber'];
// okay
const fruitArr = ['apple', 'banana', 'cucumber'];
// good
const fruits =  ['apple', 'banana', 'cucumber'];
// great
const fruitNames =  ['apple', 'banana', 'cucumber'];
// great
const fruits = [{
   name: 'apple',
   genus: 'malus'
}, {
   name: 'banana',
   genus: 'musa'
}, {
   name: 'cucumber',
   genus: 'cucumis'
}];
```

Listing 1.4: Array names

1.6.2 Booleans

Booleans can hold only 2 values, `true` or `false`. Given this, using prefixes like "is", "has" and "can" will help the reader infer the type of the variable.

```
// bad
const open = true;
const write = true;
const fruit = true;

// good
const isOpen = true;
const canWrite = true;
const hasFruit = true;
```

Listing 1.5: Booleans names

What about predicate functions (functions that return booleans)? It can be tricky to name the value, after naming the function. Usualy prefix with either `check` or `get` would be nice.

1.6.3 Numbers

For numbers, think about words that describe numbers.

1.6.4 Functions

Functions should be named using a verb, and a noun. When functions perform some type of action on a resource, its name should reflect that. For example, `getUser`, `toUppercase`.

Another common naming pattern is when iterating over items. When receiving the argument inside the function, use the singular version of the array name:

```
const newFruits = fruits.map(fruit => {return doSomething(fruit)});
```

Listing 1.6: Iterating over items

1.6.5 Blocks

Block works like a local scope, similar to brackets in C/C++. You can define local variables that won't interfere with others.

1.6.6 Summing up

Take these naming convections with a pinch of salt. How you name your variables is less important than naming them consistently.

1.7 Interoperatability

1.7.1 Mixture with C

A Fortran90+ program, subroutine, or function that will call a C function might try using the ISO C binding module. This was actually introduced as part of Fortran 2003, but your compiler may be willing to let your Fortran90 program access it. (If not, you might consider moving to Fortran 2003!).

The ISO C bindings are made available by the statement:

```
use iso_c_binding
```

You can also use fussier versions of this statement, such as

```
use, intrinsic :: iso_c_binding
```

or

```
use, intrinsic :: iso_c_binding, only : C_CHAR, C_NULL_CHAR
```

Once you have the C bindings, you need to define an interface to your C function, which might read:

```
interface
    subroutine kronrod ( n, eps, x, w1, w2 ) bind ( c )
      use iso_c_binding
      integer ( c_int ), VALUE :: n
      real ( c_double ), VALUE :: eps
      real ( c_double ) :: x(*)
      real ( c_double ) :: w1(*)
      real ( c_double ) :: w2(*)
    end subroutine kronrod
```

```
10 end interface
```

Listing 1.7: C interface

Finally, to guarantee that FORTRAN and C agree on data types, you should declare any Fortran90 variables that will be passed through the C interface with statements like this, which essentially specify the appropriate KIND parameter to guarantee compatibility:

```
1  integer ( c_int ), parameter :: n = 3
2  real ( c_double ) eps
3  real ( c_double ) x(n+1)
4  real ( c_double ) w1(n+1)
5  real ( c_double ) w2(n+1)
```

Listing 1.8: C interface

Note:

1. If you use subroutines in Fortran, set the return value of C functions to void, and use pointers because Fortran always passes by reference.
2. If you use functions in Fortran, set the return value as it is.

1.7.2 Mixture with C++

Much similar to C, except a special requirement when C++ is involved is that the extern "C" qualifier be used with all the function names that must be "visible" to the Fortran90 program. For this example, we simple declare all the C++ functions this way:

```
extern "C" {
  long long f_(long long *a, long long *b, long long *c, long long *d);
  void ftest_(int *a, int *b);
}
```

1.7.3 Mixture with CUDA

See this demo.

1.8 Code Development

using FORD (FORtran Documenter) to build html: FORD

I need to look deeper into this app for a better understanding of the structure of BATSRUS. It will also be useful for a diagram showing the structure of a complex code. Ford is written in Python and is target specifically at Fortran.

Graphviz can be used to generate graph descriptions of the program.

1.9 Attribute

A variable with the SAVE attribute retains its value and definition, association, and allocation status on exit from a procedure. All variables accessible to a main program are saved implicitly.

Optimization	stack array	pointer	heap array	no small array
-O0	0.998 s	1.829 s	1.120 s	1.045 s
-O2	0.928 s	1.519 s	0.940 s	0.884 s
-O3	0.886 s	0.853 s	0.884 s	0.839 s
-Ofast	0.890 s	0.868 s	0.889 s	0.848 s

A local variable that is initialized when declared has an implicit SAVE attribute. Explicit initialization of a module variable that is not in a common block implies the SAVE attribute.

Variables don't need to be deallocated by the programmer. The garbage collection technique handles this.

When intent(out) is used with a derived type, any component not assigned in a procedure could become undefined on exit. For example, even though a%y was defined on entry to this routine, it could become undefined on exit because it was never assigned within the routine.

As an alternative, use intent(inout) to avoid the issue.

1.10 Pointer

Does pointer affect performance? I have a simple test program for showing the differences between using pointers, stack allocated arrays, and heap allocated arrays. The results show that:

With higher level of optimization, pointer performs better than local copies!

1.11 Debug

I once encountered a very strange bug in a serial Fortran 77 code. It behaved like a local unused variable will cause the program to give different output.

Chapter 2

Object Oriented Programming

2.1 Introduction

OOP on access control

OOP may be slower (if not written carefully)

You can organize your data by defining your own structures and derived types. A good usage of user defined objects will simplify your program a lot. As poor Fortran people we are used to simply translate formula, we do not care about paradigms What Paradigm is the following?

```
subroutine solr(toll,Qm,gamma1,a1,u1,p1,rho1,gamma4,a4,u4,p4,rho4,u,p,a,fF)
  real*8 a2,p2,rho2,a3,p3,rho3,ps,us
  real*8 pn,pn_1
  real*8 a,u,p
  real*8 w1,w4,toll,Qm
  real*8 delta,gamma,gm1
  ...
  gm1=gamma-1.d0
  delta=gm1/2.d0
  alfa=(p1/p4)**(gm1/(2.d0*gamma))
  beta=alfa*delta/a1+delta/a4
  ...
```

Listing 2.1: Procedural programming style

Aside from being very ugly, the above example shows that data is **highly decoupled**. This is typical of **procedural programming** paradigm:

- TOP to down approach
- focus on the operations, not on the data
- data handling unsafe, GLOBALS are often necessary
- hard to add new functionality or change the work flow without going back and modifying all other parts of the program
- not easily reusable, so programmers must often recreate the wheel
- difficult to maintain

Change your point of view!

- down to TOP approach

- key idea: *The real world can be accurately described as a collection of **objects** that interact*

How-To-OOP, the Pillars

- Encapsulation;
- Inheritance;
- Polymorphism.

2.1.1 Encapsulation

Encapsulation is about grouping of functionality (methods) and related data (members) together into a coherent data structure, **classes**.

Implementation is hidden from the rest of the program, aka Application Program Interface.

Do you need to know exactly how every aspect of a car works (engine, carburettor, alternator)? No — you only need to know how to use the steering wheel, brakes, accelerator

In practice,

- restrict access to data (and data hiding):
- more control;
- easy to debug;
- easy to maintain;
- avoid side effects;
- easy to parallel.

The Consequences:

- design by-contract
- modularize;
- abstraction;
- develop by tests.

Example: gas_t

So, where is encapsulation? The user of **gas** class is not aware of how to compute the speed of sound. He/she simply uses the speed of sound; cannot manipulate gas data directly a bad user cannot make density negative manually; can re-use the same names defined into **gas_t** module: encapsulation allow easy handling of names-collision (safety names space).

2.1.2 Inheritance

Inheritance enables new objects to take on the properties of existing objects, the **base class**. A child inherits visible properties and methods from its parent while adding additional properties and methods of its own.

In practice,

- re-use existing classes:

- reduce code-duplication:
- easy debug;
- easy maintain;
- inherit members/methods;
- re-use implementation;
- classes hierarchy;
- abstract is better, but specials happen
- specialize only when/where is necessary.

The Consequence

- design from down to up
- from very down! Abstract all!
- specialize ⇒ inherit from abstract to concrete cases
- avoid to re-invent the wheel

So, where is inheritance? The development of **multifluid gas** class. We do not need to re-implement all the behaviors of gas class: multifluid is a special gas avoid-re-invent the wheel; While the user of **multifluid gas** class found the same facility of the previous gas class: avoid to re-learn from scratch; could re-use his/her own applications designed for gas objects

2.1.3 Polymorphism

Polymorphism is the concept that multiple types of objects might be able to work in a given situation. If you needed to write a message on a piece of paper, you could use a pen, pencil, marker or even a crayon. You only require that the item you use can fit in your hand and can make a mark when pressed against the paper.

In other words it means, one method with multiple implementation, for a certain class of action. And which implementation to be used is decided at both runtime/compiletime (dynamic/static). Overloading is static polymorphism while, overriding is dynamic polymorphism.

In practice,

- design for family of classes (objects);
- exploit inheritance;
- exploit encapsulation.

The Consequences:

- be ABSTARCT!

2.1.4 High level abtract programming

2.2 Basics

Object Orientation concepts:

- Encapsulation:

  Hiding the details of data representation behind some interface.

- Inheritance:

  Allows us to customise existing classes to suite our purposes.

- Polymorphism:

  Allows us to make methods and classes which will work on any of a range of different data types, even types which did not exist when the library was written.

Besides, there is also nested classes: a class can be defined inside another class.

2.3 Encapsulation

How do we chose objects? One approach:

1. Write down a description of the problem, then underline all the nouns.
2. Simple things may become properties of objects (i.e. Member variables).
3. More complex things may become objects.
4. Others are abstract or uninteresting or processes and will be ignored or take other forms.

2.4 Inheritance

Inheritance provides an efficient way to reuse code. If we want to make a new class which is similar to an existing class, we can use inheritance to do this without touching the original class. The new class is called a subclass or derivedclass. It inherits from its superclass, or baseclass. The new class has all the members and methods of the parent class, plus any more that are defined.

Properly used, inheritance aids code reuse: first define base classes which are generic.; then derive more specific classes, have more details.

Example

```
Class Atom
{ double x,y,z,occ; };
Class Atom_isotropic : public Atom
{ double u_iso; };
Class Atom_anisotropic : public Atom
{ double u11,u22,u33,u12,u13,u23; };
```

At first, spotting how to use inheritance can be tricky. One (inefficient) approach while learning inheritance:

1. Write all the specific classes (using cut and paste for any shared code).
2. Look for any members and method code which can be shared.
3. Implement a base class containing the common features.

Even for experienced programmers, class design evolves after the first implementation.

2.5 Polymorphism

Polymorphism is the ability of classes, methods, and functions to work on a range of different data types, even types which did not exist when the library was written.

Two forms:

- Runtime polymorphism: You can use a derived class wherever you can use its base class.
- Templates [C++/Java only]: You can write template classes and functions which can take any type of data.

2.5.1 Runtime

There are (small) memory and performance overheads for runtime polymorphism.

Suppose our Atom class implements a method `density_at_xyz(x,y,z)`, for a stationary atom. `Atom_isotropic` and `Atom_anisotropic` will override this method with methods appropriate to atoms with thermal motion. A method for calculating electron density might take a list of atoms, not caring what type of atom is involved. A later developer may then add another atom type (e.g. `Atom_disordered`) with a clever method for calculating density. The electron density calculation will still work!

Therefore in C++, runtime polymorphism only occurs for classes which have virtual methods, and then only those methods of a class which are explicitly declared as virtual (i.e. Can be overridden). Polymorphism only occurs when handling a reference (or pointer) to a class which contains virtual methods.

2.5.2 Templates

Templates are a second form of polymorphism, implemented in C++, which has no performance or memory overheads. Template polymorphism occurs at compile time. It works on any class (which has the right sort of API), whether or not there is an inheritance relationship involved.

Example: `std::vector`: a resizeable array of data of some type.

- Incredibly useful: use it whenever you want an array whose size isnt absolutely immutable.
- Makes memory allocation (and therefore memory leaks) obsolete.
- There are also a whole range of related types, e.g. Singly and doubly linked lists, associative arrays, etc.

```
int n = 10;
std::vector<int> i;
std::vector<float> f( 6 );
std::vector<double> d( n, 1.0 );
std::vector<Cell> c;

// get the current size
int old_size = d.size();
// resize the list
d.resize( 20 );
```

```
// sum the values in a list
double sum = 0;
for ( int i = 0; i < d.size(); i++ )
sum = sum + d[i];

// add to the end of the vector
d.push_back( 3.142 );
// remove from the end of the vector
double x = d.pop_back();

// insert at 3 from the front
d.insert( d.begin()+3, 2.718 );
// delete from three from the end
d.delete ( d.end()-2 );

// sort the list
std::sort( d.begin(), d.end() );
```

- The type of data comes in angle brackets <> after the class name. This tells the compiler to compile a vector for that type of member.
- The data type can be any builtin or user defined type or class.
- We can optionally define initial size, and value.
- The `std::vector` class has a method size which returns the current size of the list.
- The `std::vector` class has a method resize which changes the current size of the list.
- The `std::vector` class has a method which looks like standard array subscription (i.e. overrides the bracket operator []), which allows us to get at the data is if it were in a normal array.
- The `std::vector` class has methods which add to or remove from the back of an array. Performance is good it is not uncommon to build up a large array one element at a time.
- The `std::vector` class has methods to insert and deletes at arbitrary positions in the list. (There are performance overheads of course a linked list may be better).
- The `std::sort` algorithm is the most efficient algorithm known. We usually want to sort by key: Make `std::vector<std::pair<keytype,datatype> >` containing the list of keys and data, and apply a `std::sort`.

Chapter 3

Intrinsic Parallelism

3.1 Co-Array

Co-Array Fortran (CAF) is a simple parallel extension to Fortran 90/95. The coarrays feature of Fortran 2008 provides a Single Program Multiple Data (SPMD) approach to parallelism, which is integrated into the Fortran language for ease of programming. An application using coarrays runs the same program, called an image, in parallel, where coarray variables are shared across the images in a model called Partitioned Global Address Space (PGAS).

1. It uses normal rounded brackets ( ) to point to data in local memory.
2. It uses square brackets [ ] to point to data in remote memory.
3. Syntactic and semantic rules apply separately but equally to ( ) and [ ].

What Do Co-dimensions Mean?

A CAF Image is a process Processes have NO data sharing by default separate memory maps.

The declaration

```
real :: x(n)[p,q,*]
```

means

1. An array of length n is replicated across images.
2. The underlying system must build a map among these arrays.
3. The logical coordinate system for images is a three dimensional grid of size.
4. (p, q, r) where $r = num\_images()/(pq)$.

Coarrays are used to split the trials across multiple copies of the program. They are called images. Each image has its own local variables, plus a portion of any coarrays shared variables. A coarray can be a scalar. A coarray can be thought of as having extra dimensions, referred to as codimensions. To declare a coarray, either add the CODIMENSION attribute, or specify the cobounds alongside the variable name. The cobounds are always enclosed in square brackets. Some examples:

Examples of Co-Array Declarations:

```
real :: a(n)[*]
```

```
real :: b(n)[p,*]
real :: c(n,m)[p,q,*]
complex,dimension[*] :: z
integer,dimension(n)[*] :: index
real,allocatable,dimension(:)[:] :: w
type(field), allocatable,dimension[:,:] :: maxwell
real, dimension(100), codimension[*] :: A
integer :: B[3,*]
```

3.2 Do Concurrent

This has not been proved to be useful (2018), according to the expert Vladimir on StackOverflow.

Chapter 4

Compilation

From stack overflow by jalf:

FLOPS is, as the name implies FLoating point OPerations per Second, exactly what constitutes a FLOP might vary by CPU. (Some CPU's can perform addition and multiplication as one operation, others can't, for example). That means that as a performance measure, it is fairly close to the hardware, which means that 1) you have to know your hardware to compute the ideal FLOPS on the given architecture, and you have to know your algorithm and implementation to figure out how many floating point ops it actually consists of.

In any case, it's a useful tool for examining how well you utilize the CPU. If you know the CPU's theoretical peak performance in FLOPS, you can work out how efficiently you use the CPU's floating point units, which are often one of the hard to utilize efficiently. A program which runs 30% of the FLOPS the CPU is capable of, has room for optimization. One which runs at 70% is probably not going to get much more efficient unless you change the basic algorithm. For math-heavy algorithms like yours, that is pretty much the standard way to measure performance. You could simply measure how long a program takes to run, but that varies wildly depending on CPU. But if your program has a 50% CPU utilization (relative to the peak FLOPS count), that is a somewhat more constant value (it'll still vary between radically different CPU architectures, but it's a lot more consistent than execution time).

But knowing that "My CPU is capable of X GFLOPS, and I'm only actually achieving a throughput of, say, 20% of that" is very valuable information in high-performance software. It means that something other than the floating point ops is holding you back, and preventing the FP units from working efficiently. And since the FP units constitute the bulk of the work, that means your software has a problem.

It's easy to measure "My program runs in X minutes", and if you feel that is unacceptable then sure, you can go "I wonder if I can chop 30% off that", but you don't know if that is possible unless you work out exactly how much work is being done, and exactly what the CPU is capable of at peak. How much time do you want to spend optimizing this, if you don't even know whether the CPU is fundamentally capable of running any more instructions per second?

It's very easy to prevent the CPU's FP unit from being utilized efficiently, by having too many dependencies between FP ops, or by having too many branches or similar preventing efficient scheduling. And if that is what is holding your implementation back, you need to know that. You need to know that "I'm not getting the FP throughput that should be possible, so clearly other parts of my code are preventing FP instructions from being available when the CPU is ready to issue one".

I found an interesting presentation on the Internet: "Do theoretical FLOPs matter for real application's performance?" This is done by an engineer from AMD who uses AMD Interlagos processors to demonstrate that theoretical FLOP/s isn't a good indicator of how applications such as CFD will perform. The key points from his presentation:

1. Most of CFD apps with Eulerian formulation use sparese linear algebra to represent the linearized Navier-Stokes equations on non-structured grids.
2. The higher the discretization schemes, the higher the arithmetic intensity.
3. Data dependencies in both spatial and time prevent vectorization.
4. Large datasets have low cache reutilization.
5. Cores are waiting most of the time to get new data into caches. Once data is on the caches, the floating point instructions are mostly scalar instaed of packed.
6. Compilers have hard time in finding opportunities to vectorize loops. Loop unrolling and partial vectorization of independent data help very little due to cores waiting to get that data.
7. Overall, CFD has low performance from FLOP/s point of view.

4.1 Compilers

By default, nagfor will assign NaN to uninitialized variables, while gfortran will assign 0 to them. There are options of choosing initialization value for most compilers.

gfortran on Mac is different from gfortran on Linux. Based on what I see during hybrid testing, gfortran on Linux performs much better.

4.2 Common Optimization Techniques

4.2.1 Inlining

Save function overload by replacing the function call statement with the function code itself ((process called expansion). It may increase the executable size. If target function is called from another module, then inlining can only be done at link time optimization. However, some compilers does not support this feature (e.g. pgfortran).

4.2.2 Vectorization

"Vectorization" (simplified) is the process of rewriting a loop so that instead of processing a single element of an array N times, it processes (say) 4 elements of the array simultaneously $N/4$ times.

The difference between vectorization and loop unrolling: Consider the following very simple loop that adds the elements of two arrays and stores the results to a third array.

```
1 for (int i=0; i<16; ++i)
2     C[i] = A[i] + B[i];
```

Listing 4.1: simple loop

Unrolling this loop would transform it into something like this:

```
for (int i=0; i<16; i+=4) {
    C[i]   = A[i]   + B[i];
    C[i+1] = A[i+1] + B[i+1];
    C[i+2] = A[i+2] + B[i+2];
    C[i+3] = A[i+3] + B[i+3];
}
```

Listing 4.2: Loop unrolling

Vectorizing it, on the other hand, produces something like this:

```
for (int i=0; i<16; i+=4)
    addFourThingsAtOnceAndStoreResult(&C[i], &A[i], &B[i]);
```

Listing 4.3: Vectorization

Where "addFourThingsAtOnceAndStoreResult" is a placeholder for whatever intrinsic(s) your compiler uses to specify vector instructions. Note that some compilers are able to auto vectorize very simple loops like this, which can often be enabled via a compile option. More complex algorithms still require help from the programmer to generate good vector code.

4.2.3 Prefetching

Prefetching means making data available in the cache before the data consumer places it request, thereby masking the latency of the slower data source below the cache. (Obviously branch prediction lies in this category.)

4.2.4 Compiler directives

- !dir$ concurrent
- !dir$ ivdep
- !dir$ interchange
- !dir$ unroll
- !dir$ loop_info [max_trips] [cache_na] ... Many more
- !dir$ blockable

4.2.5 Patches

It is good for preformance to avoid leftmost array dimensions that are a power of 2 for multi-dimensional arrays where access to array elements will be noncontiguous. Since the cache sizes are a power of 2, array dimensions that are also a power of 2 may make inefficient use of cache when array access is noncontiguous. If the cache size is an exact multiple of the leftmost dimension, your program will probably make use of the cache less efficient. This does not apply to contiguous sequential access or whole array access.

One work-around is to increase the dimension to allow some unused elements, making the leftmost dimension larger than actually needed. For example, increasing the leftmost dimension of A from 512 to 520 would make better use of cache:

```
REAL A (512,100)

DO I = 2,511
  DO J = 2,99
    A(I,J)=(A(I+1,J-1) + A(I-1, J+1)) * 0.5
  END DO
END DO
```

Listing 4.4: Patch example

In this code, array A has a leftmost dimension of 512, a power of two. The innermost loop accesses the rightmost dimension (row major), causing inefficient access. Increasing the leftmost dimension of A to 520 (REAL A (520,100)) allows the loop to provide better performance, but at the expense of some unused elements.

Because loop index variables I and J are used in the calculation, changing the nesting order of the DO loops changes the results.

4.3 Compiler Info

Table 4.1: Cray, Intel and GNU compiler flags

Feature	Cray	Intel	GNU
Listing	-hline=a	-opt-report	-fdump-tree-all
Vectorization	-O1 and above	-O2 and above	-O3 or using -ftree-vectorize
Inter-procedural optimization	-hwp	-ipo	-flto
Floating-point optimizations	-hfpN, N=0...4	-fp-model [fast—, fast=2—precise—, except—strict]	-f[no-]fast-math -funsafe-math-optimizations
Suggested optimizations	(default)	-O2 -xHost	-O2 -mavx -ftree-vectorize ffast-math -funroll-loops
Aggressive optimization	-O3 -hfp3	-fast	-Ofast -mavx -funroll-loops
OpenMP	(default)	-qopenmp	-fopenmp
Variable sizes	-s real64	-real-size 64	-freal-4-real-8
	-s integer64	-integer-size 64	finteger-4-integer-8

Based on my experience, inter-procedural optimizations in gfortran do not work for BATSRUS. Other flags, for example, the flag `-ip` and `-ipo` in ifort works and give better performance.

The link-time optimization (i.e. IPO) is a step that examines function calls between files when the program is linked. This flag must be used to compile and when linking. Compile times may become very long with this flag, however depending on the application there may be appreciable performance improvements when combined with the -O* flags. This flag and any optimization flags must be passed to the linker, and gcc/g++/gfortran should be called for linking instead of calling ld directly.

All the slow down when turning on OpenMP flag for ifort is due to `ModFaceFlux`. I should really dig into the optimization report and figure out why.

Register instruction matters. AVX and SSE instruction sets are the two most popular one.

At least on a Cray machine, gfortran compiles reasonably fast compared with ifort.

`ModWritePlot` takes time to compile.

On Stampede2, I can compile BATSRUS with ifort $-O3$ flag. It gives slightly better results than $-O2$. With `-ipo`, I can get the best performance. Using `-Ofast` is slower than using `-O3 -ipo`. Notice that hyperthreading is turning on by default, and it really looks strange to me. I don't fully understand how hyperthreading works.

On Stampede2, the results for no optimizations (`-O0`) surprise me. gfortran is 3 times faster than ifort! Without OpenMP flags, ifort runs 20% faster. Using multiple threads causes the timing from proc 0 to have very strange behavior: a significant amount of time is spent nowhere! The built-in `time` command shows that with 2 threads each CPU utilizes $\sim 150\%$. Theoretically this should be $\sim 200\%$!

4.3.1 Hyper-threading

Hyper-threading has been introduced by Intel back in 2002, but it's actual performance has always been a mystery. While this technique claims to be helpful in process schduling and reducing idle time for the CPUs, nowadays BIOS and operating systems like Linux are usually good enough to organize processes so there's not much space for hyper-threading. In fact, in the area of high performance computing, hyper-threading is usually reported to be detrimental to performance.

Some major findings from experienced programmer:

- Hyper-Threading made no difference under Linux (results were the same with it on or off).
- Windows 2008 R2 with Hyper-threading on showed very poor thread scaling. Disabling Hyper-Threading in the BIOS greatly improved the parallel performance with Windows 2008 R2.
- Overall performance on Linux was better than Windows.

4.3.2 Tools and Commands

Check Linux pseudo files and confirm the system details

- `cat /proc/cpuinfo` >> provides processor details
- `cat /proc/meminfo` >> shows the memory details
- `/user/sbin/ibstat` >> shows the interconnect IB fabric details
- `/sbin/sysctl -a` >> show details of system (kernel, file system)
- `/usr/bin/lscpu` >> shows cpu details including cache size
- `/usr/bin/lstopo` >> shows the hardware topology
- `/bin/uname -a` >> shows the system information
- `/bin/rpm -qa` >> shows the list of installed products including versions
- `cat /etc/redhat-release` >> shows the redhat version
- `/usr/sbin/dmidecode` >> shows system hardware and other details (need to be root)
- `/usr/bin/numactl` >> checks or sets NUMA policy for processes or shared-memory
- `/usr/bin/taskset` >> shows cores and memory of numa nodes of a system

4.3.3 Practical Tips for Boosting Performance

- Check the system details thoroughly (Never assume!)
- Choose a compiler and MPI to build your application (All are not same!)
- Start with some basic compiler flags and try additional flags one at a time (Optimization is incremental!)
- Use the built-in libraries and tools to save time and improve performance (Libs & tools are your friends!)
- Change compiler and MPI if your code fails to compile or run correctly (Trying to fix things is futile!)
- Test your application at every level to arrive at an optimized code (Remember the 80-20 rule!)
- Customize your runtime environment to achieve desired goals.
- Always place and bind the processes and threads appropriately (Life saver!)
- Gather, check and correct your runtime environment.
- Profile and adjust optimization and runtime environments accordingly.

Chapter 5

BATSRUS

5.1 General

Usually CFD codes are memory-bound. You would rarely see a finite-difference or finite-volume code has a CPU efficiency more than 20%. No exception for MHD codes like ours. In that case, it makes much less sense to save intermediate variables to improve performance! In fact, some CFD codes, for example Clover Leaf, scarifies memory to improve performance.

With all the current modifications, my Ganymede steady state run has a 5% speedup.

5.1.1 Unknowns

1. Why does `update_state_normal` return if `time_accurate .and. Dt == 0.0`? Isn't that a waste?
2. A full implicit run can give slightly different results when using different number of processors (e.g gray diffusion test). This is the observation for GMRES. Don't know about BICGSTAB. Stiffness may be taken into consideration.
3. What is `advance_localstep` in `ModLocalstep` ? It seems to be related to subcycling.
4. What is EEE?
5. If we are using semi-implicit scheme for hall term, why is init called twice?
6. Are there any communications between the hall regions?
7. gammawave: what is this?
8. There is an issue for the output `dx` in logrithmic spherical coordinates. Ask Gabor and fix it.
9. `split_strings` in `ModUtility` uses too much memory. From one performance test, I remember Fortran is not good at parsing intergers.
10. For multifluid cases, only the hydrodynamics part needs individual fluid species.
11. There is a reason for doing flux conservation in the current way. For the source terms calculations, e.g. $-p\nabla \cdot \mathbf{u}$, the flux term can be corrected during message passing, but it has to be multiplied by a coefficient which is related again to the source term at cell centers. Therefore it is usually better to do it beforehand (like in `ModAdvanceExplicit` for the conservative fluxes).

12. In saving outputs, there are many tricks. One of them is how to save AMR data in regular .out format without the connectivity information. Checkout the DxSaveOutput=0 and DxSaveOuput=-1, especially when AMR is on.

5.1.2 Issues

The multi-fluid MHD module, which serves as an extension to the single fluid MHD, is introduced into BATS-R-US several years ago. The extension from single fluid MHD to multi-fluid MHD causes performance penalties for the original single fluid runs because of unnecessary string copies and function calls for setting up the state for different fluid species. This part is now being optimized by preventing useless operations to give a 40% improvement on the single fluid simulations.

Because of the integrity of all kinds of numerical methods for solving the face flux for the Riemann problem, each parameter is set individually without interfering with each other for the flexibility of adding new schemes. However, the old `FaceFlux` module sets all the parameters repeatedly during each timestep, which is modified to be done only once in the initial timestep to avoid redundant computation.

Temporary arrays are created when we try to pass slices of a large array which are strided in memory into another function or subroutine. This potentially slows down the code. We avoid temporary arrays in most cases by allocating local arrays and pass as assumed shape array.

Using Fortran modules without the `ONLY` keyword inherits all the public variables and subroutines from the module, which may cause unexpected bugs in the program because of duplicate names. We avoid the unwanted behavior by always listing the public attributes and using `ONLY` keyword to import variables and functions as needed.

I removed `UsePoleDiffusion` from `ModFaceFlux`.

Decomposing state in `ModCharacteristic.f90` for Roe flux is spending a lot of time. No wonder it is the case.

Impact of multi-fluid module to single-fluid runs

There are performance penalties for multi-fluid runs with the old implementation. In `ModFaceFlux`, there are many places like `select_fluid` but the function is not being called. I removed unnecessary string copy in `select_fluid`; return if $nFluid = 1$. This is a significant improvement: by implementing some of the above optimization, Flops/s increases from 180Mflops to 250Mflops, and also a significant decrease in data cache refills from L2. Impressive!

Preventing array temporaries

Fortran runtime warning: An array temporary was created. This happens in BATSRUS when we are trying to pass part of a big array into another function. This potentially slows down the code.

Mostly in `ModFaceValue`, `tvd_reschange`, `fine1_vii, fine2_vii, finef_vii`, `ModPartImplicit`, `get_face_flux`, `impl2expl`, `implicit2explicit`, `ModWritePlotIdl`, `ModPartImplicit` and `ModFaceFlux`. Also some in the BATL library.

MPI barriers

Currently there are two additional MPI barriers in the explicit schemes: `expl1` and `expl2`. They can be turned off in `PARAM.in`. If they are off, the only place where all processors are synchronized is the `isend/irecv` in `BATL_pass_cell`. The `BATL_pass_face` still uses the old

way of communication, but since it is very fast, there is no point of rewriting it in the new way.

Issues about ModFaceFlux

Previously each block select its own flux type, even if it is the same for all. Now a new init function is added for setting the flux types once. It sets flux type outside the block loop, which is nice for performance.

`ModFaceFlux` dominates the time, and it is about 3 times slower than `ModFaceValue`. What Gabor remembered is it is as expensive as `ModFaceValue` before.

Penalty for using OpenMP

A serious problem came up: the current code compiled with openmp runs much slower than the one compiled without openmp in single thread runs! For shocktube test, 50% slower. 2 threads gives roughly the same speed as 1 pure mpi. Oh my god.

I checked the runs with no compiler optimization. More than 2 times slower! Unbelievable!

Question on stackoverflow

Strange things: the bad performance never happens on local linux machines!

I believe there's a deeper reason underneath.

Current observations:

1. Using gfortran on linux gives nearly the same timing ($0.83s$);
2. Using ifort on linux gives 1.5 times differences in timing ($0.73s$ vs $1.03s$);
3. Using ifort on Yuxi's Linux: the timing is not very steady. It can vary within a factor of 3. ($0.54s$ to $1.48s$) Using 2 threads has no speedup in timing, but using 4 threads has some effect.
4. Using nagfor on Mac gives almost 10 times differences in speed;
5. Using gfortran on Mac gives 2 times differences in speed.

The test results on 07/01/2019 using various compilers and platforms are shown in Figure 5.1, 5.2, and 5.3.

`unused_B` is causing some performance issues.

Limiters in `ModFaceValue.f90` may be improved. I tried, and failed. In flux limiters, intel detects anti-dependencies, but it left them unvectorized for $-O2$. Proven dependency? There also seems to be a lot of mispredicting branches.

Compiler suggests re-ordering the case and if statements.

`get_physical_flux`: vectorize serialized functions inside loop

Data alignment

data not aligned: (because nVar is not a multipler of 4?)

```
primitive_VG = State_VGB(:,:,:,:,iBlock)
State_Old(:,:,:,:,iBlock) = State_VGB(:,:,:,:,iBlock)

Flux_V = Flux_V * Area
```

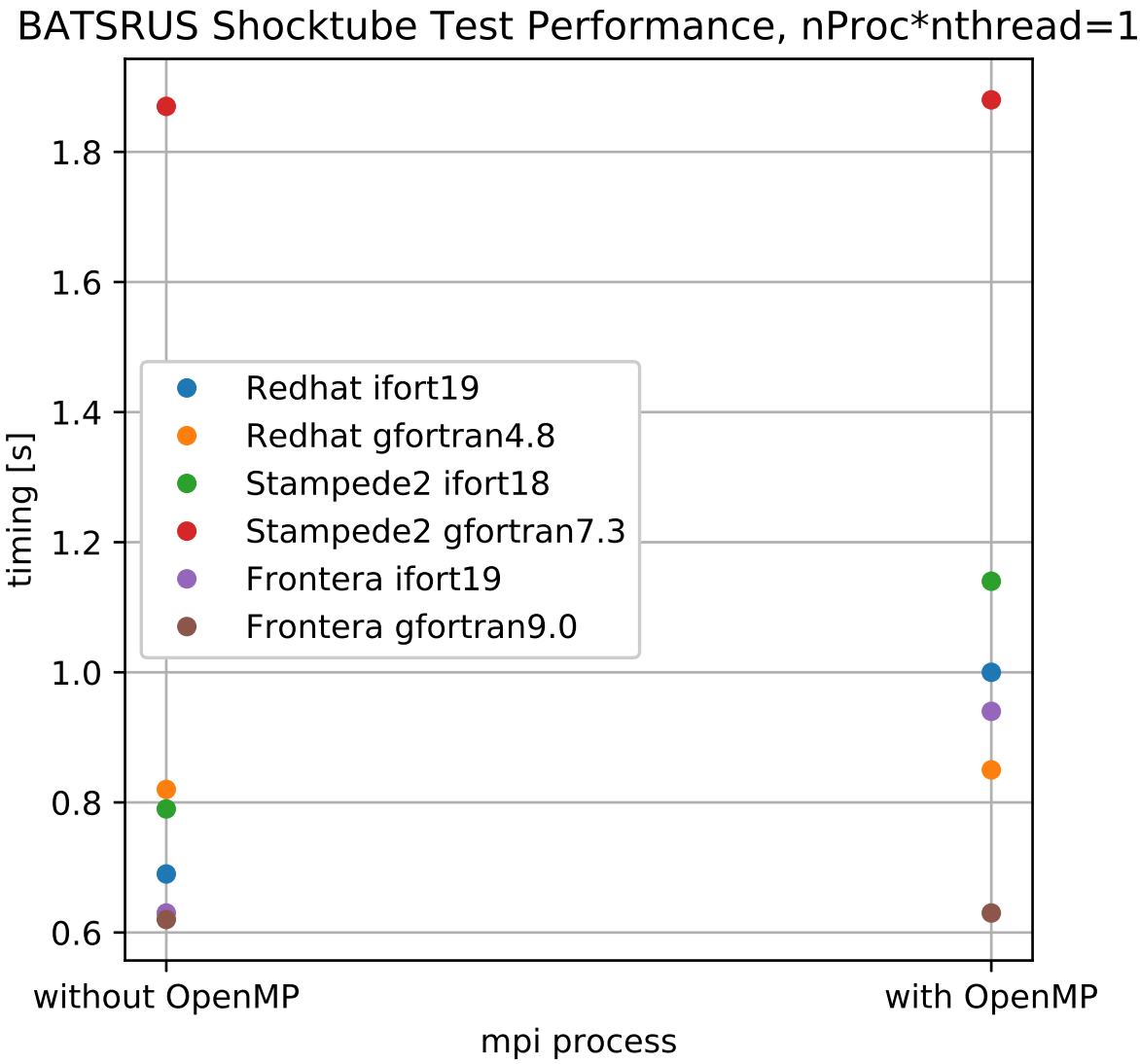


Figure 5.1: Comparisons of BATSRUS compiling with or without OpenMP flag.

Ineffective peeled/remainder loop present

If anti-dependency exists, use a directive where k is smaller than the distance between dependent items in anti-dependency.

`MPI_reduce` gets stuck with many MPI processes?

Performance

Compared the 2019/05/20 version with 2019/01/01, the current face flux calculation runs $\sim 30\%$ slower in the Ganymede test!

Now I am trying the find out what causes the issue. The 2019/02/06 code seems closer to the 01/01 code performance.

My first guess is that Igor's MhdFlux or HdFlux has penalties. However, after some check this is not the case. In the end, surprisingly, the `norm2` introduced in `ModUtility` for pgfortran not implementing the intrinsic function is the reason! This causes 30% performance loss with

gfortran on Mac and 10% loss with ifort on Linux for the flux calculation.

Memory Deallocation

Because BATSRUS is designed for multi-sessions, memories may be freed in the middle of the runs between sessions. There are potentially a lot of places where this is neglected.

BATL API

The BATL library now has a strange logic: there are multiple modules containing their own variables and functions, but they should be collected together so that a outside user can call them with a unique module name `BATL_lib`. There are restrictions on this for some versions of ifort.

Resistivity

Why is `set_resistivity` being called inside `calc_heat_exchange`? It is actually called three times in one timestep for a second order scheme!

Calculate heat exchange tries to close the gap between ion and electron temperatures. Resistivity may change between runs, so it is reset every time. This may cause unnecessary work!

Additionally, a temperory array is allocated repeatedly, which is probably a waste of time.

Another potential issue: Unused_B(iBlock)

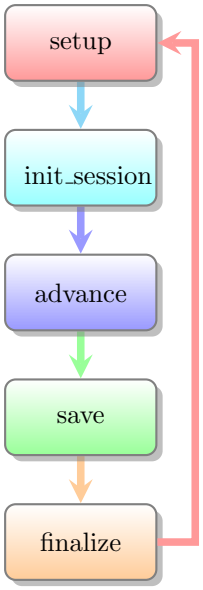
If we use AMR, we may have unused blocks in the array. This means that the load balancing among threads may not be optimal. Depends how threads are launched. Maybe it works just fine. Could be interesting to test.

OpenMP overhead

I want to check if the openMP overhead by compiling with openMP is a fixed number or is dependent on the problem size.

5.2 Code Structure

On the highest level, there's a session loop which looks like



`ModMultiIon`: calculate source terms for multi-ion MHD, usually used with point-implicit schemes.

`ModCalcSource`: module for calculating explicit/implicit source terms. Only one public function `calc_source`

`ModThreadedLC`: seems to be related to the sun. UseFieldLineThreads

`Mod_Block_Data`: moves cache data from block to block for the sake of load balance and adaptive mesh refinement.

5.3 OpenMP Modules

Modules that are currently modified with OpenMP directives are:

- `ModAdvance`
- `ModAdvanceExplicit`
- `ModB0`
- `ModBorisCorrection`
- `ModFaceFlux`
- `ModFaceGradient`
- `ModFaceValue`
- `ModFaceBoundary`
- `ModCellBoundary`
- `ModMultiFluid`
- `ModMultiIon`
- `ModPointImplicit`
- `ModViscosity`
- `ModCurrent`
- `ModFaceGradient`
- `ModWaves`
- `ModHallResist`
- `ModHeatConduction`
- `ModCoronalHeating`
- `ModRadDiffusion`
- `ModRadiativeCooling`
- `ModChromosphere`
- `ModUpdateState`
- `ModThreadedLC`
- `ModSemiImplicit`

- ModResistivity
- ModPhysics
- ModPartImplicit
- ModMessagePass
- ModGeometry
- ModChromosphere
- BATL_region
- ModLinearSolver
- ModExactRS
- ModPhysics
- ModRestartFile

Now exchange_energy, update_check are also multi-threaded. exchange_energy is for transforming between temperature and energy?

ModConserveFlux can be multi-threaded, but it is not currently because it takes too little time.

ModConstrain is rarely used, so it is not multi-threaded.

5.4 Multi-thread Tests

Currently all the overnight tests for BAT-S-RUS are running with 2 threads. Table 5.1 summarizes the accuracy results.

Table 5.1: BATSRUS nightly tests

Test Name	Status	Description
test_func		
test18	?	P_{max} diff
test20	?	
2bodyplot	✓	
amr	?	time accurate, AMR, error down to machine ϵ
anisotropic	✓	
awsomfluids	X	lack of input data files
chromo	✓	semi-implicit
comet	✓	point-implicit
cometCGfluids	✓	point implicit, BlockData
cometCGhd	✓	
cometfluids	X	point-implicit, crashed in `ModUser` neutral atmosphere
cometfluids_restart	X	
corona	✓	
coronasph	✓	AMR
earthsph	✓	part-implicit, difference in time-accurate run
eosgodunov	X	crash, cannot even run serial
fivemoment_alfven	✓	explicit
fivemoment_langmuir	✓	explicit
fivemoment_shock	✓	point-implicit
fluxemergence	X	lack of include file
ganymede	✓	semi-implicit, at least works with serial implicit solver
graydiffusion	?	full-implicit, hd, diff since 2 step
hallmhd	✓	semi-implicit, part-implicit, `BATL_region` allocation
laserpackage	X	
magnetometer	✓	
mars	✓	fixed `user_calc_sources`
mars_restart	✓	fixed `user_calc_sources`
marsfluids	✓	point-implicit
marsfluids_restart	✓	
mercurysph	✓	explicit, Hall resistivity, `BATL_region`
mhdions	✓	point-implicit
mhdnoncons	✓	
multifluid	✓	
multiion	✓	point-implicit
outerhelio	✓	
outerheliopui	✓	fixed `ModUser`
partsteady	✓	randomly crashed, sometimes good?
region2d	✓	
saturn	✓	user sources
shockramp	?	5^{th} order scheme, AMR, sometimes diff
shocktube	✓	
spectrum	X	compile needs data
titan	✓	`ModUser` similar to Mars
titan_restart	✓	
twofluidmhd	✓	`ModUserWaves`
venus	✓	`ModUser` similar to Mars
venus_restart	✓	
viscosity	✓	`ModViscosity`, `ModFaceFlux`

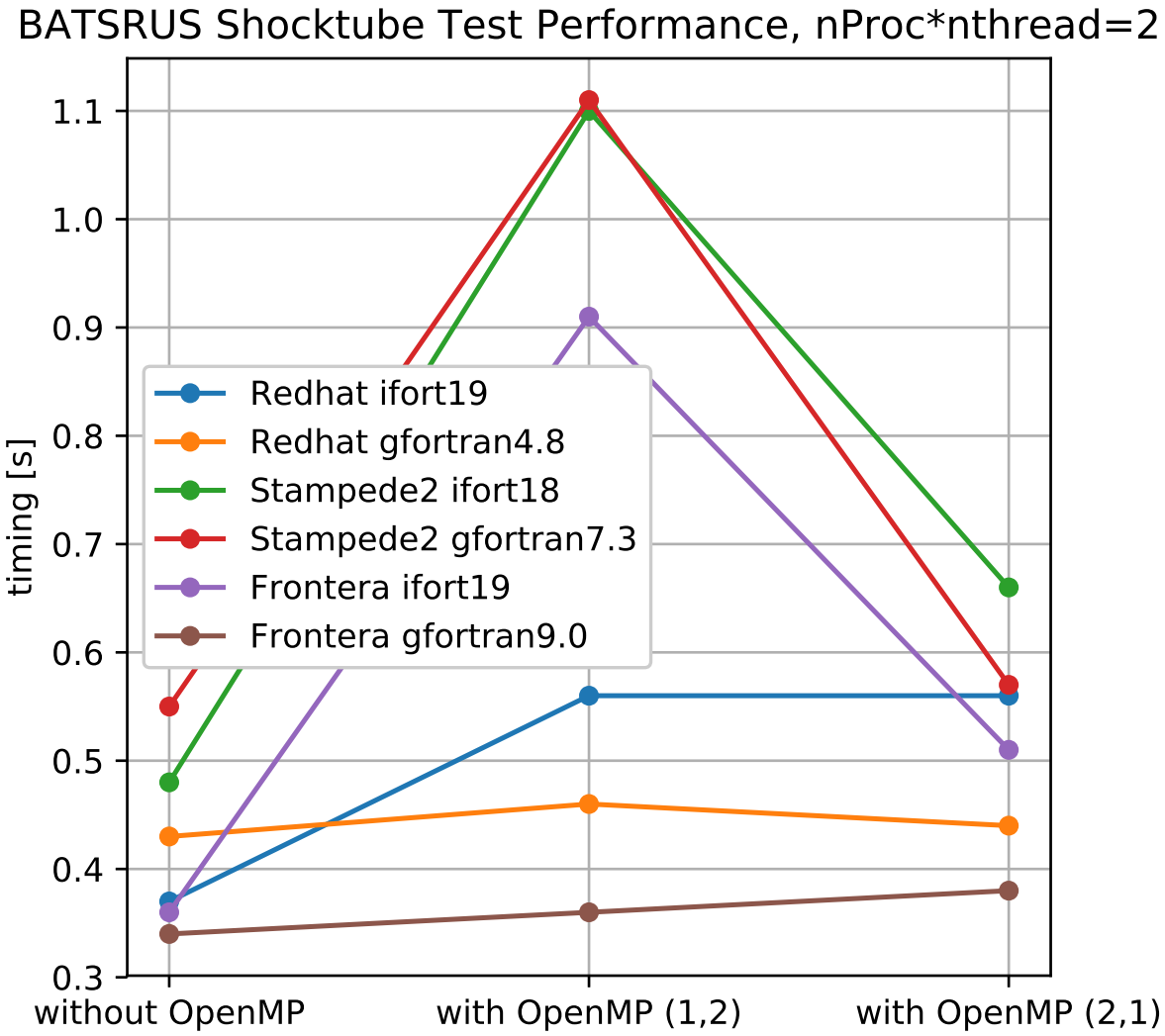


Figure 5.2: Comparisons of BATSRUS compiling with or without OpenMP flag. Three vertical lines represent 2 MPI compiling without OpenMP, 1 MPI and 2 threads, and 2 MPI and 1 thread.

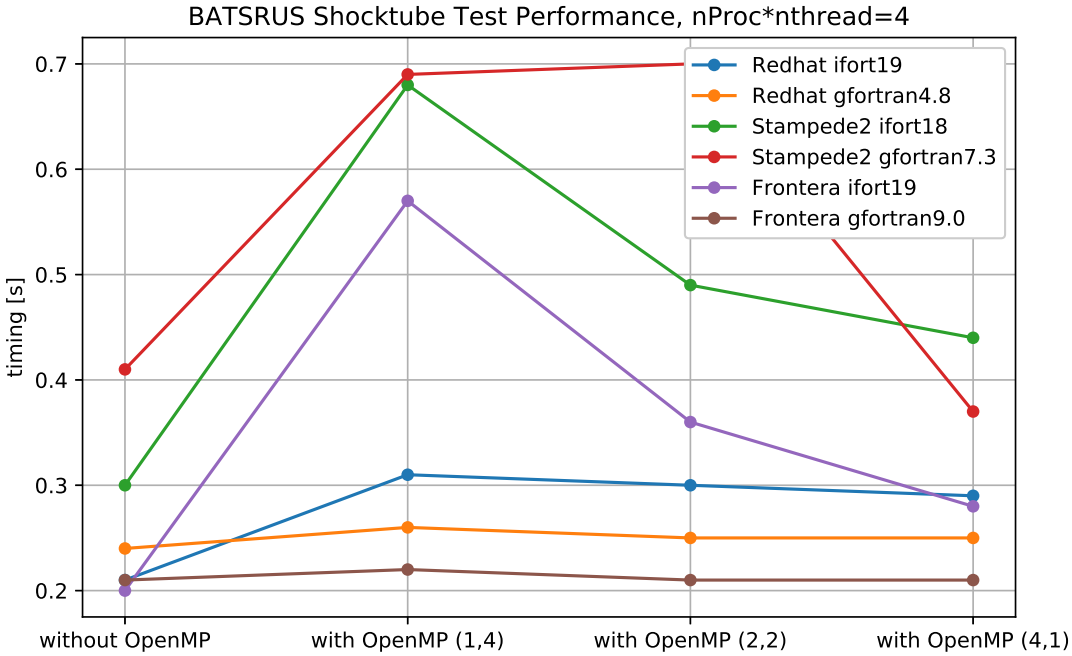


Figure 5.3: Comparisons of BATSRUS compiling with or without OpenMP flag. Four vertical lines represent 4 MPI compiling without OpenMP, 1 MPI and 4 threads, 2 MPI and 2 threads, and 4 MPI and 1 thread.

5.4.1 Auto-checker

We had an idea of writing a script for automatically checking the module variables that are changing inside block loops. It is really interesting. Start from `share/Scripts/AddTest.pl` for the regular expressions.

Always not sure about `var_I` type of variables.

5.4.2 Issues

It is annoying that OpenMP doesn't have a threadprivate region syntax. I added the threadprivate declarations right after the variables.

`user_calc_sources` in ModUserMars calls `user_impl_source`, which is why multi-threading not working.

`var_I` that is set at the initial time shouldn't be declared as threadprivate!

```
!hyzhou: iBoundary,iFace,jFace,kFace is defined here only because they are
!used inside user_set_face_boundary. It would be much better if we could
!remove this prone-to-error usage!!!
```

It turns out that some var with _I shouldn't be declared as threadprivate. Those that are set at the inital setup and used later should not be threadprivate.

openmp module var question

An old openmp fortran question list

I can see upper/lowercase difference in log file for Titan test.

Inside GMRES, there is a dot product for normalization. There doesn't seem to be a gain in speed if I use OpenMP workshare.

I fixed the problem that fully implicit solver cannot work in `impl_newton_init` by adding threadprivate clause to `ModHallResist` and `ModWaves`.

For Mecury test, it turns out that I missed the module variable `Area` in `BATL_region` and `UseHallGradPe`.

Finally, this earthsph test passed! The final problem is in the rotation of the magnetic axis: `omega_D` should be declared as threadprivate.

Point implicit scheme debugging

`update_point_implicit` is called per thread. `update_state_normal`, if `UsePointImplicit`, call `update_point_implicit` `iVarPointImpl_I` shouldn't be threadprivate?

I now understand why the point-implicit solver shows some strange behaviours. That is all due to initializations?

I realized one big issue when passing function as an argument: OpenMP seems to totally ignore the function after the first evaluation? This `user_calc_sources` is passed as a pointer to `calc_point_impl_source`, and it is lost after one call!

Problem solved: that was due to the improper scope of some initialized logical variables in the function. These logicals are set to true at the beginning of a loop and set back to false at the end of the loop. All those strange multi-thread behaviors show up if I don't declare them as

threadprivate: they are shared, they may change value in the middle of the loop due to the asynchronization execution of threads!

Now after a while, the issue showed up again, in cometfluids and europa2fluidsPe tests.

Thread scheduling

Load imbalance

Nowait clause

5.5 Point Implicit Scheme

- `read_point_impl_param`
- `init_point_implicit_num`
- `init_mod_point_impl`
- `clean_mod_point_impl`
- `update_point_impl`

In subroutine `update_point_implicit(iBlock, calc_point_impl_source, init_point_implicit)`, the latter two arguments are functions.

```
! Switch to implicit user sources
IspointImplSource = .true.

! Initialization

! Store explicit update
StateExpl_VC = State_VGB(:,1:nI,1:nJ,1:nK,iBlock)

! Put back old values into the implicit variables so the update is relative to
    time level n
State_VGB(iVar,i,j,k,iBlock) = StateOld_VGB(iVar,i,j,k,iBlock)

! Calculate unperturbed source for RHS
call calc_point_impl_source(iBlock)

! Calculate (part of) Jacobian numerically if necessary
if(.not.IsPointImplMatrixSet)then
   ...
end

! Do the implicit update
do k=1,nK; do j=1,nJ; do i=1,nI
   ! Do not update body cells
   if(.not.true_cell(i,j,k,iBlock)) CYCLE

   DtCell = Cfl*time_BLK(i,j,k,iBlock) * iStage / real(nStage)

   ! RHS is Uexpl - Uold + Sold
   do iIVar=1,nVarPointImpl
      iVar=iVarPointImpl_I(iIVar)
      Rhs_I(iIVar) = StateExpl_VC - StateOld_VGB + DtCell * Source_VC
   end

   ! The matrix to be solved for is A = (I - beta*Dt*dS/dU)
   do iIVar = 1, nVarPointImpl
      iVar = iVarPointImpl_I(iIVar)
```

```
36        do iJVar = 1, nVarPointImpl
37            jVar = iVarPointImpl_I(iJVar)
38            Matrix_II(iIVar,iJVar) = - BetaStage*DtCell*DsDu_VVC(iVar,jVar,i,j,k)
39         end do
40         ! Add unit matrix
41         Matrix_II(iIVar,iIVar) += 1.0
42      end
43
44      ! Solve A.dU = RHS
45      call linear_equation_solver(nVarPointImpl, Matrix_II, Rhs_I)
46
47      ! Update: U^n+1 = U^n + dU
48      do iIVar=1,nVarPointImpl
49         iVar = iVarPointImpl_I(iIVar)
50         State_VGB(iVar,i,j,k,iBlock) = StateOld_VGB(iVar,i,j,k,iBlock) + Rhs_I(
       iIVar)
51      end
52
53      ! Set minimum density (no negative density allowed)
54   end; end; end
```

Listing 5.1: Point implicit scheme

Point implicit solver is operated per block, so only a linear solver per block is required.

5.6 Part Implicit Scheme

Part implicit means that only a portion of blocks are updated with implicit schemes. Obviously full implicit scheme is a subset of part implicit scheme. The main module consists of four functions:

- `read_part_impl_param`
- `init_mod_part_impl`
- `clean_mod_part_impl`
- `advance_part_impl`

By default the Newton iteration is only performed one step. `solve_linear_multiblock` chooses an implicit solver for all the implicit-treated blocks on the current processor. The key part of algorithm:

```
1  ! Initialize some variables in ModImplicit
2  call implicit_init
3  ! Get initial iterate from current state
4  call explicit2implicit
5
6  call MPI_allreduce(nImpl, nImplTotal)
7
8  call MPI_allreduce(NormLocal_V, Norm_V)
9
10 ! Advance explicitly treated blocks if any
11 if(UsePartImplicit .and. nBlockExplALL > 0) then
12    if(UseBDF2) then
13       ! Save the current state into the previous state
14       do iBlock=1,nBlock
15          ...
16       end
17    end
18
```

```
   if(.not.UsePartImplicit2) then
      ! Select Unused_B = not explicit blocks
   end

   ! Advance explicit blocks
   call advance_explicit(.true., -1)

   if(.not.UsePartImplicit2) then
      ! update ghost cells for the implicit blocks to time level n+1
   end

   call exchange_messages

   ! Make sure the implicit scheme is only applied on implicit blocks
end

!\
! Advance implicitly treated blocks
!/

! Let other parts of the code know that we are inside the implicit update
! Switch off point implicit scheme (no need)
! Switch off merging the cells around the poles
! Use implicit time step

if(.not.UseBDF2)then
   ! Save the current state into ImplOld_VCB so that StateOld_VGB can be restored
end

! Initialize right hand side and x_I. Uses ImplOld_VCB for BDF2 scheme.
call impl_newton_init

! Save previous timestep for 3 level scheme
if(UseBDF2)then
   !hyzhou: this is almost the same as not using BDF2!!!
end

! Newton-Raphson iteration and iterative linear solver
NormX = BigDouble
nIterNewton = 0
do
   nIterNewton = nIterNewton + 1
   if(nIterNewton > MxIterNewton)then
      error
      EXIT
   end if

   ! Calculate Jacobian matrix if required
   if(ImplParam%DoPrecond)then

   end

   ! Update rhs and initial x_I if required
   if(nIterNewton > 1) call impl_newton_loop

   ! Solve implicit system
   ! For Newton solver the outer loop has to converge,
   ! the inner loop only needs to reduce the error somewhat.

   if(UseNewton) ImplParam%ErrorMax = 0.1
   call solve_linear_multiblock

   if(ImplParam%iError /= 0) error; end
```

```
82
83    ! Update w
84    call impl_newton_update
85
86    if(IsConverged) EXIT
87 end do
88
89 if(UseConservativeImplicit) call impl_newton_conserve
90
91 ! Put back implicit result into the explicit code
92 call implicit2explicit(Impl_VGB)
93
94 ! Make explicit part available again for partially explicit scheme
95 if(UsePartImplicit)then
96    ...
97 end
98
99 ! Exchange messages, so ghost cells of all blocks are updated
100 call exchange_messages
```

Listing 5.2: Part implicit scheme

5.7 Semi-implicit Scheme

Semi-implicit scheme means that only part of the variables are solved with implicit schemes. The main module also consists of four functions:

- `read_semi_impl_param`
- `init_mod_semi_impl`
- `clean_mod_semi_impl`
- `advance_semi_impl`

The key algorithm in `advance_semi_impl`:

```
1  ! Check semi-implicit blocks, store B field in SemiAll_VCB
2  call get_impl_resistivity_state(SemiAll_VCB)
3
4  do iVarSemi=1,nBarSemiAll,nVarSemi
5     ! Set right hand side
6     call get_semi_impl_rhs(SemiAll_VCB, Rhs_I)
7     ! Calculate Jacobian matrix if required
8     if(SemiParam%DoPrecond)then
9        call get_semi_impl_jacobian
10    end
11
12    ! solve implicit system
13    call solve_linear_multiblock
14
15    if(SemiParam%iError /= 0) call error_report
16
17    ! Update solution
18    n = 0
19    do iBlockSemi=1,nBlockSemi; do k=1,nK; do j=1,nJ; do i=1,nI
20       do iVar=iVarSemiMin, iVarSemiMax
21          n = n + 1
22          if(true_cell)then
23             ! update
24          else
```

```
            ! no update
         end
      end
   end; end; end; end
end

! Put back semi-implicit result into explicit code
do iBlockSemi=1,nBlockSemi
   call update_impl_resistivity(iBlock, NewSemiAll_VCB(:,:,:,iBlockSemi))
end
```

Listing 5.3: Semi-implicit scheme

`get_semi_impl_rhs_block` is a wrapper which calls `get_resistivity_rhs` in `ModResistivity`.

Observation:

1. Key variables:

   `Rhs_I(nVarSemi*nIJK*MaxBlock)`

   `x_I(nVarSemi*nIJK*MaxBlock)`

2. Key functions:

   `solve_linear_multiblock`

Table 5.2: Implicit inputs

s

Variables	Meaning	Notes
`SemiParam`	solver parameters	derived type
`nVarSemi`	# of implicit vars	`ModImplicit`
`nDim,nI,nJ,nK,nBlockSemi`	grid info	`BATL_lib`
`iComm`	communicator	`ModProcMH`
`semi_impl_matvec`	Calculate $Rhs = Ax$	function
`Rhs_I`		nVarSemi*nIJK*MaxBlock
`x_I`		nVarSemi*nIJK*MaxBlock
`DoTestKrylov`		
`JacSemi_VVCIB`	Jacobian/preconditioner	nVarSemi,nVarSemi,nI,nJ,nK,nStencil,Max
`JacobiPrec_I`	pointwise Jacobi preconditioner	nVarSemi*nIJK*MaxBlock
`cg_precond`	CG preconditioner	function
`hypre_preconditioner`		ModImplHypre

`Rhs_I` is calculated by `semi_impl_matvec`, which is actually calculated in `solve_linear_multiblock`. This is called inside, for example, the Krylov loop. For the MPI version, each processor calls this solver once per time step.

`semi_impl_matvec` is the key part:

```
! Fill in SemiState so it can be message passed
SemiState_VGB = SemiAll_VCB

! Message pass to fill in ghost cells
call message_pass_cell(nVarSemi, SemiState_VGB)

do iBlockSemi=1,nBlockSemi
   ! Apply boundary conditions (1 layer of outer ghost cells)
   if(far_field_BCs_BLK(iBlock)) call set_cell_boundary

```

```
   call get_semi_impl_rhs_block
end

! Multiply with cell volume ( makes matrix symmetric)
do iBlockSemi=1,nBlockSemi
   RhsSemi_VCB *= CellVolume_GB
end

if(UseStableImplicit)then
   ...
end
```

Listing 5.4: Semi-implicit matvec

`get_semi_impl_rhs_block`: wrapper for radiation/heat conduction/resistivity calls.

`get_resistivity_rhs`: get `Rhs_VC` from `StateImpl_VG` for each block

```
do iBlockSemi=1,nBlockSemi
   SemiState_VGB = SemiAll_VCB
end do

call message_pass_cell(SemiState_VGB)

do iBlockSemi=1,nBlockSemi
   call get_semi_impl_rhs_block(iBlock, SemiState_VGB, RhsSemi_VCB)
end do

do iBlockSemi=1,nBlockSemi
   Rhs_I =  f(ResSemi_VCB)
end do
```

Listing 5.5: ???

```
advance_semi_impl
├─ get_semi_impl_rhs
│  ├─ message_pass_cell
│  ├─ set_cell_boundary
│  ├─ get_semi_impl_rhs_block
│  ├─ message_pass_face
│  └─ apply_flux_correction_block
└─ solve_linear_multiblock
   ├─ semi_impl_matvec
   │  ├─ message_pass_cell
   │  ├─ set_cell_boundary
   │  ├─ get_semi_impl_rhs_block
   │  │  ├─ get_rad_diffusion_rhs
   │  │  ├─ get_heat_conduction_rhs
   │  │  └─ get_resistivity_rhs
   │  ├─ message_pass_face
   │  ├─ apply_flux_correction_block
   │  └─ semi_precond
   └─ update_impl_*
```

5.7.1 Solution

There are four parts of the implicit solver that can be multi-threaded:

1. wrap the state variables into 1D long arrays and unwrap them after the iterative solver;

2. building the preconditioner;
3. `matvec`
4. `matmul(x_I,y_I)` inside the implicit Krylov solver.

By testing, I found that calling function `matvec` dominates the time on one processor. The timing report shows that matrix multiplication takes about 30% of the total execution time.

5.8 Testing

5.8.1 Strong scaling

test case compiler, optimization

Grid (cell + block)

mpi + openmp

The results for the implicit scheme are not very steady. 2 thread cases gives completely different timings!

5.8.2 Weak scaling

5.9 BATL

My first attempt to modify `BATL_pass_cell` failed. Although it can run successfully, but it takes more time than the original version, which is very sad to me. I compared the performance both with no optimization and with optimization, and the old version always beats my first version. For 1 MPI, sure it does speed up with multithreads; but with MPI calls for muliple MPIs, it slows down.

The problem for the first attempt turns out to be memory allocation of `Slope_VG` every time in the block loop. The performance degraded because of this memory allocation.

Something is wrong with `advect33_sph`.

Here comes the third attempt. Accidentally, I found that doing local copy in another block loop after preparing the remote copy buffer gives significant improvement in terms of speed, especially for large thread numbers. This is probably due to the compiler optimization with a certain logical instead of running decision. Also for our cases, whether or not the location of local passes is after waitall or before waitall almost doesn't matter! Now it gives almost perfect timing all the way up to 8 threads; 16 is pretty good; 32 is ok. The best performance is achieved with 2 threads.

Why the pure MPI code fails at 32768 cores: the tree structure allocations in `BATL_tree`:

1. `iTree_IA`: $32768[\text{cores}] * 256[\text{blocks/core}] * 18[\text{integers/block}] * 4[\text{byte}] * 2 \approx 1\text{GB}$
2. `iNodeMorton_I`: $32768 * 256 * 2 * 4$
3. `iMortonNode_A`: $32768 * 256 * 2 * 4$
4. `iStatusNew_A`: $32768 * 256 * 2 * 4$
5. `iStatusAll_A`: $32768 * 256 * 2 * 4$

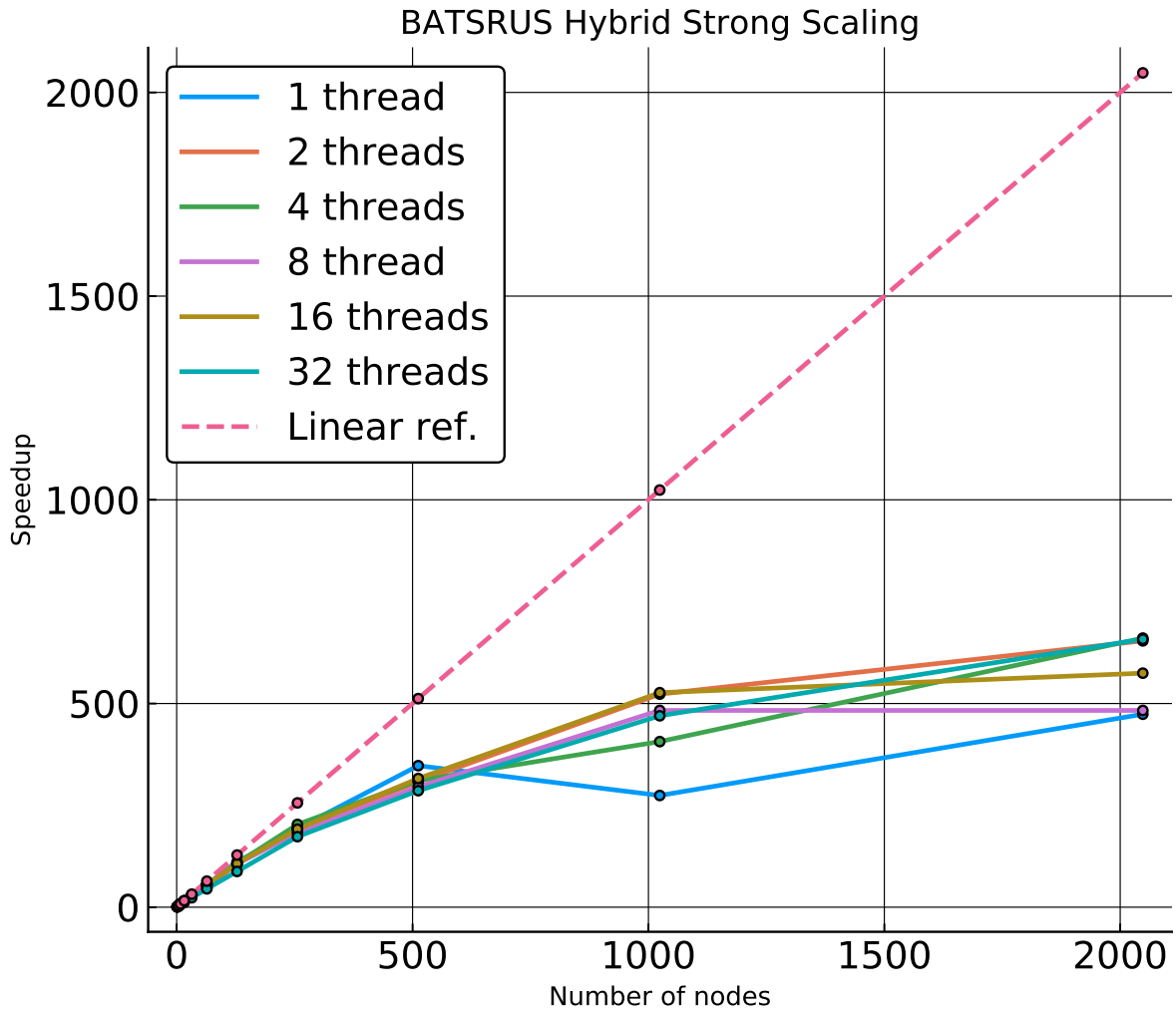


Figure 5.4: BAT-S-RUS hybrid strong scaling tests of 3D shocktube problem using second order explicit Linde scheme on Bluewaters. Each computing node has two AMD 6276 Interlagos processors, and a single XE node has 32 "integer" cores. The number of blocks per core ranges from 2048 to 1 for single thread runs.

6. `iProcNew_A`: $32768 * 256 * 2 * 4$
7. `iNodeNew_A`: $32768 * 256 * 2 * 4$

The 2 for each array accounts for the possible adaptive mesh refinement. All together these take about 2GB of memory, which exceeds the memory limit per core on Bluewaters.

5.10 Tecplot

I want to the smooth the process of writng tecplot outputs. Right now we are saving finite element type data in ACSII format from each processor, using post-processing executables to merge the data and connectivities into one large ACSII file, and then using preplot to transform it into binaries. This is a practical bottleneck for doing real simulation work.

The open source TecIO library provides functions to read and write tecplot formats easily. It

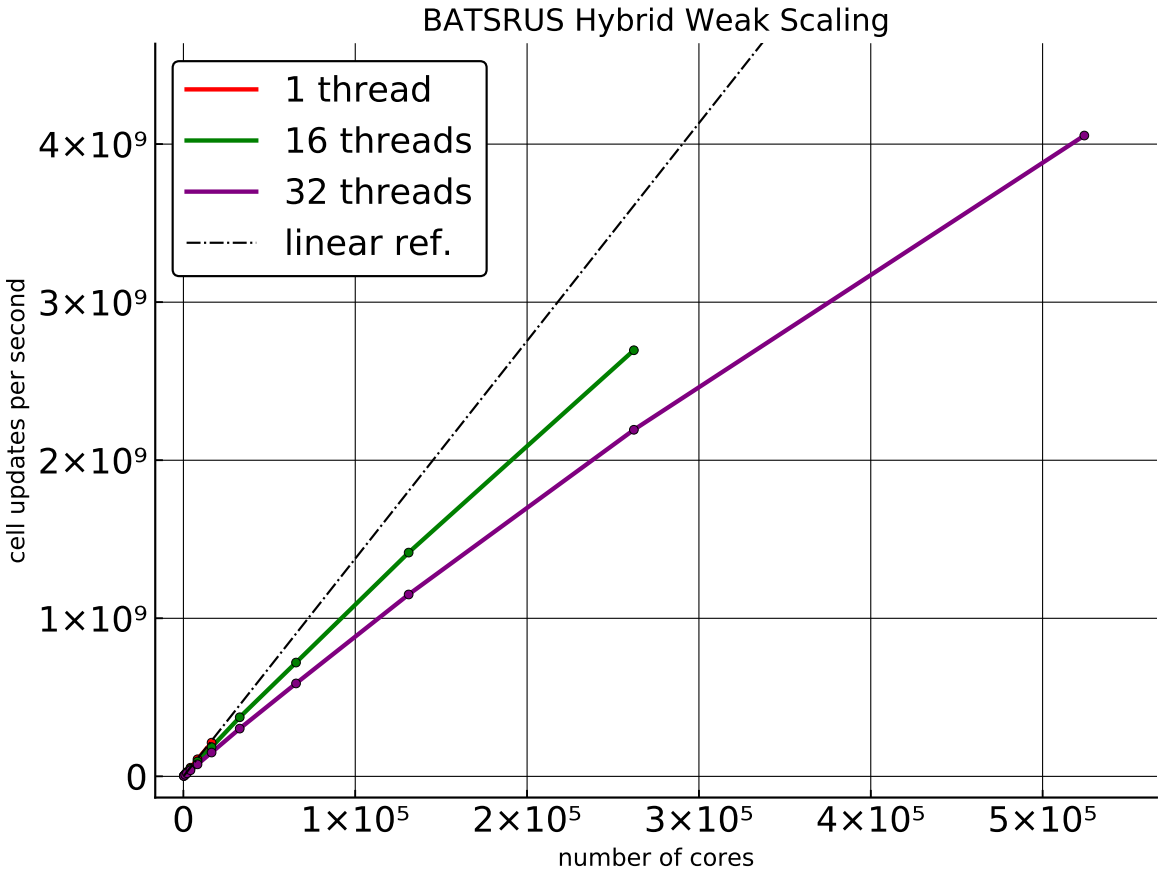


Figure 5.5: BAT-S-RUS hybrid weak scaling tests of 3D shocktube problem using second order explicit Roe scheme on Bluewaters. Each computing node has two AMD 6276 Interlagos processors, and a single XE node has 32 "integer" cores. The number of blocks per core ranges from 2048 to 1 for single thread runs.

relies on the boost library in C++ and has an interface to Fortran. In the first stage, I successfully linked the library to BATSRUS and made some dummy outputs from each processor. Then I need to design the workflow for a better IO. Remaining questions are:

1. Can I write to the same output file in parallel with TecIOMPI?
2. If I choose to write output for each processor separately, how can I combine all the pieces into one large binary file?

I am suffering from this right now. A nice solution to me now is to write one grid file (assuming no AMR) and one solution file per timestep. But I don't understand how szplt works! What are partitioned zones?

n TecIO, the portion of a zone on an MPI rank is called a partition. The use of partitions to write a single file from multiple processors is described in section 3-3 of the Tecplot 360 Data Format Guide. It only requires a couple more API calls to write the data in parallel. The most complicated part is dealing with overlapping, or ghost nodes, in the data. A couple of the example applications in the TecIO package write partitioned files. Check out brickpartitioned.cpp (or .F90)

5.11 Future Development

One thing to try for sure is OOP.

Using `threadprivate` works, but this is not good for reading codes, maintainance and performance. Readers cannot quickly catch the logic flow; maintainers are confused by the variables flying around functions; performance decreases because `threadprivate` variables are allocated on heaps instead of stacks.

Quoted from the paper Techniques Supporting threadprivate in OpenMP:

```
threadprivate must be used with care. Do not consider that a threadprivate
variable can be as efficient as a private variable. Threadprivate implies that
variables must be placed in common blocks to be kept save across parallel
regions. This may prevent some optimizations, like keeping values in registers
and the code will be slower. So use threadprivate only when the variables
involved need to keep their values across functions and parallel regions.
```

Implementation of `threadprivate` based on thread-local storage (TLS) is implemented in linux gfortran compiler and greatly reduced the overhead.

In the long run, a far better approach may be to make the flows of information in your program explicit to a reader of the code (and more flexibly configurable by a code writer), by passing information as arguments (perhaps bundled together in derived types) rather than hiding those flows through the use of global (module) variables.

5.11.1 Flux Calculation

I have a plan for rewriting the whole flux module. Goals:

1. Eliminate the usage of module private variables.
2. Eliminate passing scalars between modules except constants.
3. Try to use submodules for different schemes?
4. Do I need to keep cell center flux calculations?
5. Figure out a way to separate the implicit solver with the face flux calculation, e.g. DoTestCell.

5.11.2 Point Implicit Scheme

Try to eliminate the logicals for initiating the scheme!

5.11.3 Test Module

I don't like the current implementation of test functionalities, both the results checking and the subroutine tests.

Chapter 6

Parallel Affinity Control

The matter of thread affinity becomes important on multi-socket nodes.

Thread placement can be controlled with two environment variables:

1. `OMP_PROC_BIND`: describes how threads are bound to OpenMP places
2. `OMP_PLACES`: describes these places in terms of the available hardware.

When you're experimenting with these variables it is a good idea to set `OMP_DISPLAY_ENV` to true, so that OpenMP will print out at runtime how it has interpreted your specification.

6.1 Thread Binding

The variable `OMP_PLACES` defines a series of places to which the threads are assigned.

Example 1: if you have two sockets and you define `OMP_PLACES`=sockets, then it goes like round-robin.

Example 2: if the two sockets have a total of sixteen cores and you define `OMP_PLACES`=cores ,`OMP_PROC_BIND`=close, then it goes successively through the available places.

Example 3: if the two sockets have a total of sixteen cores and you define `OMP_PLACES`=cores ,`OMP_PROC_BIND`=spread, then it goes round-robin alternatively on the two sockets.

So you see that `OMP_PLACES`=cores and `OMP_PROC_BIND=spread` very similar to `OMP_PLACES`=sockets . The difference is that the latter choice does not bind a thread to a specific core, so the operating system can move threads about, and it can put more than one thread on the same core, even if there is another core still unused.

The value `OMP_PROC_BIND`=master puts the threads in the same place as the master of the team.

6.2 Effects of Thread Binding

Let's consider two example program. First we consider the program for computing π, which is purely compute-bound.

We see pretty much perfect speedup for the `OMP_PLACES`=cores strategy; with `OMP_PLACES`=sockets we probably get occasional collisions where two threads wind up on the same core.

Next we take a program for computing the time evolution of the heat equation:

$$t = 0, 1, 2, \ : \forall_i : x_i^{t+1} = 2x_i^t - x_{i-1}^t - x_{i+1}^t$$

This is a bandwidth-bound operation because the amount of computation per data item is low.

6.3 Place Definition

There are three predefined values for the `OMP_PLACES` variable: *sockets, cores, threads.* You have already seen the first two; the threads value becomes relevant on processors that have hardware threads. In that case, `OMP_PLACES`=cores does not tie a thread to a specific hardware thread, leading again to possible collisions as in the above example. Setting `OMP_PLACES`=threads ties each OpenMP thread to a specific hardware thread.

There is also a very general syntax for defining places that uses a

```
location:number:stride
```

syntax.

Examples:

1. `OMP_PLACES`="{0:8:1},{8:8:1}" on a two-socket design

   Equivalent to *sockets* on a two-socket design with eight cores per socket: it defines two places, each having eight consecutive cores. The threads are then places alternating between the two places, but not further specified inside the place.

2. `OMP_PLACES`="{0},{1},{2},...,{15}"

   Equivalent to *cores.*

3. `OMP_PLACES`="{0:4:8}:4:1" on a four-socket design

   states that the place 0,8,16,24 needs to be repeated four times, with a stride of one. In other words, thread 0 winds up on core 0 of some socket, the thread 1 winds up on core 1 of some socket, et cetera.

6.4 Binding Possibilities

Values for `OMP_PROC_BIND` are: *false, true, master, close, spread.*

1. master: collocate threads with the master thread.
2. close: place threads close to the master in the places list.
3. spread: spread out threads as much as possible.

A safe default setting is

```
export OMP_PROC_BIND=true
```

which prevents the operating system from migrating a thread. This prevents many scaling problems.

As an example, consider a code where two threads write to a shared location.

```
// sharing.c
#pragma omp parallel
  { // not a parallel for: just a bunch of reps
    for (int j = 0; j < reps; j++) {
#pragma omp for schedule(static,1)
      for (int i = 0; i < N; i++){
#pragma omp atomic
  a++;
      }

    }
  }
```

Listing 6.1: sharing

There is now a big difference in runtime depending on how close the threads are. We test this on a processor with both cores and hyperthreads. First we bind the OpenMP threads to the cores:

```
OMP_NUM_THREADS=2 OMP_PLACES=cores OMP_PROC_BIND=close ./sharing
run time = 4752.231836usec
sum = 80000000.0
```

Next we force the OpenMP threads to bind to hyperthreads inside one core:

```
OMP_PLACES=threads OMP_PROC_BIND=close ./sharing
run time = 941.970110usec
sum = 80000000.0
```

Of course in this example the inner loop is pretty much meaningless and parallelism does not speed up anything:

```
OMP_NUM_THREADS=1 OMP_PLACES=cores OMP_PROC_BIND=close ./sharing
run time = 806.669950usec
sum = 80000000.0
```

However, we see that the two-thread result is almost as fast, meaning that there is very little parallelization overhead.

6.5 Affinity control outside OpenMP

There are various utilities to control process and thread placement.

Process placement can be controlled on the Operating system level by `numactl`. Each compiler has its own way to setting the thread affinities. Check them out for a specific machine.

Chapter 7

MHD Code Design

7.1 Principles

- SOLID: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion
- DRY: Don't Repeat Yourself
- KISS: Keep It Simple, Stupid!
- POLA: Principle of Least Astonishment
- YAGNI: You Aren't Gonna Need It
- POLP: Principle of Least Privilege

7.2

Do not duplicate codes. For example, in the boundary condition setup, you should not write individual functions for different dimensions. This will lead to hard-to-follow codes as well as hard-to-detect bugs.

7.3 References

Enzo has cuda involved! It is in C. It probably uses 1D index for multi-dimensional arrays.

Enzo

Enzo source code

Athena has the most complete description and code design in C++!

Athena

Gamera

See the Matlab source code.

BATSRUS has now become a monster. I have an idea of rewrite the kernel structure to make it more like LEGO.

7.4 Data Structure

Almost all the C++ code has their own template for multi-dimensional arrays. From what I can see, each variable is stored continuously in the grid, instead of being stored altogether. I feel like this is faster than what we do in BATSRUS.

For example, in Enzo each variable is stored separately, but there is a Field Type ID defined for finding specific variable. In Athena (C++), a 4D array of size [N4 × N3 × N2 × N1] is accessed as

```
A(n,k,j,i) = A[i + N1*(j + N2*(k + N3*n))]
```

where the trailing index inside the parentheses is indexed fastest.

According to my test in my Matlab MHD code, this gives about 20% speed up for the face flux calculation, and up to 10% speed up for the source term calculation.

If I want to test with BATSRUS, I need to write a script to substitute all the patterns.

For a block-based code, there is always this question of where to put the global variables and "local" variables per block. I don't like current solution in BATSRUS that messes everything together!

We can start from a clean version of BATSRUS, and gradually move all those threadprivate block-local variables inside functions. A nice solution would be, for example, create a class for storing all the block-independent intermediate variables. In this way we can avoid passing too many arguments among functions.

We need OOP features to write clearer codes!

7.5 Numerical Schemes

There are many numerical schemes to choose from. I really like Athena's idea of selecting a specific solver during compilation time instead of runtime: we can avoid many unnecessary complexities of branching inside loops and surely achieve better performance. If you think about it, who don't have time to compile the code if they need to switch to another scheme?

Likewise, the extra features like multi-fluid, multi-species and six-moments should all be separated.

Based on my test results, Gabor's `ModFaceFlux` is very well written so that including 4 or 5 scheme together does not result in tramendous slow down. However, even if this reduces the total number of lines, it is really not good for maintainance. Any new comer to the code would find it hard to follow as so many branches are seen within several layers of function calls. It would be much cleaner to separate each scheme, just like the idea of object oriented programming.

For finite difference schemes, working on the grid is essentially equivalent to doing convolutions.

For the implicit solvers, find for the most suitable packages, instead of writing your own. For example, in Julia there is `IterativeSolvers.jl` that is ready for usage. It both supports matrix and matrix-free form.

7.6 Grid

I want a block mesh generator with AMR. AMRex seems now to be the optimal choice besides BATL. Otherwise I should write my own mesh library in probably Julia.

AMRex currently has successful implementation in both PIC and fluid code. It is possible to combine the two approaches within the same framework. I should evaluate how complicated AMRex is before I actually start something.

7.7 Timings

	nCell	nWork	nBlock	nIter	time
CPU	8	10	1000	100	12.5
GPU	8	10	1000	100	9.4
CPU	8	100	100	1000	197.8
GPU	8	100	100	1000	12.0
CPU_block	8	100	100	1000	58.6
GPU_block	8	100	100	1000	0.4
CPU_block	8	1000	200	1000	1259.8
GPU_block	8	1000	200	1000	1.6
CPU	8	100	10	1000	19.8
GPU	8	100	10	1000	1.3
CPU	8	10	10000	10	12.7
GPU	8	10	10000	10	9.5
CPU	8	10	100	1000	12.4
GPU	8	10	100	1000	9.5
CPU	8	50	100	1000	88.6
GPU	8	50	100	1000	10.4
CPU	16	10000	1	1000	216.8
GPU	16	10000	1	1000	2.6
CPU	16	10000	1	100	21.7
GPU	16	10000	1	100	0.4
CPU	16	100	10	1000	149.4
GPU	16	100	10	1000	1.4
CPU	16	100	10	100	14.9
GPU	16	100	10	100	0.3
CPU	16	10	10	1000	9.1
GPU	16	10	10	1000	1.2
CPU	16	10	100	1000	91.4
GPU	16	10	100	1000	10.5
CPU	32	10	10	1000	70.0
GPU	32	10	10	1000	2.0
CPU	32	10	100	1000	
GPU	32	10	100	1000	22.0
CPU	32	100	10	1000	1160
GPU	32	100	10	1000	2.3
CPU	32	100	1	1000	116.0
GPU	32	100	1	1000	0.37
CPU	64	100	10	100	913.3
GPU	64	100	10	100	1.2

Table 7.1: Timings for the skeleton BATSRUS for porting to GPU with OpenACC