Distributed Memory Parallelism in NAME

Kevin Stratford, EPCC
Ben Devenish, Ben Evans, Matthew Glover, Andrew Jones, Dave Thomson, UKMO

April 2018

AN ARCHER ECSE PROJECT

Contents

1 Introduction **2**

2 Distrubuted Parallelism **2**
2.1 General comments 2
2.2 The message passing interface context 2
2.3 Logging and Errors 3
2.4 Random number generation 3
2.5 Particle storage and memory management 4
2.6 Patch structure 4
2.6.1 Decomposition 5
2.6.2 Eulerian case 6
2.6.3 Life Cycle and Particle Storage 6
2.7 Particle communication 7
2.7.1 Data types 7
2.7.2 Message passing 8
2.8 Release of particles in parallel 9
2.9 Chemistry 9
2.9.1 Patches 9
2.9.2 Particle location in the patch structure 10
2.9.3 Chemistry state 10
2.9.4 Chemistry update 10
2.9.5 Splitting 11
2.10 Output 12
2.10.1 Lagrangian quantities 12
2.10.2 Eulerian quantities 12
2.11 Testing 12
2.11.1 Assertions 12
2.11.2 Unit tests 12
2.11.3 Reproducability 13

3 Benchmarks **13**

3.0.1 Hardware . . . 13
3.1 Standard VAAC benchmark . . . 14
3.1.1 ARCHER results: standard VAAC benchmark . . . 14
3.1.2 Cirrus results: standard VAAC benchmark . . . 16
3.2 Enhanced VAAC benchmarks . . . 16
3.3 Chemistry benchmark . . . 17
3.3.1 ARCHER results: Chemistry benchmark . . . 17
3.4 Standard greenhouse gas benchmark . . . 19
3.4.1 ARCHER results: standard greenhouse gas benchmark . . . 19
3.4.2 Enhanced greenhouse gas benchmarks . . . 19
3.5 Conclusions . . . 20
3.5.1 Advice to users . . . 20
3.5.2 Known Issues . . . 22
3.5.3 Summary . . . 22

A Appendix **23**
A.1 ARCHER Environment . . . 23
A.2 Cirrus Environment . . . 24

1 Introduction

This work describes the distributed memory parallelisation of the NAME model. It is assumed the reader is familiar with the existing threaded implementation [1].

The following section provides a description of the components developed to handle the message passing implementation, and the context in which they are used. (It is also assumed that the reader is familiar with message passing concepts.) A further section discusses a number of standard benchmarks to assess the performance of the new parallel code and identifies residual performace bottlenecks.

The final section discusses a number of open issues, advice for users, and known issues.

2 Distrubuted Parallelism

2.1 General comments

A distributed memory implementation naturally involves a certain degree of complexity above and beyond that of a serial code. To manage the complexity a number of elements of "infrastructure" were developed individually. These are discussed in the following sections.

2.2 The message passing interface context

A number of simple container types to hold information on the message passing interface (MPI) environment are provided. These include a `NameMPIComm_` which aggregates communicator handle, size, and rank.

An overall context NameMPIContext_ is provided to aggregate information on a number of communicators. While these communicators may ultimately be relevant for the code, only one duplicated from MPI_COMM_WORLD is active at the time of writing.

The context is initialised once following MPI_Init(), and remains unchanged thereafter.

2.3 Logging and Errors

Logging of messages to standard output and standard error, and to designated files is provided in parallel by a MsgHandler_ object. Options for file output to a single MPI/IO shared file, or to one file per MPI rank via standard Fortran I/O are provided.

A message handler must be initialised in the parallel context,

```
   globalMsgHandler = msgHandlerInit(globalMPIContext)
```

after which messages to standard output are available via

```
   call message(messageText, ...).
```

The log and error file handles are initialised in a separate step:

```
   call msgHandlerSetUp(globalMsgHandler, ...)
```

after which message() will propagate to the log file and error file (as appropriate) in addition to standard output.

For backwards compatibility with the serial version, where the sole method for issuing messages was

```
   call message(messageText, errorCode, ...)
```

the call injects the globalMsgHandler object and defers action to the procedure msgHandlerMessage(...). message() retains integer errorCodes (cf. ErrorCode_ defined below). The msgHandler_ object carries relevant state including the current worst error code to date on a per-rank basis.

It is expected that normal termination will give rise to a sequence of:

```
   call msgHandlerSynchronise(...)
   call msgHandlerExitMessage(...)
   call msgHandlerFinalise(...)
```

to produce a consistent picture of the worstErrorCode on all ranks and an appropriate closing message.

2.4 Random number generation

Random number generation for particle updates is performed on a per-particle basis with the current state being a component of the Particle_ type:

```
   type :: Particle_
     ...
     integer (int32) :: iRNG(2)
     ...
   end type Particle_
```

This allows both thread-safe [1] per-particle updates, and the communication of the state `iRNG` with the particle structure if the particle needs to be moved between MPI tasks. The random number generator itself is L'Ecuyer (1988).

Note. The implementation of the `GaussB()` always discards one random variate (presumably for thread-safety reasons in a previous implementation). An alternative implementation

```
    mathsRandomGaussian2()
```

is provided which allows both to returned to the caller. For convenience there is also `mathsRandomGaussian1()` returning one, and overloaded versions to allow real32/real64 arguments.

2.5 Particle storage and memory management

An 'array' container is provided to hide details of memory management related to particles from the application space.

Storage for particles, extras, and masses is provided via the `ParticleArray_` type, which may be initialised with a maximum capacity per MPI rank, e.g.:

```
   a = particleArrayInit(maxParticles, maxExtras, nSpecies)
```

The preferred way to "add' and 'remove' (perhaps better named as 'activate' and 'deactivate') particles is via

```
   call particleArrayAddParticleExtraMass(a, particle, extra, masses)
   call particleArrayRemoveParticle(a, iP)
```

where `iP` is a local particle index in storage Using this interface will prevent accidents involving memory management. (However, clients are still responsible for thread safety at the moment.)

Restart content write/read for existing objects are provided via

```
   call particleArrayRestartWrite(a, unit, ierr)
   call particleArrayRestartRead(a, unit, ierr)
```

Resources are released via

```
   call particleArrayRelease(a)
```

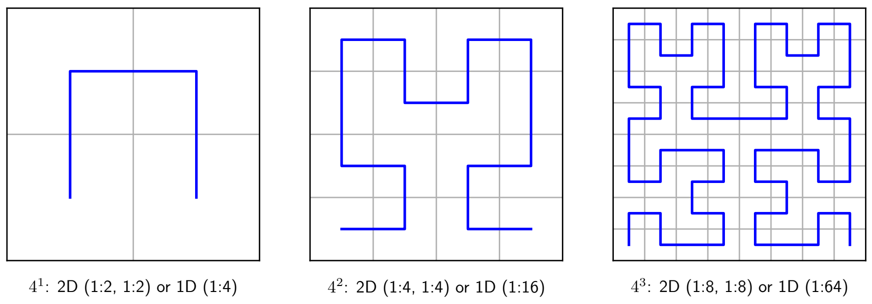
when no longer required.

2.6 Patch structure

A facility to store particles in "patches". The patch structure provides a way to organise the decomposition of particles in parallel. A number of patches is defined by the user (specified by a single integer nlevel, whence the number of patches is `nPH = 4**nlevel`) and may be viewed in either a one-dimensional picture `1:nPH` or a two-dimensional picture `(1:nxPH, 1:nyPH)` with `nxPH = nyPH = 2**nlevel`.

The mapping between the one-dimensional picture and two-dimensional picture is that of a standard Peano-Hilbert curve. (See Fig. 1.)

Assignment of particles to patches can take place in two different ways.



4^1: 2D (1:2, 1:2) or 1D (1:4)
4^2: 2D (1:4, 1:4) or 1D (1:16)
4^3: 2D (1:8, 1:8) or 1D (1:64)

Figure 1: The Peano-Hilbert picture with (from left to right) 4^1 patches, 4^2 patches, and 4^3 patches. The one-dimensional picture is represented by the bold blue line. Higher levels follow the same self-similar pattern.

1. Logical case (no Eulerian grid): particles are mapped to the logically one-dimensional patch structure via unique particle index `iUP` and appropriate modular arithmetic. The mapping is provided by `patchLogicalIndex()`.
2. Eulerian grid. Particles within the horizontal extent of the Eulerian domain are mapped to the two-dimensional patch structure based on their horizontal position in the Eulerian coordinate system.

2.6.1 Decomposition

Patches, and therefore particles, are always distributed between the available MPI ranks in the one-dimensional Peano-Hilbert picture `1:nPH`. Any given rank will "own" a number of patches `nLocal` being `[myMin:myMax]` in the global picture with offset `myOffset`. Patches are allocated to MPI ranks in contiguous blocks with rank 0 taking the left-most patches in the one-dimensional picture, and so on. (See figure 2.)

At the moment, no attempt at dynamic load balance is present, and the decomposition is static. The default decomposition simply attempts to balance the number of patches between MPI tasks. In the logical particle mapping, this should automatically provide each patch with approximately equal numbers of particles, and so balance load (in terms of particle number at least) between MPI tasks.

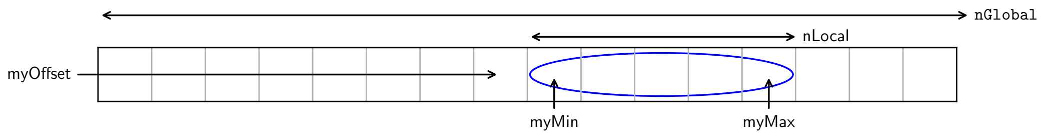


Figure 2: A schematic showing the a decomposition of patches between MPI tasks in the one-dimensional Peano-Hilbert picture. A given MPI task owns a number of patches (blue oval) with a global offset.

2.6.2 Eulerian case

If present, a logically rectangular horizontal Eulerian grid is mapped to the logically square two-dimensional Peano-Hilbert picture in blocks (of type `PatchGridBlock_`). Each block is itself logically rectangular and has an offset position in the full Eulerian horizontal grid. (See Fig. 3.)

This allows particle horizontal position to be mapped to the two-dimensional patch picture. For chemistry, this means that all particles within a given Eulerian grid cell are held on the same patch, and therefore the same MPI task. Particles in each horizontal patch (or column) are further mapped to an individual vertical level or grid cell (see "Chemistry" below).

A given simulation may involve particles which can cross the boundary of the horizontal Eulerian domain. In this case, particles outside the Eulerian domain map to patches using the logical mapping, while particles inside the Eulerian domain map using the "geographical" picture. All particles map to a patch, and a given patch may contain both "logically" and "geographically" mapped particles. Movement of particles between patches is handled in `Case.F90` and MPI communication in `mpiPatchTransfer.f90`.

Note that if the number of MPI tasks is 2^n, a uniform decomposition in the one-dimensional patch picture corresponds to a regular decomposition in the two-dimensional picture. This in turn corresponds to a regular decomposition of the horizontal Eulerian domain (if present). However, for a general one-dimensional decomposition, this is not true.

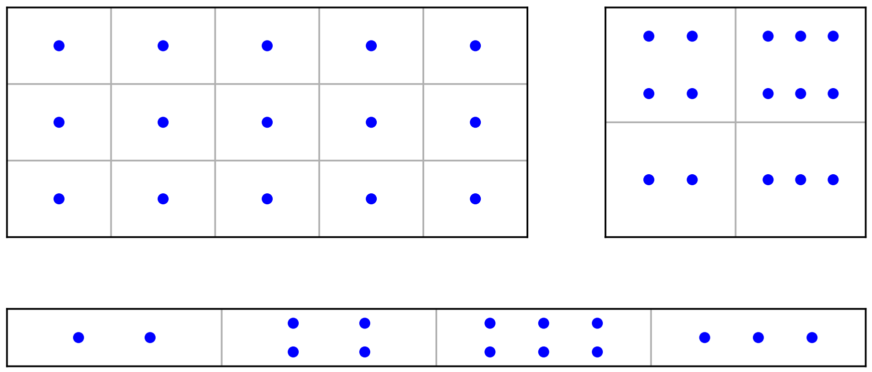


Figure 3: A schematic showing the a decomposition of Eulerian grid points (with a 5×3 horizontal Eulerian grid, top left) in the two-dimensional Peano-Hilbert picture (top right) and the one-dimensional picture (bottom) with `nlevel = 1`, ie., 4 patches.

2.6.3 Life Cycle and Particle Storage

The patches structure is instantiated via

```
  patches = patchesInit(ctxt, nlevel [, patchGridArg])
```

`nlevel` is chosen by the user, and controls the number of patches, which is `4**nlevel` in the one-dimensional PH picture (or `2**nlevel` × `2**nlevel` in the two-dimensional picture).

If an Eulerian grid is present (ie., chemistry is active) an optional `PatchGrid_` argument provides a container to supply details.

A particle may be located in the patch structure via a call

```
  iPatch =  patchIndex(patches, coords, particle)
```

Storage in the patch structure is currently an integer indirection to the main array of particles. A number of mechanisms are provided to add particles, the basis of which is to add the integer indirection `iP` to an individual patch:

```
  call patchAddParticle(patches%patch(iPatch), iP)
```

Particles are removed by marking `patch%iP() = PATCH_INDEX_NULL` followed by whole patch deletion operation

```
  call patchClearOut(patches%patch(iPatch))
```

A future implementation might consider providing one particle array per patch for actual storage (see `particleArray.f90`).

Patch resources are released via

```
  call patchesRelease(patches)
```

2.7 Particle communication

2.7.1 Data types

Particle communication is achieved via MPI derived data types. Individual message components are provided by defining MPI data types (MPI opaque handles) corresponding to each of:

```
  ShortTime_
  Particle_
  Extra_
  ParticleMessage_
```

Message data may be aggregated in the form of a fixed size `ParticleMessage_` container type. This has its own MPI data type to allow communication of a complete particle or array of particles.

Data type handles are obtained by the application via an `ParticleMPIDataTypes_` container, e.g.:

```
  handles = mpiParticleDataTypes()
```

and released (to avoid leaking handles) via

```
  call mpiParticleDataTypesRelease(handles)
```

There is little or no overhead observed in this process.

2.7.2 Message passing

Communication of particles between ranks is closely related to the patch structure and therefore the decomposition of patches. Transfers of particle information between patches may occur as and when particles change position and hence patch.

There are broadly two cases to be considered:

1. "Internal" transfers where the old patch and the new patch are held on the same MPI rank.
2. "External" transfers where the old patch and the new patch are held on different MPI ranks and require information to be passed via an MPI message.

Two data structures are provided to facilitate this process. The first is a message meta-data container `TransferListItem_`. This stores the identity of the old patch, the new patch, the new MPI rank, and so on, for each particle that needs relocation.

The entire exchange is encapsulated via a transfer request object `TransferReq_`, which holds relevant lists of particles, message buffers and so on.

A typical message exchange might follow a change in particle positions. With an existing patches structure, a `TransferReq_` object is created:

```
req = patchTransferRequest(patches)
```

Each local particle should then be interrogated to see if the new position necessitates a change in patch. If so, a meta data item is added to the request; schematically:

```
if (internal) call patchTransferRequestAdd(req, tid, item)
if (external) call patchTransferRequestAdd(req, tid, item, data)
```

Threading (if required) is explicitly included via the thread index tid. Note only external transfers require the actual particle data; internal transfers use the meta data item to relocate the particle in the patch structure via shared memory.

Clients may call

```
call patchTransferSendRecv(patches, req, coords, pa)
```

to execute the transfer in one go.

Communication is actually a three-stage process to allow possible overlap of communication with independent work:

```
call patchTransferRequestSend(patch, req)
...
call patchTransferInternal(patch, req%intList)
...
call patchTransferRequestRecv(patch, req)
```

When the final receive has returned, the client may use the updated particle data (incoming particles have arrived and outgoing particles have been removed). `patchTransferSendRecv()` combines all three steps.

With either mechanism, the request must be completed via a call to

```
call patchTransferRequestRelease(req)
```

to release resources.

2.8 Release of particles in parallel

All sources are replicated so that source information is available on all MPI tasks. Particle release from sources is also replicated on all MPI tasks: every task generates all particles. Once a particle has been generated, it can be examined to see which MPI rank should "own" the particle.

The owner actually stores the particle, while all other MPI simply ignore it. This process replicates generation of unique particle indices (`iUPs`) and initial random number state for the particle. This allows a decomposition independent result which does not depend on the number of MPI tasks.

Once appropriate effective source parameters has been identified, there are two steps:

1. `call caseReleaseParticle(...)`
   Assigns values to a particle, extra pair;
2. `call caseAddParticleExtraMassLocal(...)`
   Determines which patch, and therefore which MPI rank, owns the particle, and arranges storage as required.

The current number of particles globally (before any new ones are released) is synchronised at the start of the process to allow a check against the `maxParticles` limit. A similar check is present for the extras. No particle storage is actually 'allocated' until the point at which ownership is determined.

2.9 Chemistry

2.9.1 Patches

A number of additional data structures are used to construct chemistry "patches". The first is the `PatchChemistry_` type, which holds an array of type `PatchColumn_`, each of which represent a vertical column at a given horizontal position in the Eulerian grid. Each column allows assignment of particles to individual vertical levels.

The horizontal decomposition in the chemistry patches follows the decomposition in the Peano-Hilbert picture. In general, each PH patch holds a block of horizontal Eulerian grid points which must be further one-dimensionalised via the `patchGridBlock1D()` mechanism. The number of chemistry patches (i.e., horizontal grid points in the Eulerian grid) is then `chem%nLocal`.

A one-dimensional loop over local chemistry patches or columns is then typically

```
do iC = 1, chem%nLocal
  iX = chem%patch(iC)%iXE  ! Eulerian iX
  iY = chem%patch(iC)%iYE  ! Eulerian iY
  ! ... and a loop over the vertical
  do iZ = 1, chem%patch(iC)%nZ
    ...
  end do
end do
```

where `chem%patch(iC)%iXE, iYE` provide location in the horizontal Eulerian grid.

A utility `PatchMap_` structure allows mapping between the horizontal Eulerian grid `(iX, iY)` to the one-dimensional horizontal patch (Peano-Hilbert) picture.

2.9.2 Particle location in the patch structure

Particles are assigned to the appropriate horizontal patch at the point of release. Allocation of particles to the appropriate chemistry grid box at the current time step is then performed as a two-stage process:

1. `call caseAssignParticlesToH(...)`
   re-assigns particles to horizontal patches to allow for any change in position. This may require communication.
2. `call patchAsssignParticlesToZ(...)`
   Having been sorted in the horizontal patch structure, particles involved in chemistry may be further sorted in the vertical.

The chemistry update can then take place local to each MPI task. For the purposes of output, the global Eulerian concentration is reconstructed on rank 0 (only).

2.9.3 Chemistry state

Quantities which are required by the chemistry procedure (above and beyond those coming from the chemistry scheme itself) and that have memory between one chemistry update and the next are held by the object `ChemState_`.

The `ChemState_` object may be initialised with knowledge of the `ChemistryDefn_` and the `Patches_` structure:

```
   chemState = chemistryStateInit(chemistryDefn, patches)
```

and is updated in the `chemistryUpdate()`. Resources are released via

```
   call chemistryStateRelease(chemState)
```

The `ChemState_` object holds local quantities in the one-dimensional chemistry patch decomposition with extent `1:chem%nLocal` on each MPI task. The local quantities include

```
   chemBackField(1:nBackFields, 1:nZ, 1:nLocal)
   deltaO3(1:nZ, 1:nLocal)
```

being the background field concentrations and an ozone-related field. A replicated `initBackFields` flag controls initialisation.

2.9.4 Chemistry update

Routine `chemistryUpdate()` updates particle masses, the chemistry state, and the Eulerian field concentrations locally on each MPI rank. Threaded parallelism is used to further decompose the local chemistry patches. Note that the grid box volumes are recomputed at each chemistry update to allow that they vary in time. (There is no significant penalty in doing this if the vertical grid is fixed). Chemistry grid boxes with no particles are treated via the back fields, while others are treated using the currently active chemistry scheme.

Unified model data is (potentially) read from file at each update on each MPI task and the locally relevant data extracted in `chemsitryBackgroundFieldsFromUM()`.

The Eulerian field concentrations are global in extent. This means only the relevant portion of the Eulerian field can be updated on any one MPI task.

It is assumed that the patch description of the disposition of particles is up-to-date on entry.

2.9.5 Splitting

The splitter utility is provided to help to produce a decomposition independent particle splitting procedure for particles which carry mass above and beyond a prescribed limit. The particle mass limit is based on PM10 in this implementation, but other criteria could be imagined.

Each MPI rank determines which local particles meet the criteria for splitting. A globally replicated list of relevant particles may then be constructed which allows generation of new particles to be replicated on all MPI tasks. Global agreement on this list allows a decomposition independent result.

Construction of the global list relies on a sort, which is currently carried out in a naive implementation which is suitable for a modest number of particles.

The call sequence should first create a "splitter" object with the appropriate with communicator and list of active species:

```
splitter = splitterPM10(communicator, specieses)
```

Each rank is then required to determine the number of local particles which meet the criterion for splitting via `splitterYield()` and the consequent number of new particles. These are collected by a call

```
call splitterCollect(splitter, nLocal, nNewLocal)
```

which performs the necessary reducetion to provide a global total. splitterCollect() also allocates space for a local list of particle meta data which will allow the global list to be formed.

Each rank should then undertake a second sweep through local particles to assign the relevant meta data items in the `SplitterList_` data structure `splitter%listLocal(1:nLocal)`. A call to

```
call splitterMerge(splitter)
```

generates a replicated global list `splitter%listGlobal(1:nGlobal)`. This may be used to perform replicated actions on particles to be split.

Resources are released via

```
call splitterPM10Release(splitter)
```

The splitter has no direct dependence on the method of storage of particle information here beyond the assumption that a rank 1 array `masses(:)` holds particle masses with indices as specified in `specieses%iSpecies2Particle(:)`.

The full splitting procedure is implemented in `caseSplitMassiveParticles()`.

2.10 Output

2.10.1 Lagrangian quantities

Contributions to Lagrangian output quantities may occur on all active MPI tasks with particles. An MPI operation is therefore (potentially) required to consolidate local contributions to a single aggregate field on rank zero. This field is suitable to process for output as in serial.

For each Lagrangian field, a reduction operation, which is determined for individual requests via `fieldReqLagrangianQuantityToMPIOp()`, takes place at appropriate time steps.

2.10.2 Eulerian quantities

If chemistry is active, each MPI task makes local contributions to the replicated global Eulerian concentration. A "gather" operation is therefore required to reconstruct a unified picture of the concentration (on rank zero only). The concentration is then appropriate for output on rank zero if required.

No action is required in serial.

2.11 Testing

2.11.1 Assertions

A C-like `assert()` macro is provided via C-preprocessor directives in the file `assertion.h`. The actual implementation is via Fortran stop in the external procedure `assertion_stop()`.

A wrapped version `assertion()` declared with a pure interface is provided to allow assertions to be placed in pure procedures. This is a slight-of-hand which may generate a compiler warning (e.g., with Cray `ftn`).

2.11.2 Unit tests

A set of unit tests are provided by the modules `test*.f90` in the test directory. Each module provides a driver `test*Run()` which takes a `NameMPIContext_` as the sole argument. The driver is responisble for running individual tests.

Test failures are reported by means of the non-fatal assertion `nameMPIContextAssert()`.

Tests may be run in serial or in parallel with a arbitrary number of MPI tasks/OpenMP threads (up to a reasonable maximum).

A number of the tests require reference data files to exist (the tests will be skipped if the files are not present).

The overall test driver is `test.f90`.

2.11.3 Reproducability

Owing to a number of changes, the updated version of the code does not produce exactly the same results as the original version. Development was staged so that the changes were introduced episodically, allowing reproducability to be checked up to the point of the change, and again after the change. There were three such changes. First, and fundamentally, is the change in random number generation associated with the removal of the global random state (used at particle release). Second and third, and relating only to chemistry, are changes in the negative mass redistribution of species when re-associated with particles at the end of the chemistry step, and particle splitting. In both these cases, the new algorithm maintains reproducabilty independent of decomposition of work between threads and/or MPI tasks.

As a separate measure of reproducability, it has been verified that the results are the same (to within possible round-off) independent of the number of threads and/or MPI tasks for the standard benchmark problems.

3 Benchmarks

The following sections provide a summary of the benchmarks performed as part of the work. For brevity, details of the protocol for each benchmark on both ARCHER and Cirrus are relegated to Appendix 1. Likewise, an abbreviated summary of timings is presented in each case to illustrate salient features. The timings presented in the summary are typically based on four repeat runs of each benchmark (unless otherwise stated).

The description covers a number of "standard" benchmarks which have been used to assess the performance of the distributed memory version against the starting version. These standard benchmarks are, of necessity, limited in size. Later sections go on to describe a series of "enhanced" benchmarks which are designed to exercise the increased memory capacity of the distributed memory version (and so may not be practical in the original version).

For each benchmark, performance of version 7.2 ("SVN 68") is compared with an updated version (e.g., "SVN 359"). Benchmarks are typically run in serial (1 thread), on 1 socket (12 and 18 threads on ARCHER and Cirrus, respectively), and on a full node (2 sockets). All benchmarks run 1 thread per core, unless it is explicitly stated that two hardware threads per core are utilised. When run on more than one MPI task, benchmarks generally seek to use one MPI task per node, and one thread per core within the node. Exceptions are noted in the text.

3.0.1 Hardware

The two hardware platforms considered are:

1. UK Tier-1 ARCHER. This will refer to the main Cray XC30 service machine with nodes containing two 12-core Intel Xeon E5-2697 v2 (Ivy Bridge) chips. Each core is clocked at 2.7 GHz and can support two threads in hardware. Within a node, two sockets provide access to two 12-core chips connected by two QuickPath Interconnect (QPI) links. Each core can issue 4 double precision floating point multiply and add operations per cycle (8 flop/cycle), providing 21.6 GFlop/s per core, or roughly 0.52 TFlop/s per node. The thermal design power rating is 260 W per node.

2. UK Tier-2 Cirrus. One node has two 18-core Intel Xeon E5 2695 v4 (Broadwell) processors. Each core runs at 2.1 GHz and supports 2 threads in hardware. Each core can issue 4 double precision floating point multiply-add instructions per cycle (vector length 4 doubles), giving 16.8 Gflop/s per core, or about 0.60 TFlop/s per node. Two sockets have thermal design power rating of 290 W.

ARCHER and Cirrus have 64 GB and 256 GB DRAM per node, respectively.

3.1 Standard VAAC benchmark

The benchmark represents an eruption of Askja in central Iceland for the Volcanic Ash Advisory Centre (VAAC). No chemistry is present. Meteorological data are "cut out" from the Unified Model, there being 14 cut-out parts globally. The update-on-demand mechanism means one part (the North Atlantic including Iceland) is required at the start of the run, increasing to five parts (the entire Northern Hemisphere above the tropics) at the end of the run as ash is swept away from the source. Each part consists of around 250 MB of data. Data from 00:00 4th July 2015 is at 3-hourly intervals.

Initially there are no particles; particles are then released at a constant rate of 15,000 per hour, with the total number of particles reaching a maximum of 720,000 after around 48 hours after which the eruption ceases. The maximum number of particles specified in the input file is 10 million.

To provide a benchmark of around 1–2 hours in serial, NAME is run for 6 days model time; this requires 136 "met modules" to be read from disk.

3.1.1 ARCHER results: standard VAAC benchmark

A summary of the results for versions SVN 68 and SVN 324 of the code is given in Table 1. For the standard VAAC problem, the predominant sections of the code are the particle time step loop (LPPT) and the processing and output of data to file (P/Output) which are shown in the Table. Not included is a small (serial) overhead for the particle release mechanism (about 30 seconds). Also not shown is the relatively small contribution of the computation of particle results (around 50 seconds in serial); however, there is relatively little thread-level parallelism in this code owing to a significant number of critical sections.

The benchmark shows that in serial version at SVN 68 , a significant fraction of the time is spent in the particle loop (LPPAT). This is confirmed by a MAP profile, which shows the breakdown includes `oneParticleTimeStep` at 47%, and `calcParticleResults` at 29% of the total time. The input time for the meteorological data (here subsumed into the LPPAT figure) is around 250 seconds (with some variability at the 10% level attributed to file system performance).

In the SVN 68 results the LPPT section scales reasonably: the speed up on 12 threads is 7.4. However, there is little improvement beyond one NUMA region. The MAP profile suggests potential issues with synchronisation in `whichFlow()` (rising to 15-20% of the total execution time on 12-24 threads) are the source of this limitation. In contrast, the SVN 324 version (which removes a number of synchronisation issues) performs more robustly (a speed up 16 on 24

Execution Model	Section	SVN 68		SVN 324	
		Time (s)	Speed up S_1	Time (s)	Speed up S_1
Tasks: 1 Threads: 1	LPPT	5900	1.0	6420	1.0
	P/Output	510	1.0	630	1.0
	Total	6600	1.0	7200	1.0
Tasks: 1 Threads: 12	LPPT	800	7.4	660	9.7
	P/Output	380	1.3	310	2.0
	Total	1320	5.0	1060	6.8
Tasks: 1 Threads: 24	LPPT	550	10.7	400	16.0
	P/Output	430	1.2	310	2.0
	Total	1130	5.8	810	8.9
Tasks: 2 Threads: 24	LPPT	—	—	250	25.7
	P/Output	—	—	320	2.0
	Total	—	—	660	10.9

Execution Model	Section	SVN 68		SVN 324	
		Time (s)	Speed up S_1	Time (s)	Speed up S_1
Tasks: 1 Threads: 1	LPPT	5350	1.0	5270	1.0
	P/Output	390	1.0	430	1.0
	Total	5890	1.0	5820	1.0
Tasks: 1 Threads: 18	LPPT	460	11.6	440	12.0
	P/Output	360	1.1	210	2.0
	Total	940	6.3	500	11.6
Tasks: 1 Threads: 36	LPPT	—	—	190	27.7
	P/Output	—	—	210	2.0
	Total	—	—	500	11.6
Tasks: 2 Threads: 36	LPPT	—	—	110	47.9
	P/Output	—	—	230	1.9
	Total	—	—	420	13.9

Table 1: Top panel: ARCHER summary for a standard VAAC benchmark comparing SVN 68 with the distributed memory version at SVN 324. Times in seconds are recorded for the particle time step loop (LPPT), processing and output of data to file (P/Output) and the overall total time to solution. Four runs are presented (from top to bottom): 1 MPI task and 1 OpenMP thread (ie., in serial), 1 task and 12 threads, 1 task and 24 threads, and 2 MPI tasks, each with 24 threads. In each case, a speed up S_1 is computed based on the corresponding serial time.
Bottom Panel: The same summary for Cirrus run on 1, 18, and 36 threads (single MPI task), and 36 threads on each of 2 MPI tasks. It was not considered worthwhile running a full benchmark test for 36 threads at SVN 68 owing to lack of scaling.

cores). The serial time is actually slightly slower owing to code changes, but the threaded parallelism is more efficient. Synchronisation remains in the on-demand reading of meteorological data.

The distributed memory version provides some benefit for this problem, but the scaling is ultimately hampered by serial output operations.

Execution Model	Section	BENCHMARK VE1		BENCHMARK VE2	
		Time (s)	Speed up S_1	Time (s)	Speed up S_1
Tasks: 1 Threads: 24	LPPT	4290	1.0	4400	1.0
	P/Output	220	1.0	1640	1.0
	Total	4940	1.0	6540	1.0
Tasks: 2 Threads: 24	LPPT	2180	2.0	2240	2.0
	P/Output	270	0.8	1780	0.9
	Total	2690	1.8	4320	1.5
Tasks: 4 Threads: 24	LPPT	1140	3.8	–	–
	P/Output	300	0.7	–	–
	Total	1580	3.1	–	–

Table 2: Enhanced VAAC benchmarks VE1 (left) and VE2 (right) run on ARCHER. Times in seconds for the given sections of the code are based on four repeats in each case. The speed up S_1 for each section is based on the corresponding time on 1 node (24 threads). It was not considered worthwhile extending VE2 to 4 nodes owing to lack of scalability with increased output grid resolution.

3.1.2 Cirrus results: standard VAAC benchmark

The corresponding results for the same standard VAAC benchmark run on the Tier-2 Cirrus system at SVN 68 and SVN 324 are also summarised in Table 1. We note that it was not considered worthwhile running a full benchmark test for the SVN 68 version on 36 threads owing to lack of scaling.

The picture here is similar to that seen on ARCHER. The overall time-to-solution is slightly improved compared with ARCHER, which is probably attributable to improved memory performance. Times in serial for the two SVN versions are similar, and threaded parallelism is again reasonably effective in the LPPT section. Distributed memory parallelism is also reasonably effective here in the LPPT section. Overall distributed memory scaling is restricted by residual serial fraction including the output and the release of particles.

3.2 Enhanced VAAC benchmarks

The standard VAAC benchmark takes around 4GB of memory (high-watermark) on a single node; there is clearly no need to extend a benchmark of this scale to more than one node for the purposes of providing more memory. To test performance against larger problem sizes, two enhanced benchmarks are considered. The first, denoted benchmark VE1, increases the number of particles by a factor of 10, simply by increasing the source production rate (150,000 per hour cf. 15,000 per hour). This gives a maximum of 7,200,013 particles. The second additional benchmark, denoted VE2, increases both the number of particles as in VE1, but also increases the output grid vertical resolution by a factor of 10 (220 levels at 250 feet resolution cf. 22 levels at 2500 feet resolution).

Each of the benchmarks is run at SVN 359 on 1 and 2 nodes of the ARCHER system using 24 threads on each node. The results were extended to 4 nodes for the VE1 benchmark only. In each case the Cray compiler environment was used. A summary of the results is presented in Table 2. As in the standard VAAC benchmark, the time is predominantly in LPPT and processing output, so figures are limited to these sections.

There are two clear results. First, the time for LPPT scales well as the number of MPI tasks is increased. This is not unexpected as the size of the problem as been increased. For benchmark VE1, the second clear result is that the time to process the output increases with the number of MPI tasks. While the output itself is serial, an increase is to be expected as the `MPI_Reduce()` required for each output request generates a significant communication overhead (some of the messages are large). For benchmark VE2, where the increase in the output grid size is a factor of 10, there is a similar increase in the cost. This means output to file completely dominates the scaling.

3.3 Chemistry benchmark

The chemistry benchmark is based on a regional model for the United Kingdom which extends to north west Europe (roughly 14W–20E, 42–62N), having resolution in the horizontal Eulerian grid of 226$\times$222 points (around 10 km). There are 18 vertical levels with varying resolution extending to around 5000 m above ground level. The chemistry scheme itself [3] includes 42 different chemical species.

The model is driven by wider-area regional meteorological data at resolution of around 0.5^o in the horizontal and 50 vertical levels. The extent is roughly 100W–25E and 10–80N. Data is at three hourly intervals from 00:00 on 1st June 2008. The time extent of the benchmark was chosen to provide a serial benchmark taking about 1–2 hours in wall clock time in serial on the relevant hardware. The final choice was to to read six data sets (each around 50MB) representing around 15–18 hours model time.

There are a large number of sources which account for the generation of particles carrying the various chemical species. These sources include those representing biological land surface process at specific locations, pollution sources such as traffic, and so on. Sources are time dependent and relevant for the period of interest.

The input gives rise to a number of particles reported at around 4.1 million initially, rising to 7.7 million by the end of the run. It should be noted that a significant number of particles are generated and then immediately removed; the total number of particles generated in the course of the simulation is around 180–200 million.

The final memory requirement for the benchmark on ARCHER is around 5.8GB, rising slowly with the number of particles at the end of execution.

3.3.1 ARCHER results: Chemistry benchmark

Results are reported for the chemistry benchmark in Table 3, which is a comparison of the original code at SVN 68, and an updated code at SVN 324. As the performance of the particle sections (LPPT etc), and output, are broadly similar in character to the VAAC case, we concentrate on the parts relevant specifically to the chemistry.

There are a number of issues to be highlighted in the results. In serial the benchmark takes around 6,000 seconds; at SVN 68 the main contributions to this total are around 2,500 seconds in the particle update and accumulation for output, around 1500 seconds for release from sources, and another 1500 seconds for the chemistry itself. Of the chemistry contribution, about 200

Execution Model	Section	SVN 68		SVN 324	
		Time (s)	Speed up S_1	Time (s)	Speed up S_1
Tasks: 1 Threads: 1	Total	5880	1.0	5770	1.0
	Release	1480	1.0	1320	1.0
	Chem. Tot.	1430	1.0	1150	1.0
	Binning*	220	1.0	270	1.0
	Loop	1210	1.0	850	1.0
Tasks: 1 Threads: 24	Total	2730	2.2	1760	3.3
	Chem. Tot.	960	1.5	63	18.3
	Binning*	220	1.0	16	16.9
	Loop	740	1.6	39	21.8
Tasks: 1 Threads: 48	Total	—	—	1740	3.3
	Chem. Tot.	—	—	55	20.9
	Binning	—	—	16	16.9
	Loop	—	—	29	29.3
Tasks: 2 Threads: 48	Total	—	—	1690	3.4
	Chem. Tot.	—	—	39	29.5
	Binning	—	—	13	20.8
	Loop	—	—	14	60.7

Table 3: ARCHER standard chemistry benchmark summary showing the time taken in seconds for various execution models (MPI tasks times threads). The detail shows times for the total time related to chemistry, which is made up of binning the particles to chemistry grid cells, the chemistry scheme itself ("Loop") and chemistry splitting/diagnostics. It can be seen that while the original SVN 68 has rather poor threaded performance, the updated code SVN 324 much better on a single node. Note the time for release of particles is shown only once, but is a constant serial overhead in all cases. Timings for SVN 68 marked * are based on profile data rather than internal timing.

seconds is required to bin particles to chemistry grid cells in the Eulerian grid, while the balance is the execution of the chemistry scheme itself (the "Loop"). It should be noted that the release presents a significant serial fraction (and which remains in all cases, so it is not repeated in Table 3). In the original code, the binning was also a purely serial overhead.

At SVN 68 the scaling of the threaded implementation of the chemistry is rather poor. A profile confirms that of the approximately 20% of time in the serial case spent in the chemistry update, some 9.1% and 6.2% are spent in `chemistryScheme()` and `calcZenithAngle()`, respectively. The profile further identified a significant issue with lock contention arising from calls `calcZenithAngle()` to routines which perform Fortran internal writes. These make use of the thread locking mechanism (presumably because some shared internal buffer is used in the implementation). The proportion of time spent in these routines increases with increasing number of threads (25% of the total on 24 threads).

The message passing version (SVN 324) uses a new implementation of the both the binning procedure and the chemistry loop. The binning procedure must be preceded by the communication required to ensure all relevant particles are resident on the appropriate MPI task, but can itself include threaded parallelism. Work on each chemistry grid box can then take place on individual MPI tasks in the chemistry loop. The threaded parallelism in the loop is improved by three main measures: 1. removal of lock contention arising from `calcZenithAngle()` via a new implementation; 2. removal of 4-dimensional arrays to store local chemistry concentrations (only a local vertical column is required); and 3. improved parallelism in the loop itself by moving to a one-dimensional decomposition. The time for the particle communication, along with splitting and related diagnostics, is accounted for under the Chemistry Total.

The results show that the threaded performance of the new implementation is reasonably effective. The degree of improvement makes it worthwhile to use 2 threads in hardware on each core (usually referred to as "hyperthreading" on Intel hardware), which provides some improvement in the chemistry loop itself. (This actually suggests there is still significant memory latency to be hidden in this part of the computation.) Increasing the number of MPI tasks provides some improvement in the chemistry (each task deals with part of the chemistry domain), but the overall performance is limited by the serial overhead of releasing particles from a very large number of sources in this benchmark.

3.4 Standard greenhouse gas benchmark

The greenhouse gas benchmark is an example of a "backwards-in-time" simulation that can be performed with NAME to assess the origin of material arriving at a point of interest. The domain of interest for this problem is again the UK and the surrounding North Atlantic region.

Meteorological data are high-resolution (around 1.5km in horizontal, and 57 vertical levels) at one hour intervals between 2nd September 2017 and 13th September 2017, and are based on "cut-outs" from a global domain. For the backwards in time simulation the meteorological data cannot be read on an "on-demand" basis; this means the read and process steps for all meteorological data are effectively serialised. This will become evident in the profiles.

There is effectively one source producing 20,000 particles per hour for the first 24 hours of the simulation, giving rise to a particle number which grows linearly to 480,000. The number is stable for around 48 hours, before falling as particles leave the domain of interest. Around half (240,000) are left at the end of the simulation.

3.4.1 ARCHER results: standard greenhouse gas benchmark

Results for the standard greenhouse gas benchmark run on 1, 12, and 24 threads on ARCHER are shown in Table 4. Again, focus is restricted to parts of the code which are significant for this particular problem. The results show performance is similar in the original and the new versions, with the main particle loop parallelising reasonably well. The clear bottleneck for this benchmark is the serialised reading and processing of the meteorological data. (This being the case, results for larger numbers of MPI tasks are not reported for this problem.)

3.4.2 Enhanced greenhouse gas benchmarks

The number of particles in the standard greenhouse gas benchmark (480,000) was introduced for a simulation using lower resolution meteorological data (around 10km in the horizontal). It is therefore reasonable to require an increase in the number of particles used by a factor of 10 in each horizontal direction. One could also require a further increase in the number of particles to increase statistical accuracy.

Execution Model	Section	SVN 68		SVN 413	
		Time (s)	Speed up S_1	Time (s)	Speed up S_1
Tasks: 1 Threads: 1	LPPT	5300	1.0	5660	1.0
	MetRead	650	1.0	640	1.0
	MetProcess	610	1.0	670	1.0
	Total	6850	1.0	7230	1.0
Tasks: 1 Threads: 12	LPPT	530	10.0	510	11.1
	MetRead	620	1.0	640	1.0
	MetProcess	610	1.0	670	1.0
	Total	1900	3.6	1970	3.7
Tasks: 1 Threads: 24	LPPT	360	14.7	270	21.0
	MetRead	620	1.0	650	1.0
	MetProcess	610	1.0	670	1.0
	Total	1730	3.1	1740	4.2

Table 4: Standard greenhouse gas benchmark run on ARCHER for SVN 68 and SVN 413 versions. Times in seconds are reported for the given sections of the code, along with a speed up S_1 computed relative to the corresponding section run in serial. Overall performance is limited by serialisation of the meteorological data read and process steps.

Two additional benchmarks are therefore considered which increase the number of particles by a factor of 100, and by a factor of 1000 compared with the standard benchmark. The two new benchmarks will be referred to as Gx100 and Gx1000, respectively. These benchmarks significantly increase the computational demand on the system with a number of consequences: 1. the serialisation of meteorological processing is "hidden" to a large degree; and 2. the distributed memory becomes more important for both time-to-solution, and total memory capacity. The benchmarks are implemented by increasing the source rate to 200,000 per hour, and 2 million per hour, respectively. All other simulation parameters are unchanged.

The two enhanced benchmarks are run using the Cray compiler environment (in contrast to the other reported results). The figures are shown in Table 5. All the benchmarks use entire nodes running 24 threads (one thread per core). Benchmark Gx100 is run on up to 4 MPI tasks, that is, 4 nodes (total 96 cores). Here, the overall proportional of the particle computation is higher and scaling is reasonable to 4 MPI tasks. At this point the serial overhead of the meteorological data (again around 1200 seconds) is large enough to start to degrade scaling. Higher numbers of MPI tasks were not attempted.

Benchmark Gx1000 requires at least 4 nodes to provide enough memory to run. Results are therefore reported for 4, 8, 16 and 32 MPI tasks (up to 768 cores with 24 threads per node). Again, the parallel scaling is reasonable. Ultimately limiting here are, in addition to the meteorological data (about 1200 seconds), the serial release episode (about 900–1000 seconds) and an increase communication overhead in the output section.

3.5 Conclusions

3.5.1 Advice to users

There are a number of choices associated with the MPI version that can be made.

Execution Model	Section	BENCHMARK Gx100		BENCHMARK Gx1000	
		Time (s)	Speed up S_1	Time (s)	Speed up S_4
Tasks: 1 Threads: 24	LPPT	31400	1.0	–	–
	CPRS	1600	1.0	–	–
	Total	34600	1.0	–	–
Tasks: 2 Threads: 24	LPPT	15400	2.0	–	–
	CPRS	770	2.1	–	–
	Total	18000	1.9	–	–
Tasks: 4 Threads: 24	LPPT	7700	4.1	75600	1.0
	CPRS	400	4.0	3890	1.0
	Total	9770	3.5	84400	1.0
Tasks: 8 Threads: 24	LPPT	3380	8.1	38300	2.0
	CPRS	210	7.6	1980	2.0
	Total	5910	5.9	46400	1.9
Tasks: 16 Threads: 24	LPPT	–	–	19500	3.9
	CPRS	–	–	1070	3.6
	Total	–	–	24000	3.7
Tasks: 32 Threads: 24	LPPT	–	–	9900	7.6
	CPRS	–	–	520	7.5
	Total	–	–	13600	6.2

Table 5: Enhanced greenhouse gas benchmarks Gx100 and Gx1000 with 4.8 million and 48 million particles, respectively. These results are run on ARCHER using the Cray compiler environment. Times for different sections of the code are given in seconds. Speed up S_1 is relative to the corresponding result with 1 MPI task for Gx100. For Gx1000, which requires at least 4 nodes for memory purposes, speed up S_4 relative to 4 nodes is presented.

1. The balance of threads and MPI tasks.
   Current advice must be to run with no more than one MPI task per shared memory node, and use one thread per core. For chemistry (alone) there may be scope to use more than one thread per core if supported in hardware.

2. Peano-Hilbert patches
   The number of patches is controlled by `nlevel`. For most purposes, a value of 4 (giving $4^4 = 1024$ patches globally) should be a good starting point. The number of MPI tasks should not exceed the number of patches (and should probably be much smaller than the number of patches).

3. MPI/IO.
   The choice `useMPIIO = .false.` should be used in all situations. Experimental use of MPI-IO for output is possible, but may cause degradation of performance (owing to synchronisation of file writes) if used when running with two or more MPI tasks.

4. Extras/Puffs
   In contrast to the previous version, specifying a number of full particle extras as zero in the input file will not result in a crash. The same if true of puffs.

3.5.2 Known Issues

There are a number of outstanding issues observed with the new version.

1. On some hardware platforms, MPI/IO using MPI shared files may not work at all (it requires specific flags to be set in the file system). Advice to users is to not to use MPI/IO.
2. Platforms using Intel MPI. It has been observed that the communication required for chemistry will deadlock when using more than two MPI tasks. We speculate that this is a problem in Intel MPI. Advice to users is to use a different MPI implementation (where the problem is not observed).
3. Considerable variability has been observed in the time taken in the process outputs stage. The root cause of this variability is not understood and requires further investigation. Advice to users is to limit output request where possible.

3.5.3 Summary

The report describes a new MPI implementation in the NAME code. Performance is characterised for a number of different benchmarks and some residual performance issues identified.

A Appendix

A.1 ARCHER Environment

We have compilation via Intel 17 and profiles via Allinea MAP 7.0. The relevant module environment was:

```
  1) modules/3.2.10.2
  2) eswrap/1.3.3-1.020200.1278.0
  3) switch/1.0-1.0502.60522.1.61.ari
  4) craype-network-aries
  5) intel/17.0.0.098
  6) craype/2.4.2
  7) pbs/12.2.401.141761
  8) craype-ivybridge
  9) cray-mpich/7.2.6
 10) packages-archer
 11) bolt/0.6
 12) nano/2.2.6
 13) leave_time/1.0.0
 14) quickstart/1.0
 15) ack/2.14
 16) xalt/0.6.0
 17) epcc-tools/6.0
 18) allinea/7.0
 19) cray-libsci/13.2.0
 20) udreg/2.3.2-1.0502.10518.2.17.ari
 21) ugni/6.0-1.0502.10863.8.29.ari
 22) pmi/5.0.7-1.0000.10678.155.25.ari
 23) dmapp/7.0.1-1.0502.11080.8.76.ari
 24) gni-headers/4.0-1.0502.10859.7.8.ari
 25) xpmem/0.1-2.0502.64982.5.3.ari
 26) dvs/2.5_0.9.0-1.0502.2188.1.116.ari
 27) alps/5.2.4-2.0502.9774.31.11.ari
 28) rca/1.0.0-2.0502.60530.1.62.ari
 29) atp/1.8.3
 30) PrgEnv-intel/5.2.82
```

Compilation at SVN 68 is with options

```
ftn -Wp,-macro=no -O3 -w -extend_source -nbs -auto -no-vec \
    -DIntelLinCompiler -sox -fp-model precise -DUseConvert \
    -openmp -DUSE_TIMERS
```

In the course of the work it was recognised that `-DUSE_C_FORMAT` was essential to avoid synchronising Fortran internal writes, and was used for all later version results.

Execution is via an `aprun` command of the form

```
export OMP_STACKSIZE=8M
export OMP_NUM_THREADS=12
aprun -n 1 -N 1 -d ${OMP_NUM_THREADS} -cc numa_node ${NAME_EXE} ...
```

Option `-cc none` was used for jobs requiring more than 12 threads.

The relevant number of threads was also specified in the NAME input file via e.g.,

```
OpenMP Options:
Use OpenMP?, Threads, Particle Update Threads
Yes,        24 ,                       24
```

Use of Cray compiler environment:

A.2 Cirrus Environment

We have compilation via Intel 17 and profiles via Intel VTUNE. THE SGI message passing toolkit was used (required for `omplace`). The relevant module environment was:

```
  1) mpt/2.14                         4) intel-compilers-17/17.0.2.174
  2) intel-cc-17/17.0.2.174           5) intel-vtune-17/2017.2.0.499904
  3) intel-fc-17/17.0.2.174
```

Compilation options are as for Intel discussed above. Execution is via

```
export OMP_STACKSIZE=8M
export OMP_NUM_THREADS=18
omplace -nt ${OMP_NUM_THREADS} ${NAME_EXE} ...
```

If Intel MPI is relevant, the module environment is

```
  1) intel-mpi-17/17.0.2.174        3) intel-fc-17/17.0.2.174
  2) intel-cc-17/17.0.2.174         4) intel-compilers-17/17.0.2.174
```

and execution is via, e.g.,

```
export OMP_STACKSIZE=8M
export OMP_NUM_THREADS=36

mpirun -np 2 -ppn 1 ./${NAME_EXE} ...
```

Notes and References

[1] Müller, E.H., R. Ford, M.C. Hort, L. Huggett, G. Riley, and D.J. Thomson, Parallelistion of the Lagrangian atmospheric dispersion model NAME, *Comp. Phys. Comm.*, **184**, 2734–2745 (2013).

[2] See, e.g., *Numerical Recipes in Fortran*, W.H. Press, S.A. Teukolsky, W.T. Vettering, and B.R. Flannery (Cambridge University Press).

[3] See e.g., A.L. Redington, and R.G. Derwent, Calculation of sulphate and nitrat aerosol concentrations over Europe using a Lagrangian dispersion model, *Atmos. Environ.* 4425–4439 (2002), and references therein related to STOCHEM.