Notes on the thread-safe parallel version of the FM package

`FM_parallel.f95` is the thread-safe version of FM. It can be used with a program that uses coarrays, which is the Fortran-standard way of parallel programming. It should also be ok to use with other common ways of parallel programming (openMP, MPI, etc.).

For the parallel version, file `FM_parallel.f95` combines the thread-safe versions of four files from the standard FM: `fmsave.f95`, `fm.f95`, `fmzm90.f95`, and `FM_Sample_Routines.f95`.

To avoid confusion with the standard FM modules, the corresponding module names in the parallel version are `fmvals_parallel` and `fmzm_parallel`.

Because of restrictions imposed by the need to be thread-safe, this version of FM has several limitations compared to the regular version of FM.

1. Global variables defined in modules are not thread-safe if their values can be changed while the program runs. This means the user sets the FM precision level by defining variable `fm_significant_digits` in module `fmvals_parallel`, then compiles `FM_parallel.f95` and links it to their program. Calling `fm_set` to set the precision is not available.
2. type(im) integer multi-precision numbers have varying numbers of digits that are indirectly limited by `fm_significant_digits`. If type(im) values become too large, in intermediate or final results, the multiple threads may run out of stack space and cause the program to crash.
3. If a user program wants different values for other FM options like rounding mode, screen width for FM output, etc., they must be initialized in that module and not changed while the program runs.
4. Similarly, FM precision level cannot be changed by the user's program while it runs.
5. The `fm_random_number` random number generator cannot be used, since it depends on a global saved state to get the next number.
6. The `fm_deallocate` function is not needed and has been removed, because multi-precision variables are no longer stored in a global module database.
   The `fm_(enter` or `exit)_user_(function` or `routine`) calls are also not needed and those routines have been removed.
7. Saved values like pi, e, euler gamma, etc., are global variables in the standard version of FM. They have been removed from this thread-safe version, so they are re-computed each time they are needed. That makes some functions like trig and log/exponential functions slightly slower.
8. The global allocatable database from version 1.3 has been replaced with local allocatable arrays for the multiple precision numbers. A few routines like `fm_fprime` and `zm_fprime` that relied on raising precision a lot to overcome unstable formulas may now return unknown. Up to 4th or 5th derivatives should usually be ok, but higher derivatives may fail.

9. There are several arrays whose size is proportional to `fm_significant_digits` that are needed in each thread. A program can run out of stack space and fail if `fm_significant_digits` is set too high.

There are two sample programs that each contain the same six example computations using `FM_parallel.f95`. One uses coarrays and the other uses openmp.

To illustrate the speedup from using multiple threads, the first four examples all compute the sum of the first 3 million terms of the harmonic series.

Example 1 uses a single thread and makes explicit calls to the low-level FM routines.
Making explicit calls can be faster for some compilers.

Example 2 uses multiple threads and makes explicit calls to the low-level FM routines.

Example 3 uses a single thread and uses the `fmzm_parallel` interface.
Using the interface is much easier to code and is the usual way to use the FM package.

Example 4 uses multiple threads and uses the `fmzm_parallel` interface.

Here are the times in seconds using two different compilers and openmp or coarrays. They come from a 2021 Mac laptop and used `fm_significant_digits = 50` with 10 threads.

	gfortran openmp	nagfor openmp	nagfor coarrays
Ex 1	0.82	1.03	1.06
Ex 2	0.12	0.16	0.14
Ex 3	1.08	1.16	1.17
Ex 4	0.19	1.71	1.68

The slow parallel times for nagfor in Example 4 are caused by contention among the different threads while allocating local variables. The NAG developers say (8/2023) that this problem should be resolved in the future.

Examples 5 and 6 of the Sample programs show a more realistic use of parallel FM. The problem is to solve a heat flow partial differential equation in order to find the time that the center of a 2x2 square plate reaches a certain temperature.

Example 5 makes a single call to subroutine `heat_time` using a 41×41 grid to approximate the solution. This gives only a bit less than 3 digits of accuracy for the critical time.

Example 6 makes multiple calls with increasing numbers of grid points, then fits an error formula to that data to produce 23 accurate digits for the critical time.

This last example shows how to use routines from the `FM_Sample_Routines.f95` collection, since many of those routines require a function defined in the user's program to be passed as an input

argument. User functions that are called from one of these routines must be written using explicit calls, like Examples 1 and 2.

There are also two test programs for parallel testing of the routines in `FM_parallel.f95`. I have checked the `TestFM_parallel_openmp.f95` program using gfortran and nagfor, and I have used nagfor to check `TestFM_parallel_coarray.f95`.