Parallelisation with OpenMP

Eike H. Müller

Overview

This document describes the parallelisation of the NAME code with OpenMP.

It is split into four parts: Chapter 1 is a summary for users. We give a brief introduction to parallelisation of computer code with OpenMP, summarise the changes to the NAME code and describe how to compile and run the parallelised model.

Chapter 2 describes the parallelisation in more detail and in chapter 3 we present a detailled timing analysis for two operational scenarios, an 'air quality' and a 'volcano' run.

The first stage of this project was contracted out to Rupert Ford and Graham Riley of Manchester Informatics Ltd. at the School of Computer Science, University of Manchester, in 2008. A parallelisation strategy was developed and implemented for version 5.3 of the NAME code, section 2.2 summarises the key points of this strategy. Since then, these changes have been integrated in the current version (6.0) of the code and we addressed some shortcomings of the initial implementation. We also introduced additional improvements which lead to further performance gains, this is described in section 2.3.

The parallel code has been adapted for use in the operational air quality system. In section 3.1 we report on a detailled timing analysis of two typical benchmark runs and investigate the scalability of these benchmarks on the Atmospheric Dispersion Group's computer system.

A scaling analysis of the code that was used in the operational volcanic ash runs following the Eyjafjallajökull eruption on Iceland in April/May 2010 is presented in section 3.2. In this section we introduce some further changes to the output calculation.

In chapter 4 of this report we address current issues and suggests further improvements, in particular we investigate the still open question of the poor scaling of the parallelised chemistry loop. We suspect that this is due to bad memory utilisation and investigate the problem by measuring cache misses. A trial version of the Intel performance tool vTune was used at this stage of the work. By reordering the layout of some multidimensional arrays we are able to improve cache utilisation and speed up the chemistry loop by around 10%. It should be stressed that this should be seen as a first exploratory study with the main purpose of identifying the right tools for further profiling and performance improvement. We feel that further investigation of this issue is worthwile and that there are other parts of the code that can be improved in a similar way.

As efficient memory access and cache utilisation is vital for the performance of the NAME code we give a brief overview of modern CPU cache systems in appendix A. For future reference we list the input files for some of the benchmarks used in this report in appendix B.

Contents

Chapter 1

User summary

This chapter summarises some general background information on code parallelisation with OpenMP. Following this, the compilation of the parallelised NAME code and the input parameters controlling its behaviour at runtime are described. The chapter finishes with a description of the timer module and a discussion of bit reproducibility in parallel runs.

1.1 Background: Cores, threads and OpenMP

Most modern computer processors (or CPUs) consist of multiple *cores*, i.e. the basic compute units are duplicated several times on a single chip[1]. The Intel Xeon E5520 quadcore processor, which is the CPU of the machine this report was written on, consists of four identical cores. In the simplest setup each core executes a separate program, for example the NAME code can be run on core one while the results are analysed on a different core without impacting on the runtime[2]. This form of parallelisation is handled by the operating system. It is usually invisible to the user and does not require any modification to the source code.

On the other hand, a single program can launch multiple processes or *threads* during its execution, which are then distributed amongst the different cores. The simplest example is a `for` loop. If the individual iterations are independent, then the first can be executed on core one, the second on core two and so on. The tasks performed by the threads do not have to be identical, one thread might perform an IO task (such as reading MetData from disk) while two other threads propagate particles through the model atmosphere. Both strategies have been used for the parallelisation of NAME described in this report. This form of parallelisation requires changes to the source code. The resulting saving in runtime is a function of the number of threads and often depends on the machine the code is run on (see section 1.1.1).

At this stage the NAME code has been parallelised with OpenMP (Open Multi-Processing), which is a shared memory scheme. This means that in addition to its private data storage each thread has access to memory which is shared between the cores. OpenMP provides functionality for managing this shared memory. In contrast to message passing protocols such as MPI this is relatively easy to implement for existing code as the algorithmic structure can be left largely intact. It should be kept in mind that parallelisation always creates come (small) overhead, as data needs to be exchanged between the different cores.

[1] The next higher organisational unit in a computer cluster would be a *node*, which generally contains several processors. Parallelising NAME to run on a cluster would require the use of a message passing framework such as MPI to allow inter-node communication and data exchange.

[2] This assumes that the two programs do not access shared resources such as the hard drive.

1.1.1 Scaling

The main objective of parallelising NAME is to reduce the overall runtime of the code and it is therefore desirable to know how this time scales when the number of threads is increased. The simplest model for the time it takes to execute a parallelised code is given by Amdahl's law. This assumes that a fraction r of the runtime is spent in the parallelised section of the code. The self-relative speedup $S(n)$, i.e. the runtime on n cores compared to the runtime on 1 core is then given by

$$S(n) = \frac{T(n \text{ cores})}{T(1 \text{ core})} = \frac{1}{1 - r + r/n}. \tag{1.1.1}$$

Note that the speedup is limited by $S(n) < 1/(1-r)$ and the serial part of the code takes the same time to execute as the parallel part if $n = r/(1-r)$. Generally r and the speedup are a function of the problem size; if the number of particles in a NAME run is increased by a factor of 10 relatively more time is spent in the parallelised main particle loop. The right question to ask is often: "Given the runtime T_{serial} of a serial run and n available cores, by how much can the problem size be increased such that the resulting time of the parallelised run $T(n \text{ core})$ is not larger than T_{serial}." In section 3.2 we have demonstrated that, after making some additional (serial) optimisations to the code, the number of particles in a routine volcano run can be increased by an order of magnitude without increasing the total runtime if the model is run on 8 cores.

1.1.2 Hyperthreading

Some Intel processors now support the hyperthreading technology, which allows several threads to be executed on one physical core (which is the said to be split into different logical cores). This is described in more detail in section 4.1.3.

1.1.3 What has been parallelised?

Obviously there are limits on which parts of the code can be executed in parallel. So far the following sections of the code have been parallelised:

Particle Loop (LPPATs). This loop propagates the model particles over the period of time when they are independent, i.e. do not contribute to common output requirements. It generally parallelises very well with $r \approx 100\%$.

Particle Update Loop (CPRs). This loop over the particles calculates the contribution of individual particles to the output requirements at synchronisation times.

Chemistry Loop. This is a loop loop over the two-dimensional chemistry grid. Scalability of this loop degrades for larger numbers of processors. This is probably due to memory access problems which are discussed in detail in section 4.1.1.

Output Group Loop. Loop over the different groups of output requirements when these are written to a file.

Parallel MetRead. A separate thread can be launched for reading NWP MetData from disk while the other threads propagate the particles through the model atmosphere. At the moment only one parallel IO thread is allowed, so this section of the code does not scale in the traditional sense. It is currently not possible to use parallel IO together with MetOnDemand.

The speedup of the code depends significantly on the relative time spent in these sections of the code and can vary a lot between different setups, i.e. input files. In general, running NAME on four processors does not make the code four times as fast. In section 1.2.4 we describe how the timer routines can be used to measure the time spent in different sections of the code. Also note that if your run makes extensive use of resources shared between all threads, such as disk access, other processes run on the same machine will have an impact on the runtime.

Speedup can also depend on the load balance between different threads in a for loop. This is controlled by the scheduling strategy, at the moment this is hardwired for all parallelised loops, except for the chemistry loop where it is controlled by the environment variable `OMP_SCHEDULE`.

1.2 Using parallel NAME

1.2.1 Compilation

On the computer system of the atmospheric dispersion group the following compilers can be used to compile NAME in parallel mode:

- Intel Fortran v9.1. This is the default compiler on desktop computers. The code can also be compiled with Intel Fortran v10.0 which is loaded with the command

```
. prg_20070628
```

- Intel Fortran 11.0. This compiler is available on the servers. It is *not* the default compiler and it needs to be be invoked with the command

```
. /opt/intel/Compiler/11.0/083/bin/ifortvars.sh intel64
```

  The parallel code can *not* be compiled with the default compiler on the servers (Intel Fortran 9.0).

The code has also been successfully compiled on a Windows PC.

As for the serial code, the code preprocessor has to be compiled and stored in the `Executables_Linux` subdirectory prior to compilation of the NAME code. While the serial code can still be compiled with the compile script, the parallel code is compiled with a `Makefile`. Instead of compiling the entire source code in one go, the make utility will generate object code for each source file and link the `.o` files at the end of the compilation process. The compilation process is started by

```
make
```

Note that a complete recompilation is necessary when one of the compiler options has been changed or the code is compiled on a different system. In this case it is necessary to call a

```
make clean
```

first to remove old object and module files.

Additional options can be specified in the Makefile, default options can be overridden by specifying them as a command line parameter to make, e.g.

```
make COMPILERMODE=Debugging
```

will compile in the debugging mode instead of `Optimized` mode. The following options are available:

- `COMPILEROPTIONS={IntelLinux/IntelMac/IntelSun/Custom}` (default = `IntelLinux`):
  A set of compiler flags for the Intel compiler on different platforms are predefined in the `Makefile`. If `COMPILEROPTIONS` is set to `Custom`, these flags can be loaded from `Makefile_cflags`.
- `COMPILERMODE={Optimized,Debugging}` (default = `Optimized`):
  Compile in optimized or debugging mode.
- `VTUNE={true,false}` (default = `false`):
  Compile with support for Intel vTune performance analyzer.
- `MODEL={serial,parallel}` (default = `parallel`):
  For compatibility reasons, the code can also be compiled in serial mode by setting `MODEL` to `serial`.
- `USETIMERS={true,false}` (default = `true`):
  Compile code with support for timer module?
- `USEGRIBEX={true,false}` (default = `false`):
  The code can be compiled with support for the GRIBEX library by setting this to true. This requires a copy of the GRIBEX library which is accessible both at compile time and runtime.

If compiled in parallel mode (default), the executable will be called `nameiii_par.exe`, depending on the setup, other postfixes such as `Debug` (compiled in debugging mode), `Gribex` (compiled with support for the GRIBEX library) or `_64bit` (compiled on a 64bit platform) might be added to the name of the executable. For example, if the parallel code has been compiled in debugging mode on a 64bit system, the executable will be called `nameiiiDebug_64bit_par.exe`.

1.2.2 Running NAME

On the dispersion group's servers, the default library paths are set up to point to the Intel Fortran 9.0 libraries, which does not include the OpenMP library. This library, however, is needed at runtime as it is linked dynamically. The correct paths can be set by using

```
export LD_LIBRARY_PATH=/opt/intel/Compiler/11.0/083/lib/intel64: \
${LD_LIBRARY_PATH}
```

or by loading the Intel Fortran 11.0 environment, as described in section 1.2.1.

1.2.3 Input parameters

The behaviour of the parallel code is controlled by a set of parameters in the input file. These are summarised in a block with the label `OpenMP Options:`. The following options are available:

- `Use OpenMP?` [default = `No`]: If this is set to `Yes` the following options are used, otherwise they will be ignored and set to their default values.
- `Threads` [default = `1`]: Number of threads in all parallelised for loops. The number of threads for individual loops can be specified with the following four options.
- `Particle Threads` [default = `1`]: Number of threads in the particle loop

- `Particle Update Threads` [default = `1`]: Number of threads in the particle update loop
- `Chemistry Threads` [default = `1`]: Number of threads in the chemistry loop
- `Output Group Threads` [default = `1`]: Number of threads in the output group loop
- `Parallel MetRead` [default = `No`]: Read NWP MetData with separate IO thread. If this is option is set to `Yes` and `Parallel MetProcess` is not specified, the latter is automatically set to `Yes`, i.e. the data is read *and* processed by the IO thread.
- `Parallel MetProcess` [default = `No`]: Process NWP MetData with separate IO thread. If this is set to `Yes` and `Parallel MetRead` is `No` (or not specified), then the latter is automatically set to `Yes`.

The scheduling strategy of the chemistry loop is controlled by the environment variable `OMP_SCHEDULE`, which can be set before the start of a NAME run.

1.2.4 Timers

The code now contains a timer module to measure the time spent in individual sections of the code. To use this module, the code needs to be compiled with the `-DUseTimers` flag. Timer information is printed out to the file `<InputFileStem>Timing.txt` where `<InputFileStem>` is the stem of the input file of the NAME run. The detail at which this timing information is printed out is controlled by the `Timer Options:` block in the input file:

- `Use Timers?` [default = `No`]: Print out timer information
- `Summary Only?` [default = `Yes`]: Only print out summary information at the end of the run.

1.2.5 Reproducibility - a word of warning

Due to the limited precision, floating point arithmetics is usually non-associative, i.e.

$$\begin{aligned} a + (b + c) &\neq (a + b) + c \\ a * (b * c) &\neq (a * b) * c. \end{aligned} \tag{1.2.1}$$

In particular this implies that the value of a sum depends on the order in which the individual values are added. In the parallel version of the code the sum is split between the different threads and the order in which it is updated can not be predicted. It is thus to be expected that the results between different parallel runs or between a parallel run and a serial run differ, even if the code has been compiled in `Debugging` mode.

Chapter 2

Parallelisation strategy and implementation

This more technical chapter describes the parallelisation strategy and its implementation. Note that the development version 5.4a has been used for work in this chapter, and some of the details such as how to control the number of threads in parallel sections of the code is obsolete, please refer to chapter 1 for the current version.

2.1 Setup

All results in this report were obtained on the computing infrastructure of the Atmospheric Dispersion Group which is briefly outlined in the following.

2.1.1 Processors

Linux

The code was run both on 32bit desktop machines and on the 64bit servers of the Atmospheric Dispersion Group. Most testruns were performed on the server `els034` with two Intel Xeon X5470 (Harpertown) quadcore processors [1] (8 cores in total), with a clockspeed of 3.33 GHz and 24.6 MB main memory. Some testruns were carried out on `name-e` with two Intel Xeon X5450 quadcore processors (8 cores in total), with a clockspeed of 3.00 GHz and 16.4 MB main memory. Both processors use a two-level cache, the primary (L1) cache is local to each core and consists of a 32 kB instruction cache and a 32 kB data cache. The L2 cache is shared and has a size of 12 MB.

Results with the VTune performance analyzer were obtained on the desktop machine `eld497` with an Intel Xeon E5520 (Gainestown) quadcore processor (4 cores in total) which has a three level cache structure. Each of the four cores has a primary (L1) cache, split into a 32 kB instruction and 32 kB data cache, and a midlevel (L2) cache of 256 kB. The L3 cache is shared between the cores and has a size of 8 MB[1]. This processor supports hyperthreading so that effectively 8 threads can run in parallel, but for all runs in this report this was disabled. Hyperthreading technology is also supported for the new servers `els045` and `els047`.

The difference in processor layout and cache structures raises some questions as to whether the results can be compared between the two architectures. As we only used a trial version of VTune installed on `eld497` further investigation of this issue has to be postponed until the tool is available on both systems. It might also be worth using other freely available performance analysis tools such as the PAPI library [2].

Windows

We also compiled and ran the parallel code on the Atmospheric Dispersion Group's windows laptop with an Intel Core 2 Duo T7500 processor. Each core has a private 32 kB (write-back) data and separate 32 kB instruction cache. The L2 cache is shared between the cores and has a size of 4 MB.

2.1.2 Compilers

On the 32bit desktop machine we used Intel FORTRAN version 9.1.039 to compile the code, a more recent version (10.0.023) is available as well. The standard compiler on the servers is Intel FORTRAN 9.0 which can not be used to compile the parallel code due to a missing dynamic library (this problem can be fixed by adjusting the library path `LD_LIBRARY_PATH` at runtime). However, Intel FORTRAN 11.0 can be loaded on request and we used this version to compile the parallel code. The code can be compiled in *Debugging* and *Optimized* mode (for details, see the Makefile), all results in this report were obtained with maximal optimisation.

2.2 Initial status

The parallelisation of the NAME code was started in 2008 by Rupert Ford and Graham Riley of Manchester Informatics Ltd. at the School of Computer Science, University of Manchester. OPENMP was used to parallelise some of the computationally most expensive loops in the code.

In addition the code was adjusted such that NWP MetData can be read from disk by a dedicated IO thread while at the same time a set of worker threads use already available flow information to propagate particles through the atmosphere.

The first stage of this work is based on version 5.3 of NAME and the results have been documented in a set of reports [3, 4, 5]. The main changes that were implemented at this stage are summarised in the following.

2.2.1 Parallelisation of particle- and chemistry loops

At the start of the project `gprof` was used to profile the serial code and the results were confirmed with a timer library written by Rupert Ford and Graham Riley[1]. It was found that significant amount of the runtime was spend in three loops which can be parallelised relatively easily:

"Particle update" loop. The contribution of individual particles to the different output requirements is calculated in the subroutine `RunToSyncTimeOrMetFlowUpdateOrEndOfCase()` in the file `Case.F90`. The loop over particles was parallelised with an `!$OMP DO` directive. Due to a bug in versions 9 and 10 of the Intel FORTRAN compiler the loop could not be parallelised directly, instead it had to be moved to the separate subroutine `CalcParticleResultsSub()` (sometimes abbreviated as CPRs) which is called from within an `!$OMP PARALLEL` region (see [3] for details).

"Particle" loop. The particles are propagated forward in time in the subroutine `LoopParticlesPuffsAndTimeSteps()` (file `Case.F90`). The main loop in this subroutine was parallelised with an `!$OMP DO` directive; the loop body was moved into the separate subroutine `LoopParticlesPuffsAndTimeStepsSub()` (abbreviated LPPATs). In the original code particles and puffs are treated in the same part of the code by looping over a particle type index which is 1 for puffs and 2 for particles. So far, only the particle case has been parallelised and this loop was split up into two separate loops, the puff loop is still run in serial within an `!$OMP SINGLE` region.

[1] Based on ideas in this library we wrote a module with similar functionality (see file `Timers.F90`) which has been added to the NAME code base.

To ensure reproducibility of results when running on multiple cores the random number generator was adjusted such that a separate set of random numbers is generated for each position in the particle stack (random seed = `Fixed (Parallel)`). In addition a reproducibility flag can be set which ensures that when particles are removed from the run, the order in which free spaces become available on the particle stack is independent on the number of cores the code is run on.

As different threads access the shared variables `Results` and `DispState` which are passed as `Intent(InOut)` parameters to subroutines in `Particle.F90` and `Case.F90`, updates to these variables were protected by `!$OMP ATOMIC` and `!$OMP CRITICAL` statements in these subroutines.

"Chemistry" loop. Chemistry calculations are carried out in a Eulerian framework which requires looping over the boxes of a (three dimensional) chemistry grid. The outermost loop (over the x coordinate) in the subroutine `ChemistryUpdate()` (file `Chemistry.F90`) was parallelised with an `!$OMP DO` directive.

The number of threads which are used in each parallel region is controlled by the environment variables `PARTUPDATETHREADS`, `PARTTHREADS` and `CHEMTHREADS`. Any of these values which is not set is overwritten by `OMP_NUM_THREADS`. The scheduling strategy of the particle loop is hardwired to `"static,16"` as this is the optimal value reported in [5] (this differs from the results in [3], where `"dynamic,16"` was reported as optimal.). For the other parallel loops it is specified at runtime via the environment variable `OMP_SCHEDULE`. See section 4.1.2 for further investigations of the optimal scheduling strategy.

Results

The code was tested for two benchmarks: A simple particle run (`Example_SingleSite.txt` and a short version `Example_SingleSite_Quick.txt`), which does not include any chemistry processing and does not read MetData from disk, and a chemistry run (`air_quality_test.txt` and `air_quality_test_short.txt`).

It was verified that, if the correct random number scheme is used and the reproducibility flag is set, the results do not depend on the number of threads used (Differences in output fields are typically in the least significant figure and much smaller than statistical fluctuations). Achieving bit reproducibility is hard because many floating point addition is not associative and the parallel code will usually perform sums in different orders, depending on the number of threads used.

The self relative speedup S, which is defined to be the time taken by the code when run on one core compared to the runtime on n cores,

$$S(n) = \frac{t(1)}{t(n)} \tag{2.2.1}$$

was measured for up to 8 threads. It turns out that both the "particle" and the "particle update" loop scale reasonably well but the speedup of the chemistry loop reaches a plateau for around 6 threads and then descreases slightly [4]. The significant amount of time spent in this part of the code in some cases (in one of the benchmarks investigated in section 3.1 around 50% of the time is spent in the chemistry loop) justifies further investigation of this issue, see section 4.1.1. As shown there, both the serial and parallel performance of this section of the code can be optimized further.

2.2.2 Parallel IO

In some cases reading NWP MetData from disk can make up a significant proportion of the runtime, for an example see the EMARC run discussed in section 2.3.4. This process has been parallelised by creating a separate IO thread which prefetches MetData for the next timestep while the worker threads propagate the particles using already available flow information. The most significant changes were made to the file `NWPMet.F90` and two new files were added to the NAME code base: `IOThreadModule.F90` is used to control the behaviour of the IO thread whereas the implementation of the synchronisation mechanism between the different threads can be found in the file `OpenMPModule.F90`.

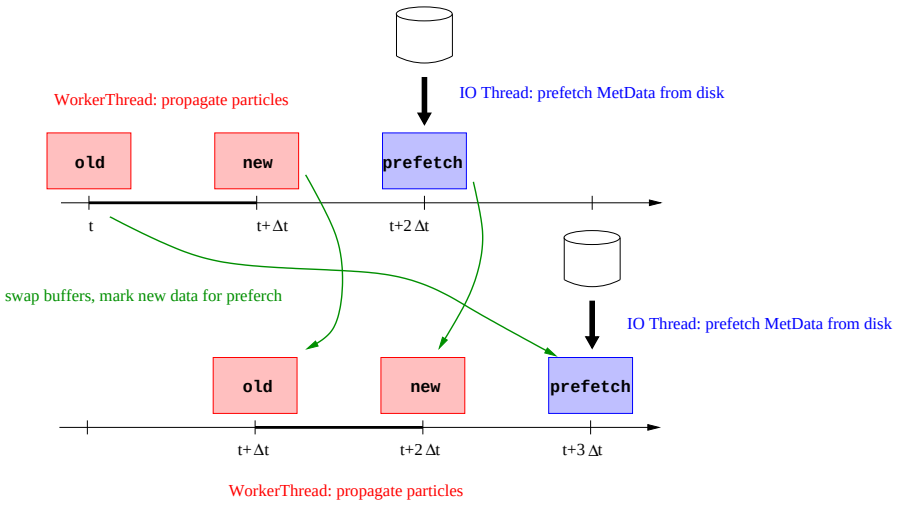


Figure 2.1: Parallel reading of NWP MetData

After its creation the IO thread repeatedly loops over the different MetModules and updates the flow information by reading new data from disk if one of the modules has been marked for prefetching by the worker threads. The process is repeated until the flag `IOLookAheadComplete` is set to `.true.` by the worker threads at the end of the run. Prefetch requests are launched by the worker threads in the subroutine `UpdateNWPMet()`. Three buffers are used for storing MetData: data for the previous and next timestep is stored in the `old` and `new` buffers (at the beginning of the run these buffers are read from file by the worker threads). Both buffers are needed as flow fields are interpolated between these two times. Data for the following is loaded into a `prefetch` buffer by the IO thread. Once this prefetched data is available and the worker thread is ready to propagate the particles over the next time interval, the buffers are swapped around according to `prefetch` $\mapsto$ `new`, `new` $\mapsto$ `old`, `old` $\mapsto$ `prefetch` and a new prefetch request is launched. The process is illustrated in Figs. 2.1 and 2.2. Synchronisation between the IO- and worker threads is realised with OpenMP locks. It turns out that the original solution to this problem works but violates the OpenMP standard, a new mechanism is proposed in section 2.3.2.

2.3 Further improvements

Following the stage 2 intermediate report [4], we integrated the changes described in the previous section into the current development version (v5.4a) of the NAME code. One of the new features in version 5.4a is the support "for Met-on-demand", which allows to update only those flow modules which are required at a particular timestep. As discussed in section 2.3.3 this requires further adaptations to the parallel code. We also implemented a number of additional improvements:

- Fixed some bugs in the original implementation (section 2.3.1)
- Implemented a new synchronisation mechanism to comply with the OpenMP standard (section 2.3.2)
- Adapted the parallel code to support Met-on-Demand (section 2.3.3)
- Implemented parallel processing of MetData with the IO thread (section 2.3.4)

All these changes are now part of the development version in the `apdg` directory and will be included in the frozen version 5.5. On Linux systems the serial code can still be compiled as usual with the compile scripts `LinuxIntelRelease` and `LinuxIntelDebug`. In addition we created a Makefile in the `/Code_NameIII` subdirectory which can be used to compile the parallel version of the code, see this file and the release notes for further details.

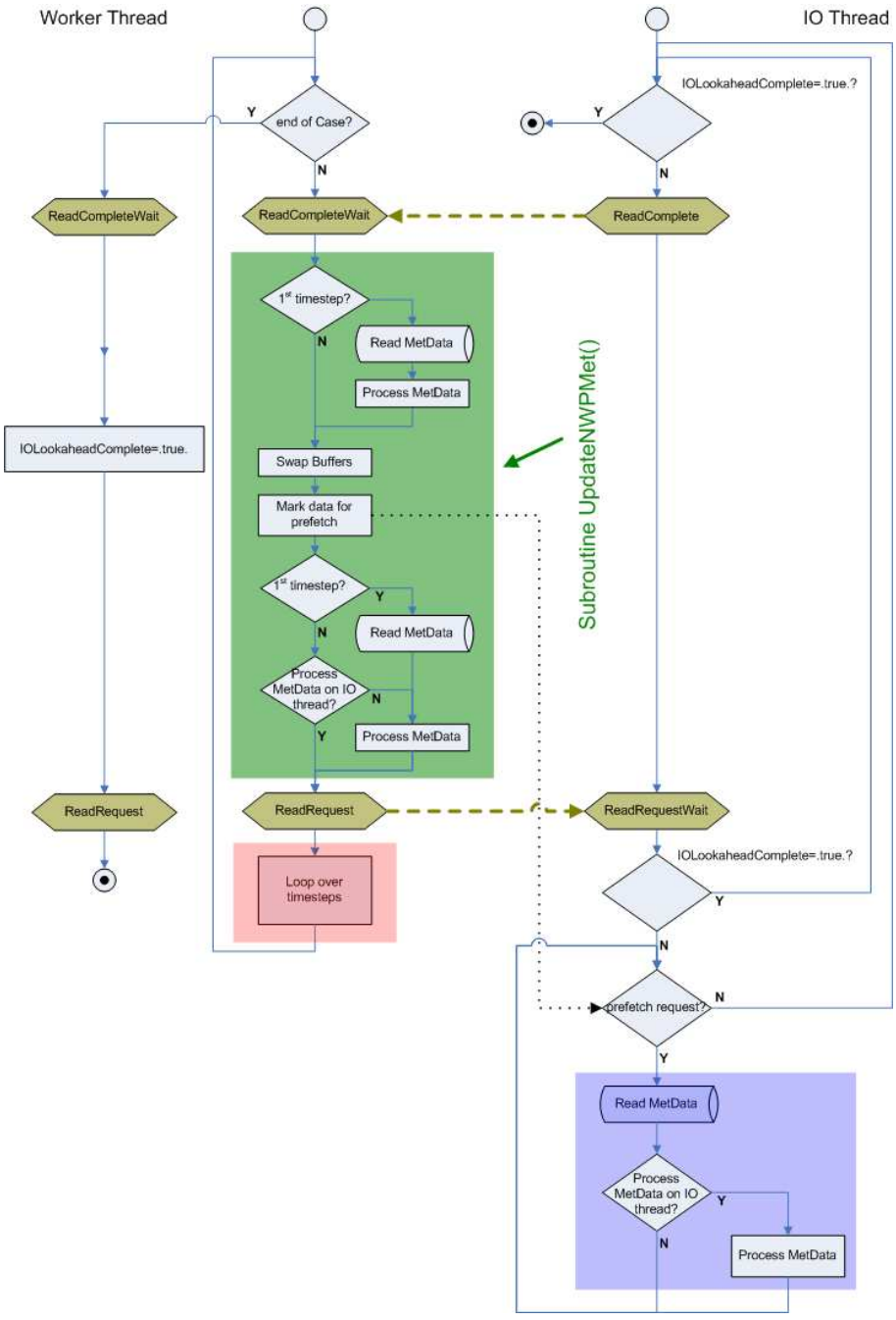


Figure 2.2: Parallel reading of MetData and synchronisation between the worker and IO thread (see section 2.2.2), based on a diagram by Rupert Ford and Graham Riley [5]. Actions of the worker thread are shown on the left, the IO thread on the right. MetData is marked for prefetch in the subroutine `UpdateNWPMet()` (green) and read from disk by the IOThread in the subroutine `ReadNWPMet()` (blue). At the same time particles are propagated through the flow fields by the worker thread (red).

2.3.1 Bugs fixed

We fixed two bugs in the implementation of parallel IO which became apparent during test runs of the parallel code:

The first is related to how the worker threads determines when to stop marking data for prefetching. It does this by checking whether the time of the MetData to be prefetched is smaller than `IOMaxLookahead`. However, in `Main_NameIII.F90` the value of `IOMaxLookahead` is set to `DispState%sLastDispTime` which is the time when the particles stop propagating. If `DispState%sLastDispTime` coincides with a time when MetData is available, this is correct and the run completes normally. However, if `DispState%sLastDispTime` lies between two MetData times, then the next MetData after `DispState%sLastDispTime` is needed for interpolation but will not be read as the time of this MetData, i.e. the prefetch time, is larger than `DispState%sLastDispTime=IOMaxLookahead`.

We fixed these problems by rounding `IOMaxLookahead` to the next MetData time in `UpdateNWPMet`, i.e. we replaced the if statement in `UpdateNWPMet()`

```
If (SMetTime3<=IOMaxLookahead) Then
  ...
End If
```

by

```
If (SMetTime3<=Time2ShortTime(Round(ShortTime2Time(IOMaxLookahead),&
    NWPMet%MetDefn%T0,NWPMet%MetDefn%Dt,up = .true.))) Then
  ...
End If
```

Additionally we modified the code such that `NWPMet%PrefetchTime` is set to `SMetTime3` by the IO thread as soon as it has read MetData from disk (before the variable was set by the worker thread when it marked data for prefetch). This ensures that `NWPMet%PrefetchTime` is only set *after* the data has been read, so that it will trigger an error in case there is a problem with reading the file. Prior to this change the code sometimes accidentially read a previous MetData file and did not realise that the time was wrong, see above.

The second bug, which caused the code to crash with a strange runtime error, is caused by the wrong size of the third dimension of the `Dummy`, `DummyU` and `DummyV` buffers in the subroutine `ReadNWPMet()`. We changed this from 2 to 3 for parallelIO.

Some smaller bugs we fixed are:

- The second index of the field `ValidAttribs` in the type `NWPMet_` needs to run from 1...3 instead of 1...2.
- In the subroutine `UpdateNWPMet()` the overall validity attribute depends on the validity attribute of the "old" and "new" buffer, not 1 and 2.

2.3.2 New synchronisation mechanism

When using parallel IO for reading MetData from disk, synchronisation between the worker and IO thread is achieved with the help of the four subroutines `ReadComplete()`, `ReadCompleteWait()`, `ReadRequest()` and `ReadRequestWait()`[2]. The worker threads can only leave the subroutine `ReadCompleteWait()` after

[2]The full names as they appear in the code are `LookaheadFileReadRequest()` etc.; in this text we drop the first part of the names for brevity.

the IO thread has called `ReadComplete()`, whereas the IO thread can only complete `ReadRequestWait()` after the worker thread has called `ReadRequest()`. Internally, this is realised with a set of three locks[3] in the file `OpenMPModule.F90`. A thread can lock (or set) a lock only if it is currently unlocked. The previous realisation assumed that a thread can release a lock, even if it has been locked by a different thread. Although this appears to work on our system and with the Intel FORTRAN compilers we used, it is not compatible with the OPENMP standard. The new implementation, which is summarised in Fig. 2.3, uses three locks: In the beginning the IO thread sets locks number 1 and 3 whereas the worker threads set lock number 2. In `ReadCompleteWait()` the worker thread tries to set lock number 1 but has to wait for this lock to be released in by the IO thread in `ReadComplete()`. After it has swapped the data buffers and launched a new prefetch request, the worker thread then releases lock number 2 in `ReadRequest()` which allows the IO thread to set this lock in `ReadRequestWait()`. This cycle repeats until `LookaheadFileReadComplete` is set to `.true.`.

This mechanism simplifies the implementation as the number of the lock a thread operates on can be cycled periodically, i.e. the index of the lock that is "passed around" between the threads changes according to $1 \to 2 \to 3 \to 1 \to 2 \to \ldots$ etc.

2.3.3 Met on Demand

Usually NWP MetData is read from disk at regular intervals during the run, for example in a 10 hour run starting at 1am, global MetData, which is available at three hourly intervals, will be read for 0z, 3z, 6z and 12z. However, flow information might not be needed at certain times, for example if the domain that is covered by a particular flow module does not contain any particles that need to be propagated. This is typically the case in runs with a "chopped up globe" [6]: Instead of storing MetData for the entire globe, the data is split up into 14 regions which are read from disk separately[4]. Particles released over Europe will only require flow information over North America later in the run, when they have propagated around the globe. In this case, the flow modules for the different segments can be loaded on demand, i.e. a specific flow module will only be updated once a particle enters its region of validity. If Met-on-demand is enabled, particles can request flow information by calling the subroutine `GetAttrib()`, which in turn calls `WhichFlow()`. This subroutine then identifies the correct flow module and updates flow information by calling `UpdateFlow()`, see Fig. 2.4. Note that `UpdateFlow()` subroutine can also be called indirectly via `UpdateMetsFlows()` and `UpdateFlows()` from `RunToRestartDumpOrSuspendOrEndOfCase()`. Without Met-on-demand this is the normal way of updating flow modules at regular intervals.

Updating flow information on demand creates a problem in the parallel implementation: If two particles, processed by different threads, enter the same region, both will request an update of the same flow module and the data will be read from disk twice. This creates conflicts when updating shared variables and wastes resources for reading data from disk. To prevent this, we created a set on locks, one for each Flow- and Met- module.

In our implementation calls to the subroutine `UpdateFlow()` within `WhichFlow()` are protected with locks, i.e.

```
Call SetFlowLock(i,j)
Call UpdateFlow(...)
Call UnsetFlowLock(i,j)
```

Here the subroutines `SetFlowLock(i,j)` (`UnsetFlowLock(i,j)`) set (release) the lock for the flow module which is uniquely defined by the pair of indices i, j. If a thread tries to update a flow module which

[3] Another possibility would be to use a shared logical variable and make one thread wait until the value of this variable has changed. The advantage of using locks is that this uses less system resources: after a certain time a thread waiting for a lock will enter a dormant state where it does not use any resources.

[4] It should be noted that splitting up the global data in 14 segments has no negative impact on the runtime: On the contrary, it was found that for a typical RIMNET benchmark the runtime is reduced by 15% by "chopping up" the globe even if all segments are loaded. The reasons for this are not clear at the moment.

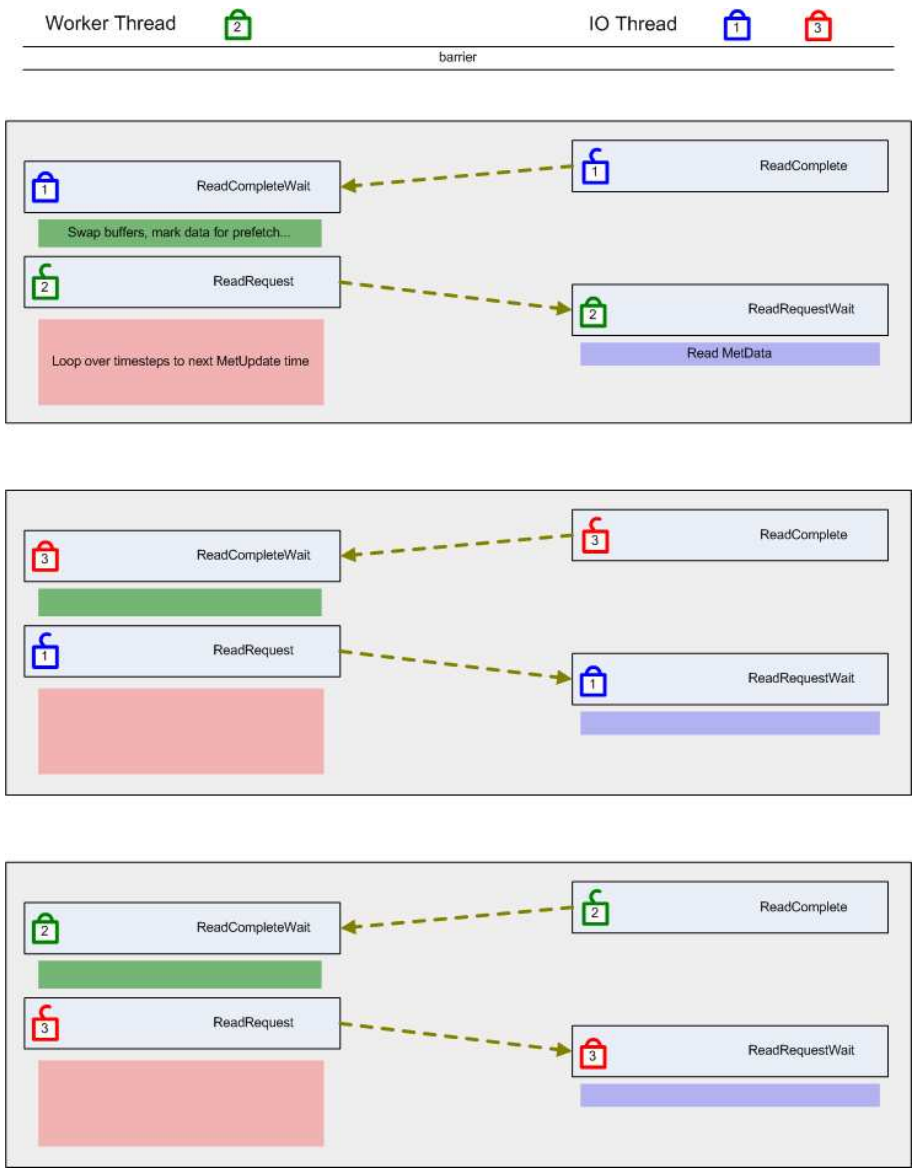


Figure 2.3: Details of the synchronisation mechanism with locks described in section 2.3.2.

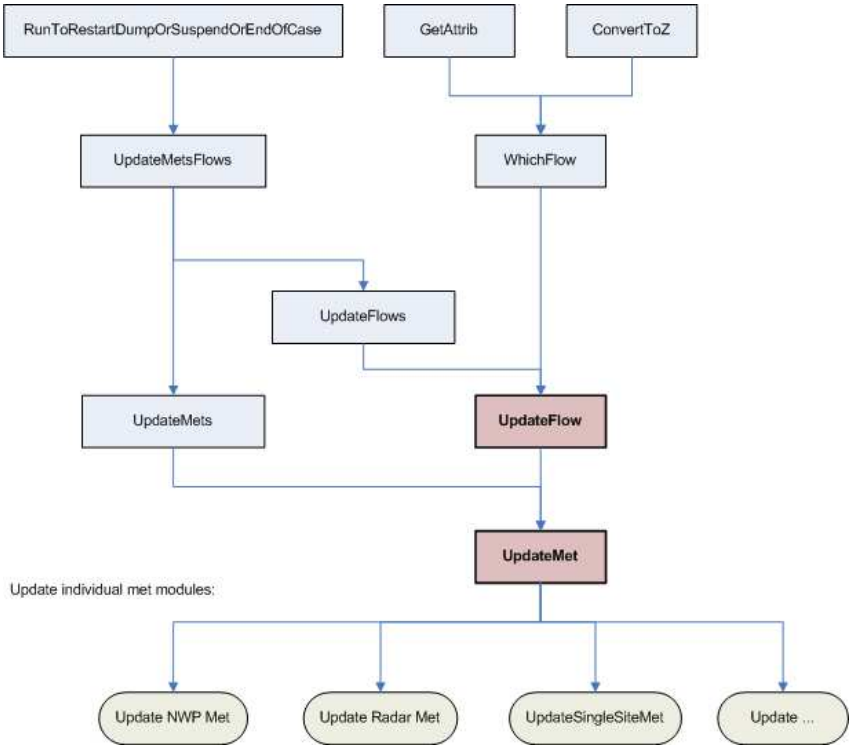


Figure 2.4: Calls to the subroutines `UpdateFlow()` and `UpdateMet()` when using Met-on-Demand.

has been locked by a different thread it will wait at `SetFlowLock()` until the other has completed `UpdateFlow()` and released the lock with `UnsetFlowLock()`. Then executing `UpdateFlow()` he second thread will then realise that the data has already been read from disk and will not read it again.

It is necessary to protect updates of Met modules separately as updates to the same Met module can be requested by two different flow modules: one can use it to obtain flow information (`WhichFlow()` is called from `GetAttrib()`) and the other can use it for a vertical coordinate transformation (`WhichFlow()` is called from `ConvertToZ()`). The same lockingg mechanism as in `WhichFlow()` is used in the subroutine `UpdateFlow()` to protect calls to `UpdateMet()`:

```
Call SetMetLock(i,j)
Call UpdateMet(...)
Call UnsetMetLock(i,j)
```

A further change to the file `Unit.F90` was necessary to prevent two threads using the same unit id number when opening MetData files: In `GetNewUnit()` and `CloseUnit()` access to the (`Intent(InOut)`) variable `Units` is now protected with an `!$OMP CRITICAL` section.

Note that currently Met on Demand can *not* be used together with parallel IO, the code will stop with an error if both schemes are used together.

2.3.4 Parallel processing of MetData

After reading NWP MetData from disk in the subroutine `ReadNWPMet()` the raw fields are processed to calculate secondary quantities in `ProcessNWPMet()`. Fig. 2.6 shows an activity plot for a typical EMARC run (input file is given in Fig. B.1), in this example two worker threads were used. The figure

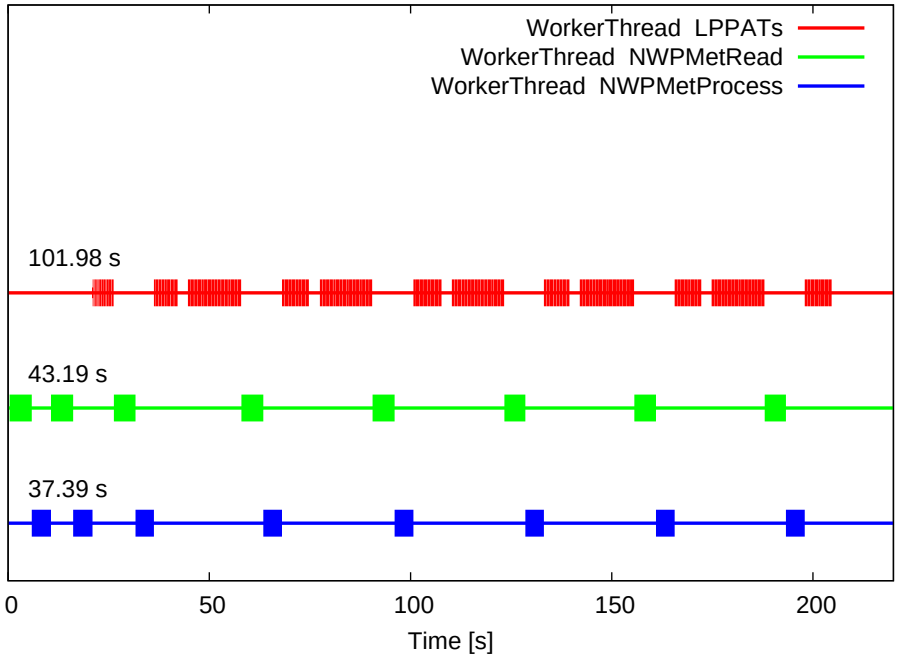


Figure 2.5: Activity diagram for the EMARC testcase in section 2.3.4. The top row shows the "particle" loop whereas the two bottom rows show the time spent in the subroutines that read and process MetData. Two worker threads are used and parallel IO is disabled.

shows the time spent in the parallel particle loop (LPPATs) and in the subroutines `ReadNWPMet()` and `ProcessNWPMet()` and illustrates that processing of MetData can take as much time as the reading from disk itself.

We modified the code to perform the MetData processing with the IO thread. This required adding two new parameters `OldIdx` and `NewIdx` to the subroutine `ProcessNWPMet()` (and the subroutines called from there) as this subroutine can now be called by the IO thread and operates on combinations of all three MetData buffers: `old`, `new` and `prefetch`. When running with more than one thread this created a segmentation fault which we traced back to updating of some of the fields with the FORTRAN wildcard of the form `Field1(:,:,...) =` SomeFunctionOf(`Field2(:,:,...)`). The reason for this is not clear but it seems to be related to a similar problem and possibly bug in the intel compiler reported by Helen Webster: she found that in some runs arrays were overwritten without warning. One possible solution is to compile the code with the option `-heap-arrays` which forces the compiler to puts automatic arrays and arrays created for temporary computations on the heap instead of the stack. This is now a default compiler option in the Makefile. The influence of this compiler flag on the runtime has not been tested yet.

The behaviour of the IO thread is controlled by the environment variable `USEPARALLELIO`. If this is set to `READONLY`, data will be read with the IO thread but processed with the worker thread. If it is set to `TRUE`, the data will be read *and* processed by the IO thread; in all other cases no IO thread is created and the data is read and processed by the worker threads. Figs. 2.6 and 2.7 show an activity plot with a parallel IO thread.

Note that the first two MetData files are always read by the worker threads as they are needed to propagate the particles over the first timestep.

In this particular example, using the IO thread to read MetData reduces the total runtime to 82% compared to the original setup where all data is read and processed by the worker threads. If the IO thread also processes the data, the runtime is reduced even further to around 72%.

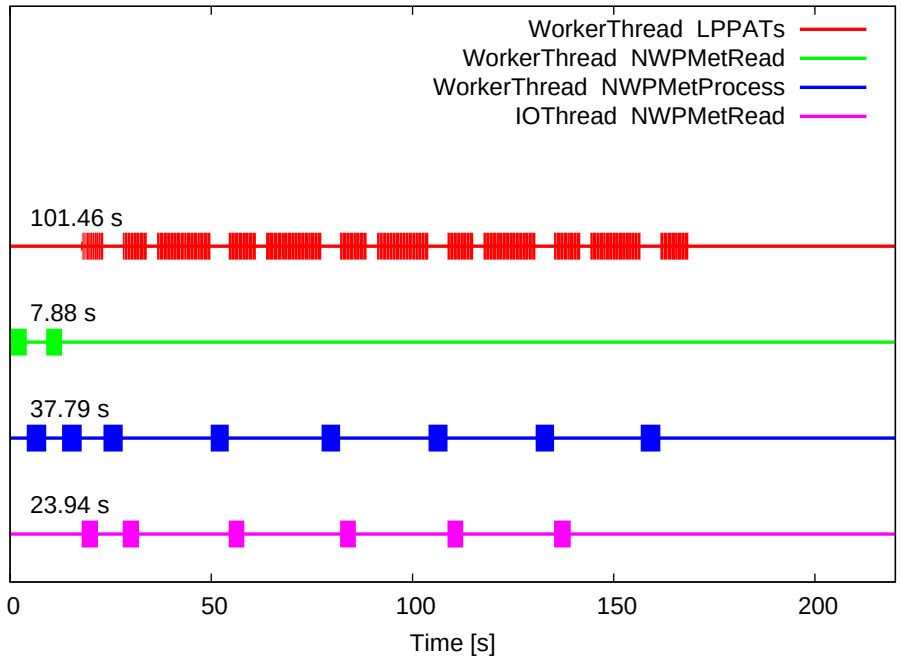


Figure 2.6: Activity diagram for the EMARC testcase in section 2.3.4. The second and third row show the time the worker threads spent on reading and processing MetData, the bottom row shows the time spent on reading data with the IO thread. Two worker threads are used and MetData is read with the IO thread but processed by the worker thread.

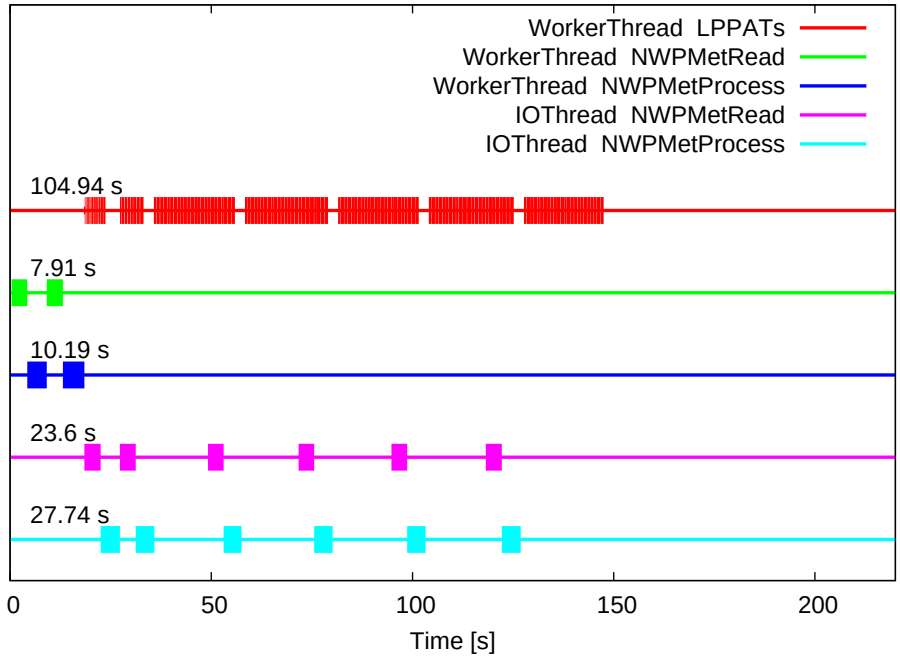


Figure 2.7: Activity diagram for the EMARC testcase in section 2.3.4. The second and third row show the time the worker threads spent on reading and processing MetData whereas the two bottom rows show the corresponding activity by the IO thread. Two worker threads are used and MetData is read and processed with the IO thread.

2.3.5 Skipping of MetData

Usually NWP MetData is available at regular intervals and the flow fields are interpolated linearly in time between the data files. At the moment the Unified Model stores the atmospheric conditions every three hours up to a forecast time of 72z. After this time it generates 6 hourly forecasts up to 144z. NAME can only deal with one MetUpdate timescale and as a temporary fix three-hourly data files were created after 72z by linear interpolation with an external program. This is not very efficient as (1) resources are needed for generating and storing the interpolated files and (2) NAME spends time on reading redundant information from disk. This problem becomes more urgent with the recent upgrade of the Unified Model to PS23 and the resulting increase in data volume. Ultimately one might want to introduce an additional timescale in the MetDefinition file to deal with MetData at different time intervals but as a temporary fix we implemented a mechanism for skipping missing MetData files. This means that whenever NAME can not find a MetData file it will print out a warning message, increase the internal temporal interpolation interval (which is used to construct the interpolation coefficients in the subroutine `GetTCoeffs()`) and try to read data at the next MetUpdate time. This is repeated a maximal `MaxSkip` number of times (at the moment this parameter is hardwired to `MaxSkip=1`, i.e. no more than one missing file can be skipped). For UM forecast data, where the NWP Met update interval is set to 3 hours, but only 6 hourly files are available after 72z, this implies that NAME will skip the missing file at 75z and try to read the next file at 78z. If it cannot find this file it will stop with an error message, as before. Internally this is realised by adding two parameters to the subroutine `ReadNWPMet()`, which reads MetData from disk. If `AllowSkip` is set to `.true.` and NAME can not find the requested MetData file, the parameter `Skip` will be set to `.true.` and a warning message will be printed out. If, on the other hand, `AllowSkip` is false an the requested file does not exist, the code will print out an error message and set the `error` flag to `.true.`. Figs. 2.8 and 2.9 show flowcharts of the subroutines

If parallel IO is enabled, skipping of MetData is not allowed and a missing file will always cause the code to stop with an error.

Tests

We verified that skipping an interpolated MetData file has no impact on the final results (within rounding errors). We first extracted an atmospheric variable and checked that its value the same over the interpolation time. We then examined the output particle concentrations in the test run and again found that they agree within tolerance in both runs.

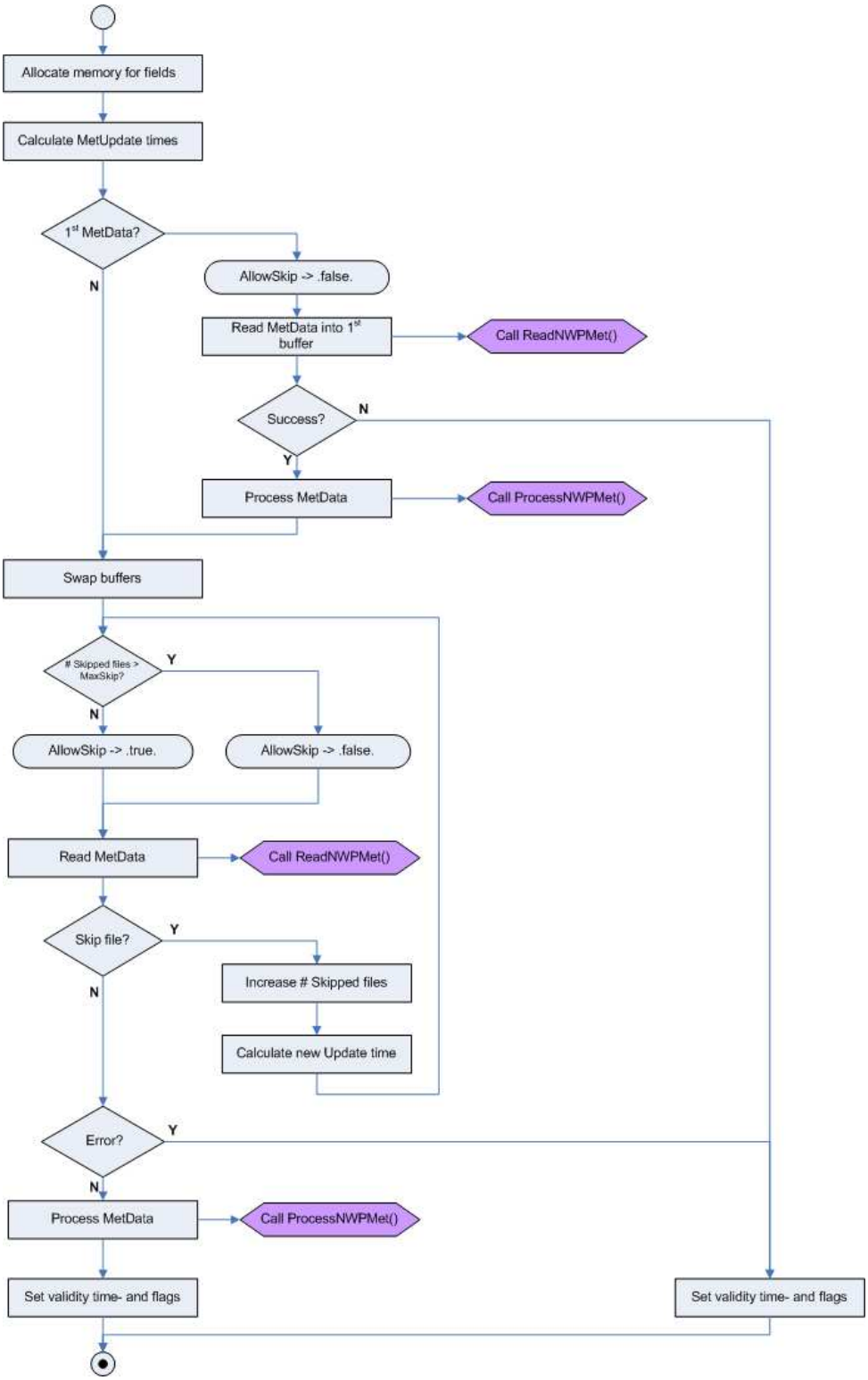


Figure 2.8: Subroutine `UpdateNWPMet()` and skipping of MetData files as described in section 2.3.5.

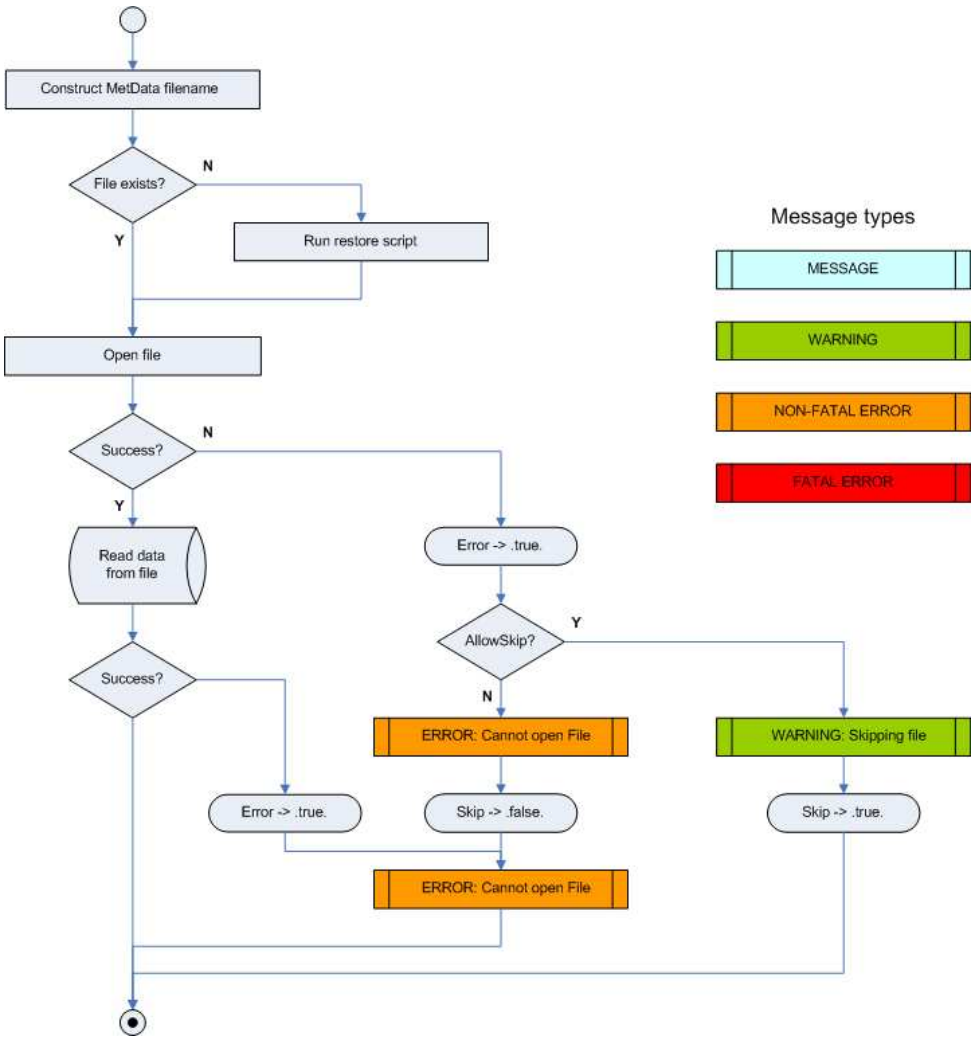


Figure 2.9: Subroutine `ReadNWPMet()` and skipping of MetData files as described in section 2.3.5.

Bibliography

[1] Intel Xeon specifications: `http://ark.intel.com/Product.aspx?id=35430` (X5470 "Harpertown"), `http://ark.intel.com/Product.aspx?id=34446` (X5450 "Harpertown"), `http://ark.intel.com/Product.aspx?id=40200` (E5520 "Gainestown"), `http://ark.intel.com/Product.aspx?id=29761` (Core 2 Duo T7500)

[2] Performance Application Programming Interface `http://icl.cs.utk.edu/papi/`

[3] Graham Riley and Rupert Ford: *NameIII Parallelisation Study for the MetOffice - Stage 1 Intermediate report, DRAFT v0.1* 28 October 2008

[4] Graham Riley and Rupert Ford: *NameIII Parallelisation Study for the MetOffice - Stage 2 Intermediate report, DRAFT v0.1* 18 March 2009

[5] Graham Riley and Rupert Ford: *NameIII Parallelisation Study for the MetOffice - Stage 2 Final report, v1.0* 4 March 2010

[6] Lucy S. Davis *Global cut out domains* Met Office Internal report md17_1_1 (27/01/09)

Chapter 3

Case studies

3.1 Timing of air quality benchmark

The parallel code will be used in the operational air quality system. In the following we present timing results for two benchmarks and discuss the performance improvement from parallelisation of the code.

3.1.1 Setup

The performance and scaling of the code was investigated for two realistic testcases which represent typical runs of the air quality (AQ) system:

- a **coarse** benchmark which in its original configuration runs for approximately 8 hours (on 1 core) and
- a high resolution (**hires**) benchmark which takes approximately 12 hours to complete.

The high resolution benchmark contains substantial chemistry processing, which takes up nearly half of the runtime (see Tab. 3.1). Both benchmarks were run on the 64bit server `els034`. Parallel IO is switched off for all testruns. The input files can be found in appendix B.

3.1.2 Results

We first identified parts of the code that contribute significantly to the overall runtime. Timers were inserted to monitor the activity between two successive MetData updates, which are taken to be relatively late in the run (updates 46/47 in the coarse- and 22/23 in the hires-run) to allow the particle concentrations to reach equilibrium. The results are shown in Figs. 3.1 and 3.2, both runs use one thread each.

The coarse testrun is dominated by the "particle" and "particle update" loops, whereas a significant amount of time in the high resolution testrun is spent in the "chemistry" loop. In the latter case, a non-vanishing fraction of the time is spent in the `Release()` subroutine, which has not been parallelised yet. Currently parallelisation of this subroutine is not a high priority as in most other configurations it only takes up a minor fraction of the total runtime. In both cases reading (and processing) of MetData only takes up a neglible fraction of the runtime and it is not sensible to use a dedicated IO thread for this. Since the majority of the runtime is spent in parallel sections of the code we expect a significant performance gain when running on more than one core.

The sync time in the hires run was 10 m, with a (global) MetData update every three hours (i.e. 18 sync times in the interval shown in Fig. 3.2). The irregular pattern of the "Release" and "CPRS" intervals

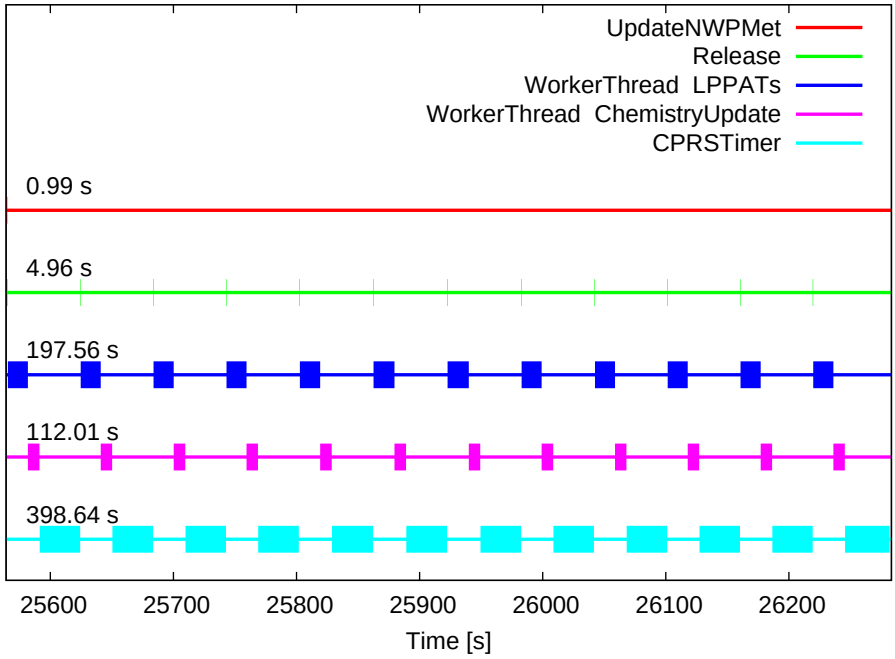


Figure 3.1: Interval timing, coarse AQ benchmark. We show the time spent on loading NWP MetData from disk (`UpdateNWPMet()`), the `Release()` subroutine, the parallelised "particle" and "particle update" loops (`LPPATs` and `CPRS`) and the chemistry subroutine `ChemistryUpdate()`.

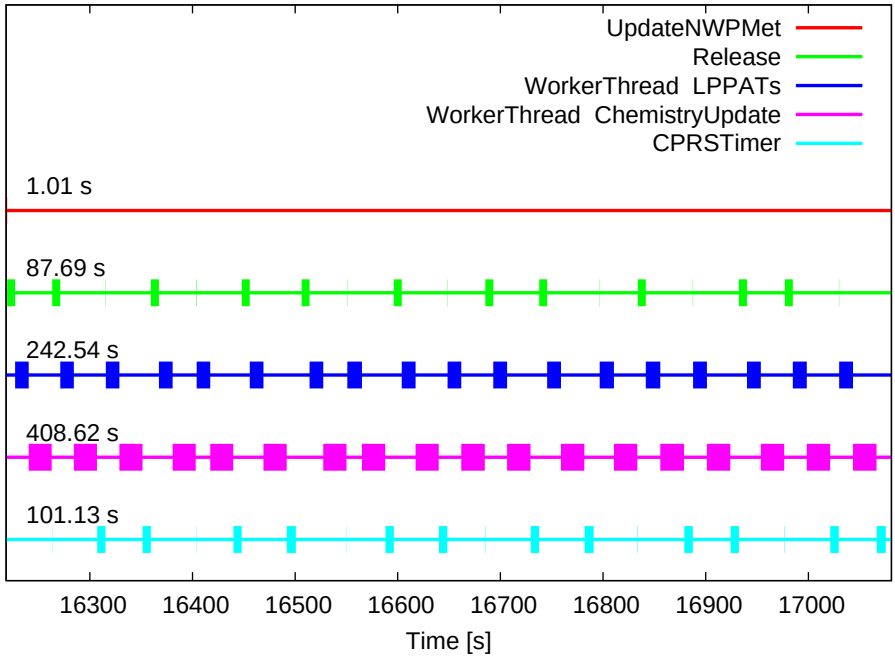


Figure 3.2: Interval timing, hires AQ benchmark. We show the time spent on loading NWP MetData from disk (`UpdateNWPMet()`), the `Release()` subroutine, the parallelised "particle" and "particle update" loops (`LPPATs` and `CPRS`) and the chemistry subroutine `ChemistryUpdate()`.

benchmark	LPPATs	Chemistry	CPRs	total
coarse	29%	17%	52%	98%
hires	28%	47%	11%	86%

Table 3.1: Time spent in different parallelised sections of the code, data taken for a 1 thread parallel run of the two AQ benchmarks.

# threads	total	LPPATs	Chemistry	CPRs
serial	486 m	—	—	—
1	462 m	132 m	78 m	241 m
2	266 m	75 m	53 m	128 m
4	159 m	42 m	41 m	65 m
6	118 m	33 m	31 m	44 m

Table 3.2: Timing results, coarse AQ benchmark

can be explained by the other timescales in the problem: concentration averages are required every 15 m and the source strength is 0.001particles/ s (corresponding to the release of 1 particle every 16 m 40 s. Both particle release and output calculation can only be performed at sync times.

Timing results are shown in Tabs. 3.2 and 3.3, the (self-relative) speedup is plotted as a function of the number of threads in Figs. 3.3 and 3.4 (due to a (now fixed) bug in the timer module, individual speedup data is not available for more than 2 threads). Note in particular that running the parallel code with one thread is faster than the serial code. The reasons for this are not clear at the moment, but probably the `-openmp` compiler flag will lead to additional optimisations.

These results confirm the earlier observation of the poor scaling of the chemistry loop. The bad overall scaling of the hires benchmark can be explained by the relatively larger time spent in the subroutine `ChemistryUpdate()`.

3.1.3 Short air quality testcase

As the air quality runs take several hours to complete we also use a simplified testcase. The inputfile `air_quality_test_short.txt` (see Fig. B.2) is the same as the one was used by Rupert Ford and Graham Riley. This testcase runs for several minutes and is therefore more suitable for quick testing. At some point the timing results reported in the following should, however, be backed up by more extensive runs with a full air quality testcase.

3.1.4 Serial/parallel parts of `ChemistryUpdate()`

For historic reasons the time spent on chemistry processing is measured by timing the entire subroutine `ChemistryUpdate()`. One possible explanation for the bad scaling of the chemistry code would be a

# threads	total	LPPATs	Chemistry	CPRs
serial	727 m	—	—	—
1	682 m	192 m	323 m	77 m
2	445 m	111 m	200 m	45 m
4	331 m	—	—	—
6	295 m	—	—	—

Table 3.3: Timing results, hires AQ benchmark

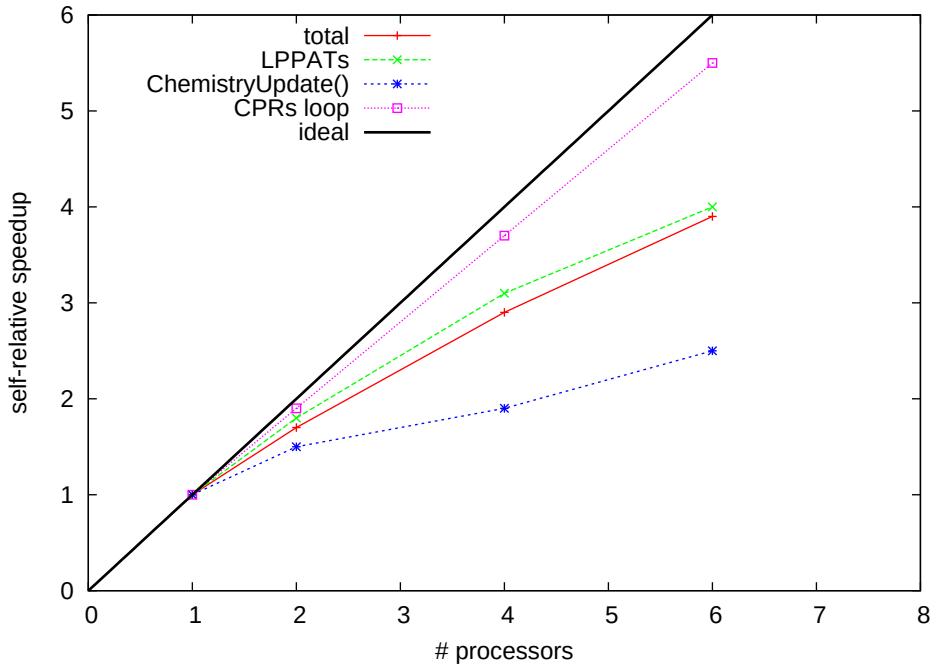


Figure 3.3: Self relative speedup, coarse AQ benchmark. We show the total speedup, the speedup of the "particle" and "particle update" loops and the speedup of the subroutine `ChemistryUpdate()`.

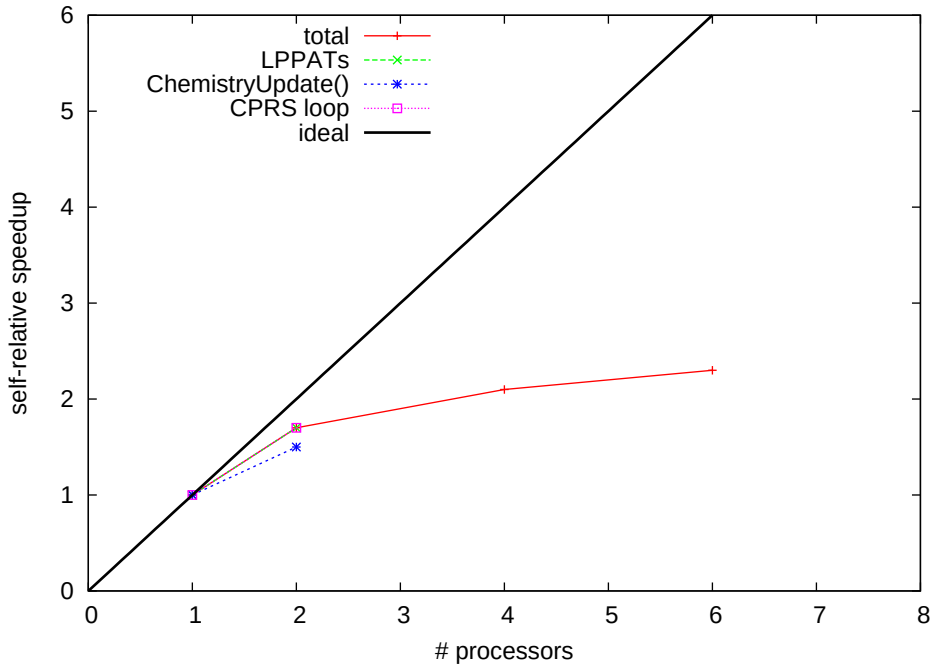


Figure 3.4: Self relative speedup, hires AQ benchmark. We show the total speedup, the speedup of the "particle" and "particle update" loops and the speedup of the subroutine `ChemistryUpdate()`.

# threads	`ChemistryUpdate()`	chemistry loop
1	119.48 s	111.00 s
2	83.08 s	74.66 s
3	72.01 s	63.60 s
4	69.42 s	61.02 s
5	69.42 s	61.35 s
6	72.13 s	63.72 s
8	79.49 s	71.07 s

Table 3.4: Timing of the entire subroutine `ChemistryUpdate()` and the parallelised chemistry loop only

significant proportion of serial code in this subroutine. In this section we exclude this possibility by further timer measurements.

If a fraction r of the subroutine is parallelised, Ahmdal's law predicts that the (ideal) speedup on n cores will be

$$S_n = \left(1 - r + \frac{r}{n}\right)^{-1} < \frac{1}{1-r}, \tag{3.1.1}$$

the transition from linear to constant behaviour occurs where the serial and parallel code take the same amount of time to execute

$$n_{\text{crossover}} \sim \frac{r}{1-r}. \tag{3.1.2}$$

We added a further timer to measure the time spent in the parallel section of `ChemistryUpdate()`. The results are shown in Tab. 3.4, the self-relative speedup is plotted in Fig. 3.5. The testcase used was `air_quality_test_short`.

We conclude that the time spent in the serial part of the code is small compared to the time spent in the parallel chemistry loop; the parallel loop on its own shows poor scaling, for this particular testcase it is even worse than for the air quality benchmark in Fig. 3.3. In particular the slope of the curve is slightly negative for $n > 4$ (this has to be compared to the positive slope in Figs. 3.3 and 3.4 and might be due the relatively short runtime in the `air_quality_test_short.txt` testcase). As around 93% of the time is spent in the parallel sction of the code, we would expect $n_{\text{crossover}} \sim 13$ from Ahmdal's law, which clearly does not agree with our observations and suggests that other problems cause the poor scaling.

3.1.5 Chemistry gridsize and loop order

We tested the influence of the relative size of the x- and y dimensions of the chemistry grid on the time spent in the chemistry loop by alternatively dividing the same physical domain into 100×50 or 50×100 boxes. Again, the `air_quality_test_short` testcase was used, the scheduling strategy was set to `"dynamic,16"` to allow for good load balancing. In addition we interchanged the order of the x and y loops in the chemistry loop for both grid layouts. Originally the loop over the x coordinate is the outermost iteration which is parallelised with an `!$OMP DO` directive. The two cases we tested are:

- x-loop is outer loop (case x in Tab. 3.5, original setup)

```
!$OMP PARALLEL DO
Do iX = 1, HGrid%nX
  Do iY = 1, HGrid%nY
  ...
  End Do
End Do
```

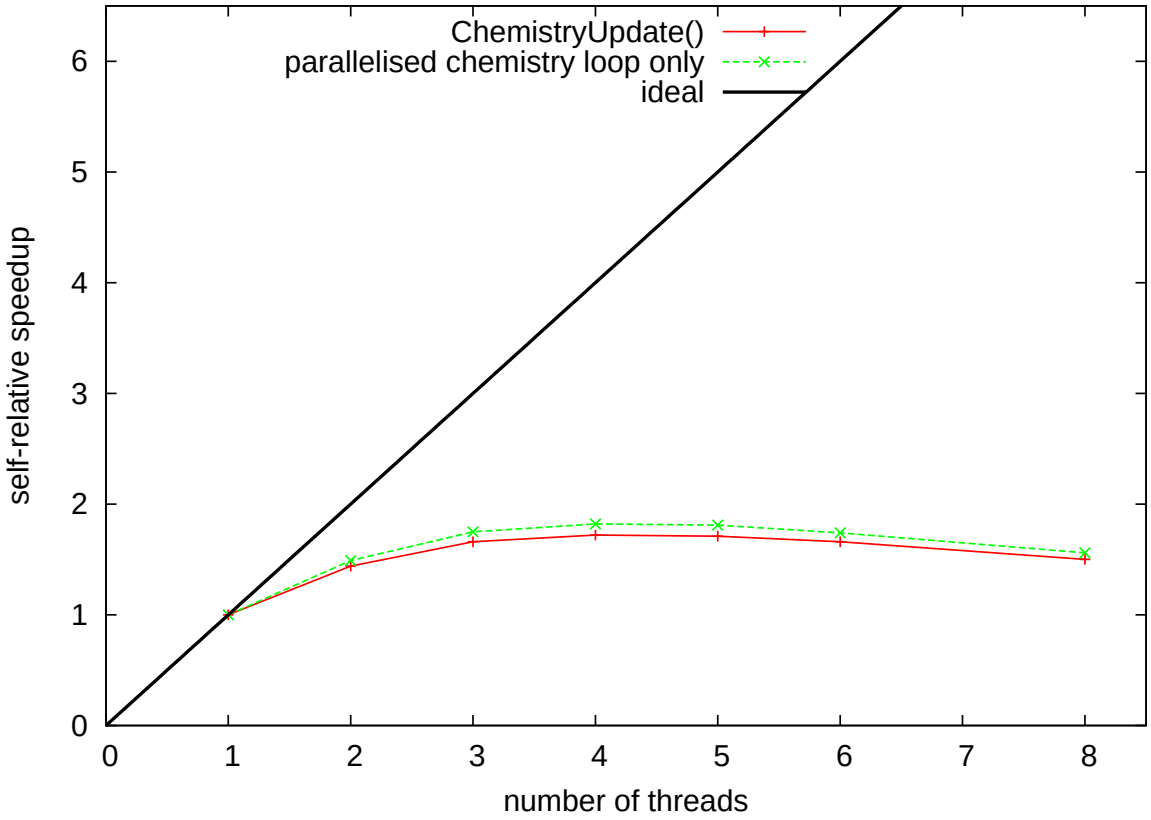


Figure 3.5: Self relative speedup of the entire subroutine `ChemistryUpdate()` and the parallised chemistry loop only

grid size ($x \times y$)	case x	case y
100×50	75.73 s	72.15 s
50×100	75.51 s	73.72 s

Table 3.5: Time spent in the subroutine `ChemistryUpdate()` for different chemistry grid sizes and loop orders

- y-loop is outer loop (case y in Tab. 3.5)

```
!$OMP PARALLEL DO
Do iY = 1, HGrid%nY
  Do iX = 1, HGrid%nX
  ...
  End Do
End Do
```

In both cases the code was run on 4 cores. The results are collected in Tab. 3.5. As can be seen from these numbers the dependence on the relative size of the x- and y direction of the chemistry grid is very weak, which suggests that the load is well balanced in both cases, irrespective of the grid layout. On the other hand, interchanging the loop order reduces the runtime by a few percent.

This is plausible when looking at the different arrays in the structure `Type(ChemistryState_)`. The spatial part of these arrays is usually laid out in the form `(x,y,z)`. In FORTRAN the first index of an array runs fastest and memory access can be optimised by looping over this index in the innermost loop.

For the moment we will leave the order of the x- and y loop as it is but in section 4.1.1 we will change the layout of some of the arrays in `Type(ChemistryState_)`. As shown there this leads to a significant speedup of the chemistry loop at the order of 10%.

3.2 Timing of the operational volcano code

In section 3.2.2 we present a detailled timing analysis of the code after parallelising the "particle" and "particle update" loops and investigate the scaling of the parallel code for different particle release rates. We verify that the changes made to the code have no impact on the final results. It turns out that for volcanic ash runs a significant fraction of the runtime is spent in the subroutine which writes the output results to disk. In subsection 3.2.2 we suggest some simple code changes which improve the efficiency of this subroutine dramatically. Finally, in subsection 3.2.2, we investigate the scalability of the code on the new server `els035` which will be used for operational volcanic ash runs. Scaling of the parallelised particle loop is very good for up to 8 cores, and the scalability improves for higher particle release rates. With all improvements described in this document the runtime of the parallel version of the code on 8 cores, with a tenfold increase in the release rate, is now comparable to the original runtime of the serial code.

In section 3.2.3 we study the impact of these improvements on the ensemble runs on ECMWF data and present a timing analysis for a simplified ensemble run. On four cores the new parallelised code is more than twice as fast as the original serial code.

A final verification study is presented in section 3.2.4 where we confirm that the parallelised code produces the same results as the previous serial version 5.4.3.

3.2.1 Setup

As a first step, we created a copy of the `Name 5.4.3` code, which is currently used for volcanic ash runs, and parallelised the "particle" (LPPATs) and "particle update" (CPRs) loops as described in more detail in section 2.2.1. In contrast to version 5.4a, reading of MetData has not been parallelised yet, and this would require significantly larger code changes. The new code is stored as version 5.4.4 in the `NameIIILibrary` directory.

3.2.2 Deterministic runs

The scripts `EYJAFJALLAJOKULL_variablesource_routine.scr` and `EYJAFJALLAJOKULL_expensive.scr` were adapted such that the first only generates the 5 day red, black & gray plots and the latter does not produce any graphical output at all; the time spent in IDL subroutines is not taken into account in the following timing analysis. In the routine script, the number of particles emitted per hour was set both to 1562 (the *standard* setup, currently used for operational forecasts) and to 6000 (the *MoreParticles* setup). The emission rate in the latter case is identical to that in the *expensive* run. In the future, increasing the emission rate by one order of magnitude will increase the statistics which is necessary when averaging over thinner verical layers. For all runs described in this report, the start time was fixed to 06z on 08/05/2010 when the volcano entered a phase of increased activity. After a spinup time of 6 days, forecasts are produced from 12z on 08/05/2010 to 00z on 14/05/2010.

Verification

We first checked that the parallel code produces the same results as the operational code. The code was run in parallel mode on four cores on the server `els045`[1] and the results were compared to those from a serial run of the v5.4.3 code on the same machine. Both codes were compiled in `Optimized` mode, the executable for version 5.4.3 was copied from the `NameIIILibrary` directory in `apdg`, this version had been compiled with version 9.0 of the Intel Fortran compiler. In contrast, version 11.0 of the compiler was used to compile the parallel code. In Figs. 3.6 to 3.11 we compare the results from the parallel run to those of the serial run. The random seed was set to `Fixed (Parallel)`. Different levels of ash concentration are plotted for the flight levels 000-200 at 12z from 08/05/2010 to 13/05/2010. We can not see any significant qualitative differences between the two runs

We also compared the results of a serial run to a parallel run on one core. Even in this case, the resulting fields do not agree bitwise, this is most probably due to the fact that the code was compiled in `Optimized` mode in both cases. To verify this, we compiled both the serial and parallel code in `Debugging` mode and ran both codes over the spin-up time until 00z on 08/05/2010. For this time period we compared the file `DebugInfo_Particle1_C1.txt` which contains information about one particle trajectory. In contrast to the `Optimized` runs, we find bitwise agreement between the results from the serial and parallel version.

A more detailled verification study of the final version of the parallel code is presented in section 3.2.4.

Initial Timing

Total runtime The total time for the serial NAME run in the standard setup is 1 h 26 m, whereas the parallel run takes 1 h on 4 cores and is slightly slower than the serial run on one core (1 h 31 m). The standard run with an emission rate of 6000 particles per hour (*MoreParticles* setup) takes 1 h 35 m on four cores and 3 h 36 m on one core, i.e. increasing the release rate by a factor of four and running on four cores will result in about the same runtime as using the original release rate and the serial code. The *expensive* run completes within 1 h 49 m on four cores and 5 h on one core, equivalent to a speedup factor of 2.7. The expensive run on four cores is only around 20% slower than the original serial routine run. We conclude that this configuration benefits most from the parallelisation we have implemented so

[1] This server is currently used for operational volcanish ash runs.

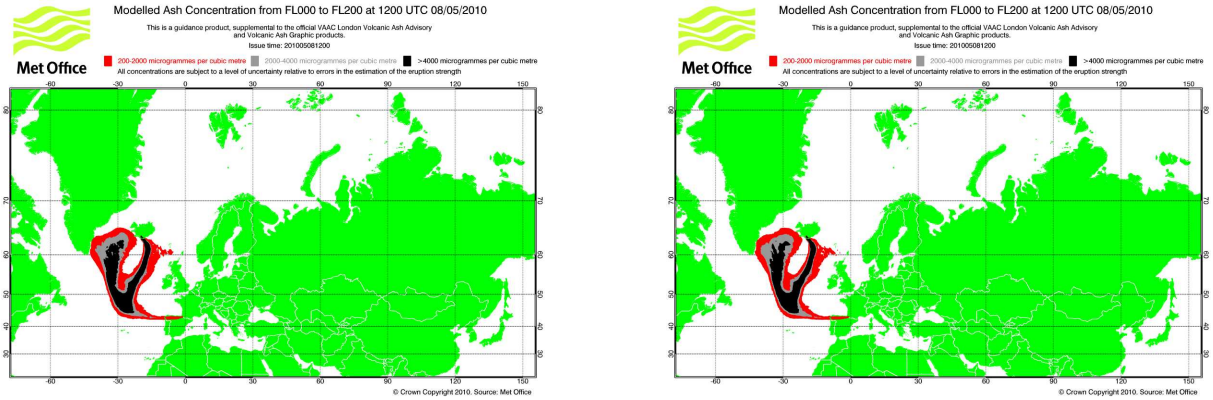


Figure 3.6: FL000-200, 12:00 on 08/05/2010. Serial (left) and parallel (right)

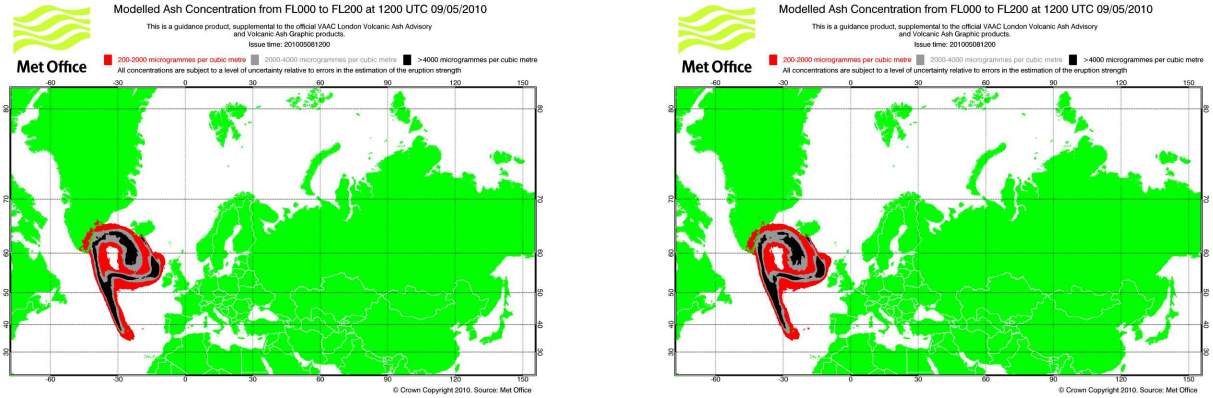


Figure 3.7: FL000-200, 12:00 on 09/05/2010. Serial (left) and parallel (right)

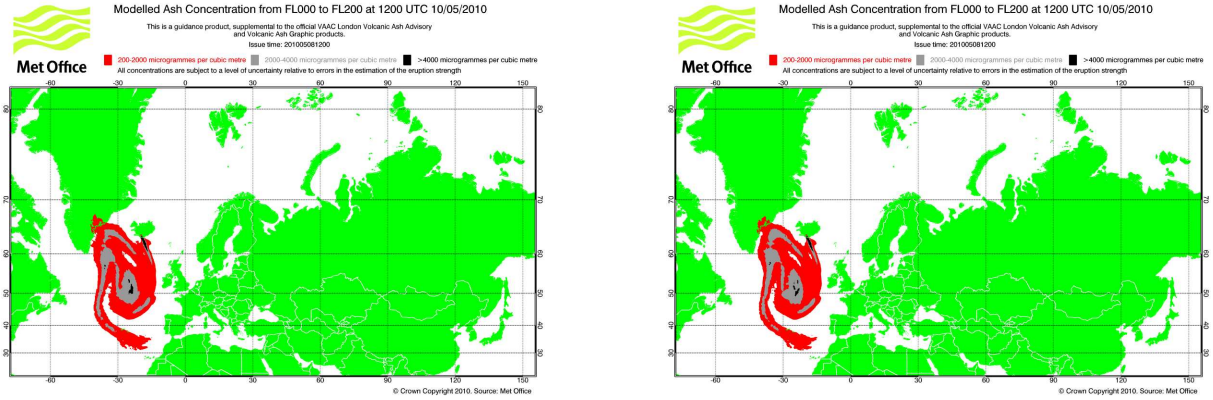


Figure 3.8: FL000-200, 12:00 on 10/05/2010. Serial (left) and parallel (right)

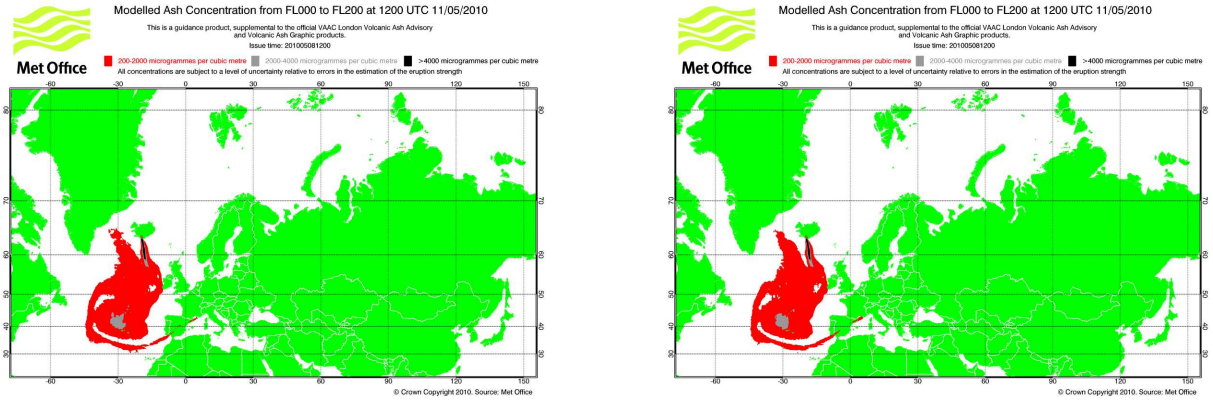


Figure 3.9: FL000-200, 12:00 on 11/05/2010. Serial (left) and parallel (right)

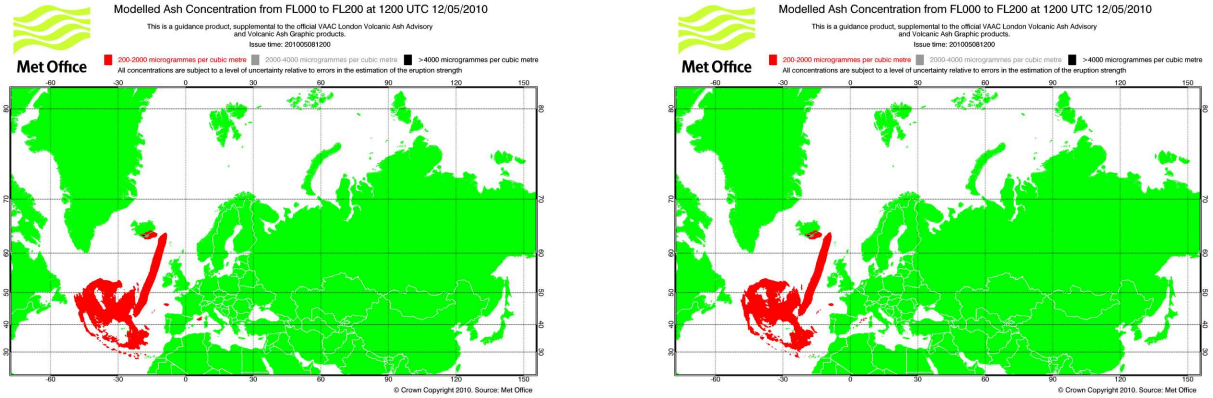


Figure 3.10: FL000-200, 12:00 on 12/05/2010. Serial (left) and parallel (right)

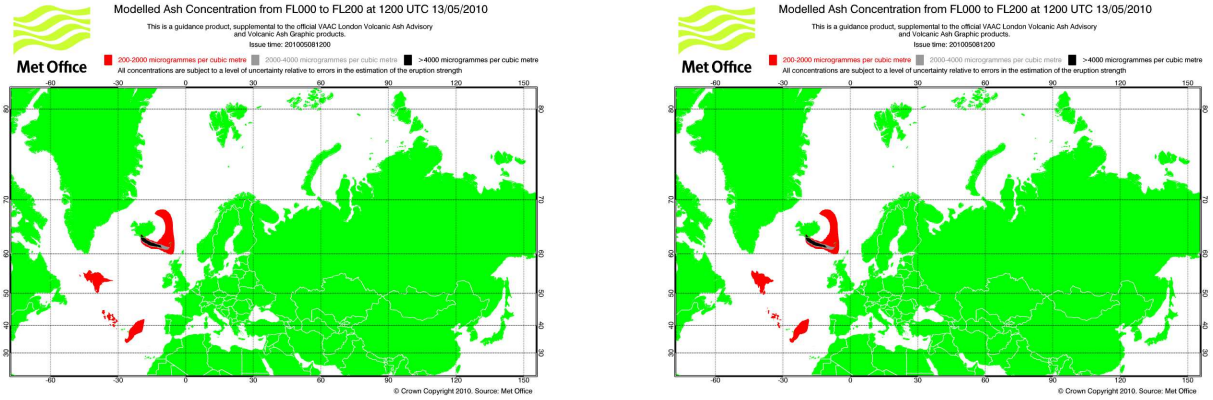


Figure 3.11: FL000-200, 12:00 on 13/05/2010. Serial (left) and parallel (right)

far. As discussed below, this is because a larger part of the runtime is spent in the parallel section of the code.

Detailled timing Using the timer library, we measure the time spent in the following sections of the code:

- Parallelised particle loop (`LPPATs`)
- Reading of MetData from disk (`NWPMetRead`)
- Processing of MetData (`NWPMetProcess`)
- Release of particles, i.e. calls to the subroutine `Release()` in `RunToSyncTimeOrMetFlowUpdateOrEndOfCase()` (Release)
- Processing and outputting of results, i.e. calls to the subroutine `ProcessAndOutputResults()` in `RunToSyncTimeOrMetFlowUpdateOrEndOfCase()` (`PAORs`)

Note that, as the code is run with Met-on-Demand, MetData can be read while executing the main particle loop or from within the `Release()` subroutine, which is why the timers `LPPATs` and `Release` can overlap with `NWPMetRead/Process`. We have, however

$$t(\texttt{LPPATs}) + t(\texttt{Release}) + t(\texttt{PAOR}) \quad \leq \quad t_{\text{total}} \tag{3.2.1}$$

and indeed the three terms on the left hand side of (3.2.1) account for most of the runtime. In Tab. 3.6 we collect timing information for the routine run both in the *standard* and *MoreParticles* setup, the corresponding results for the expensive run can be found in Tab. 3.7. We find that in the routine runs on 4 cores most of the time is spent in the subroutine processing the output, followed by the particle loop and the `Release` subroutine, whereas for the *expensive* run the particle loop dominates the runtime, which explaines the good overall scaling. We remind the reader that only the particle loop has been parallelised (but any calls to `NWPMetRead/Process` are still executed serially when using Met-on-Demand, which has an impact on the scalability). The scaling of this part of the code improves when the particle number is increased, i.e. the self relative speedup increases from 3.0 to 3.5. An interval plot, which shows which subroutines are active at a given time during code execution, is shown in Fig. 3.12. As can be seen from this figure, the run can be split into two parts: During the spinup time, when no output is required, the run is dominated by calls to the `Release` subroutine, the particle loop and `NWPMetRead`. Figs. 3.13 and 3.14 show timing results from the beginning and the end of the spinup period. As more particles are released into the atmosphere, the relative importance of the particle loop increases. At the beginning `NWPMetRead` overlaps with `Release`, but not with the particle loop. This is because MetData for Iceland and the north Atlantic is loaded on demand in the `Release` subroutine, but none of the particles have travelled far enough to request MetData for different parts of the globe. Actually, it is very likely that most of the time attributed to `Release` is spent in `NWPMetRead` called from within this subroutine.

In contrast, during the forecast period most of the runtime is spent in the subroutine that processes and ouputs the results, see Fig. 3.15. Towards the end of the forecast period, as particles get removed from the atmosphere, the amount of time spent in the output subroutines is reduced but still significant, see Fig. 3.16.

Scaling

Based on the scaling information of the particle loop, we try to give rough estimates for the runtime on different numbers of cores. To account for the imperfect scaling of the particle loop, we assume that it can be split into a serial part and a parallel part, where the latter takes up a fraction r of the time,

$$t(\texttt{LPPATs}, n \text{ cores}) \quad = \quad \left(\frac{r}{n} + (1-r)\right) t(\texttt{LPPATs}), \tag{3.2.2}$$

section	*standard*			*MoreParicles*		
	1 thread	4 threads	speedup	1 thread	4 threads	speedup
`LPPATs`	2869 s	963 s	3.0	10289 s	2953 s	3.5
`NWPMetRead`	207 s	208 s		224 s	218 s	
`NWPMetProcess`	222 s	206 s		244 s	232 s	
`Release`	159 s	158 s		162 s	161 s	
`PAORs`	2332 s	2350 s		2336 s	2338 s	
Total runtime	91 m	60 m	1.5	216 m	95 m	2.2

Table 3.6: Timing results for different sections of the code for routine runs. The code was run both with the *standard* emission rate of 1562 particles/hour and in the *MoreParticles* setup with 6000 particles/hour.

section	*expensive*		
	1 thread	4 threads	speedup
`LPPATs`	16177 s	4517 s	3.6
`NWPMetRead`	232 s	231 s	
`NWPMetProcess`	254 s	241 s	
`Release`	165 s	165 s	
`PAORs`	869 s	875 s	
Total runtime	299 m	109 m	2.7

Table 3.7: Timing results for different sections of the code for expensive runs.

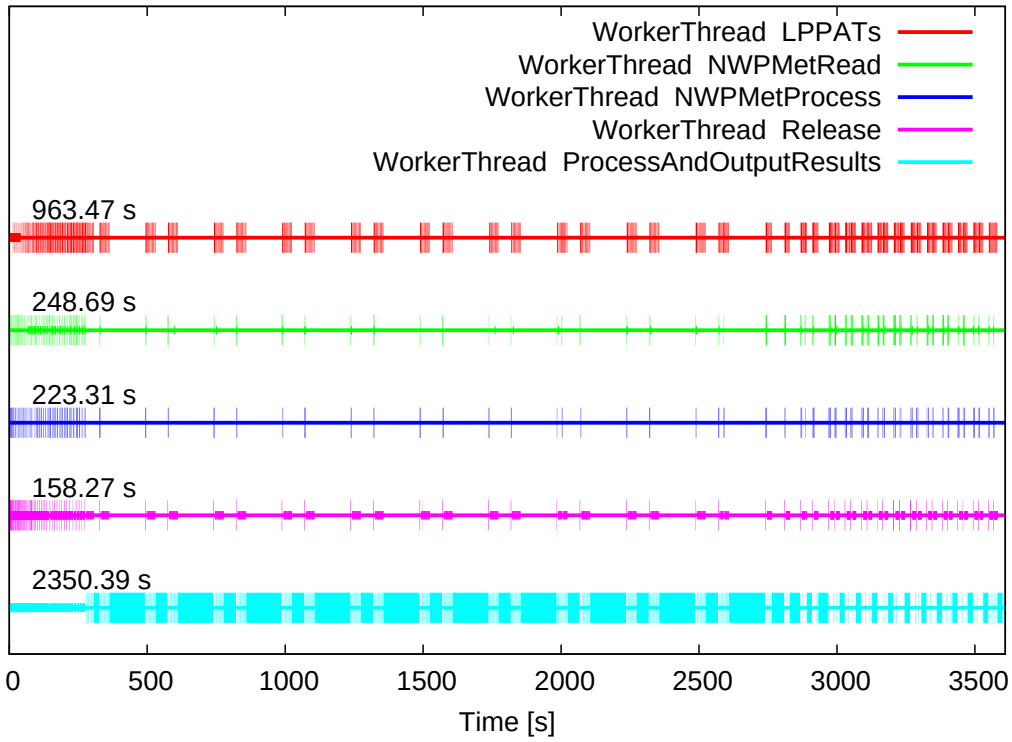


Figure 3.12: Time spent in different sections of the code in a routine run in the *standard* setup.

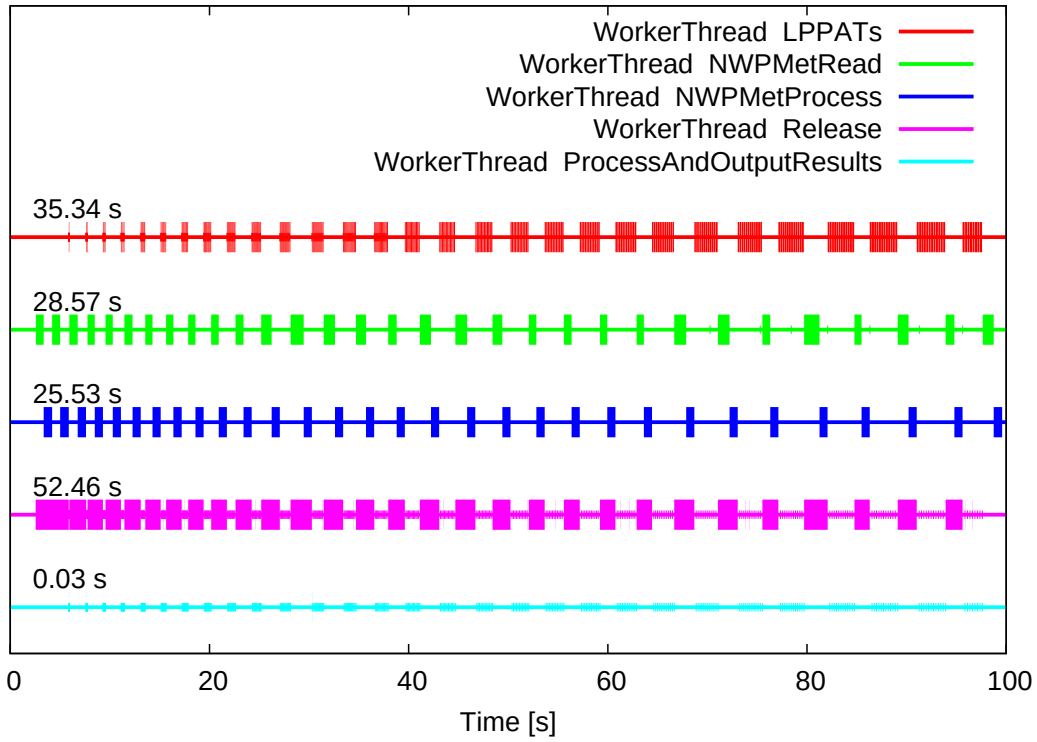


Figure 3.13: Time spent in different sections of the code at the beginning of the spinup period

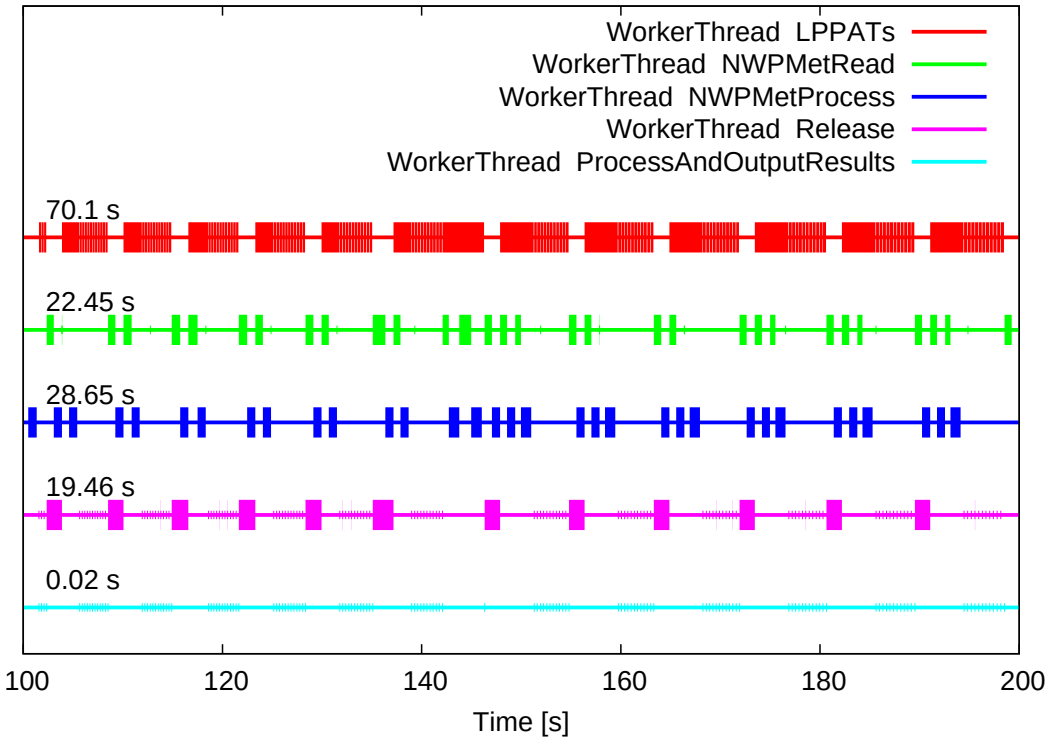


Figure 3.14: Time spent in different sections of the code towards the end of the spinup period

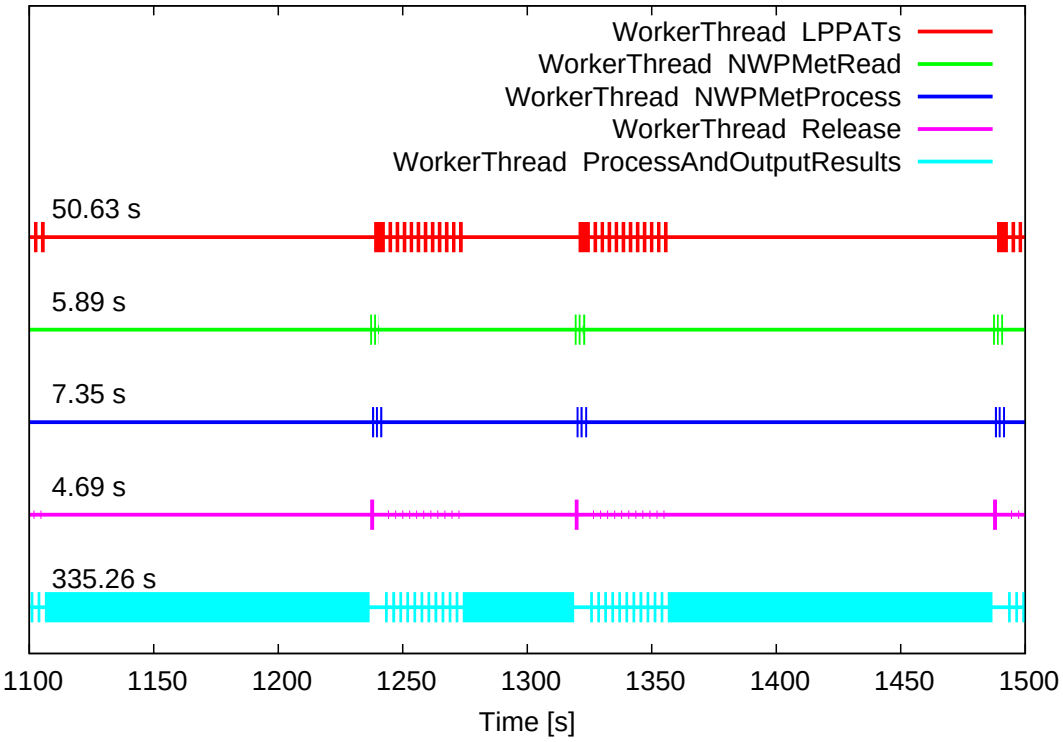


Figure 3.15: Time spent in different sections of the code during the forecast period

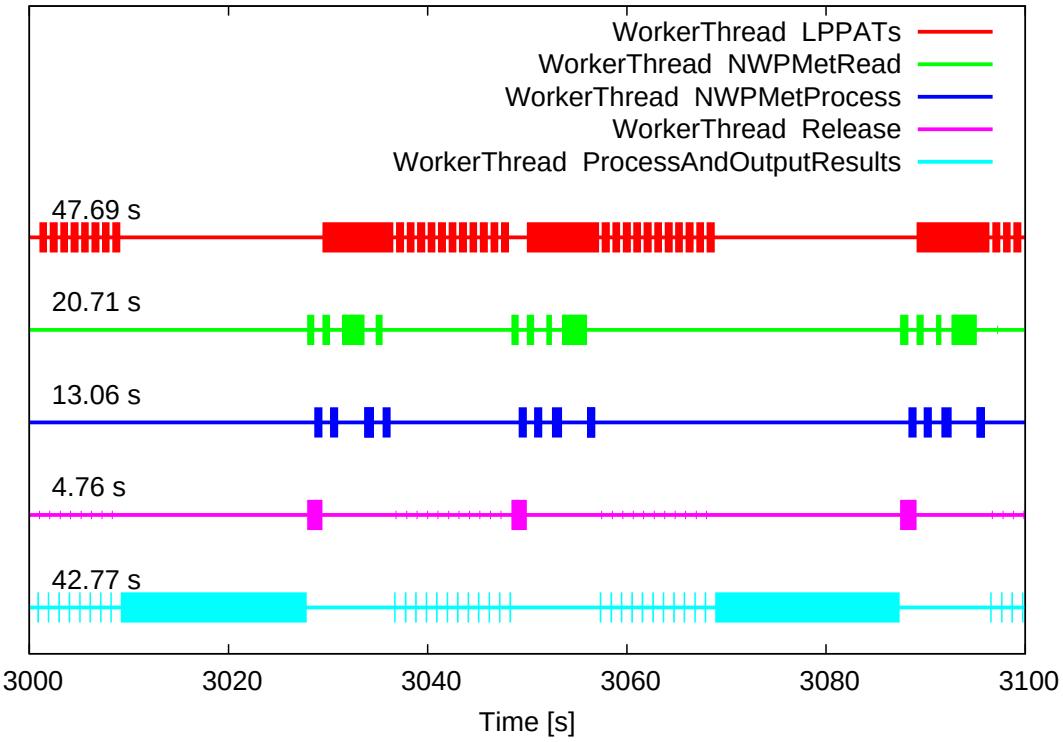


Figure 3.16: Time spent in different sections of the code at the end of the forecast period

where n is the number of cores and the time on the right hand side is that measured on one core. We then find that the fraction of time spent in the parallel code is

$$r = \frac{4}{3}\left(1 - \frac{t(\texttt{LPPATs}, 4\text{ cores})}{t(\texttt{LPPATs})}\right) \tag{3.2.3}$$

For the total time we assume

$$t_{\text{total}}(n) \approx \left(\frac{r}{n} + (1-r)\right) t(\texttt{LPPATs}) + t(\texttt{Release}) + t(\texttt{PAOR}) \tag{3.2.4}$$

For the *standard* run we use

$$\begin{aligned} t(\texttt{LPPATs}) &= 2870\,\text{s} & r &= 89\% \\ t(\texttt{Release}) &= 160\,\text{s} & t(\texttt{PAOR}) &= 2340\,\text{s}, \end{aligned} \tag{3.2.5}$$

and for the *MoreParticles* run we replace

$$t(\texttt{LPPATs}) \mapsto 10290\,\text{s} \qquad r \mapsto 95\%. \tag{3.2.6}$$

For the expensive run we use

$$\begin{aligned} t(\texttt{LPPATs}) &= 16180\,\text{s} & r &= 96\% \\ t(\texttt{Release}) &= 160\,\text{s} & t(\texttt{PAOR}) &= 870\,\text{s}\,. \end{aligned} \tag{3.2.7}$$

In Fig. 3.17 we show the estimated runtimes for different numbers of cores for the *standard*, *MoreParticles* and expensive runs based on formula 3.2.4. It should be kept in mind, however, that scaling is likely to break down for more than eight processors, for a more detailled scaling analysis of the improved code on the operational server `els035` see section 3.2.2.

From Tab. 3.6 we also see that for the routine run, multiplying the number of particles by a factor of four increases the time spent in the particle loop by a factor of 3.6 (3.1 on four cores) whereas all other timed sections of the code depend very weakly on the number of particles. Provided a similar scaling holds for the particle loop in the expensive run, we estimate an upper limit for an expensive run with twice (four times) the number of particles by doubling (quadrupling) the time spent in the parallel section of the particle loop. This is shown by the two upper blue lines in Fig. 3.17.

Provided this scaling analysis is correct, the expensive run on 8 cores takes about the same time as the routine run on 4 cores and is about 50% faster than the original serial routine run. The runtime is unchanged if the number of particles in the expensive run is doubled and the code is executed on 16 processors.

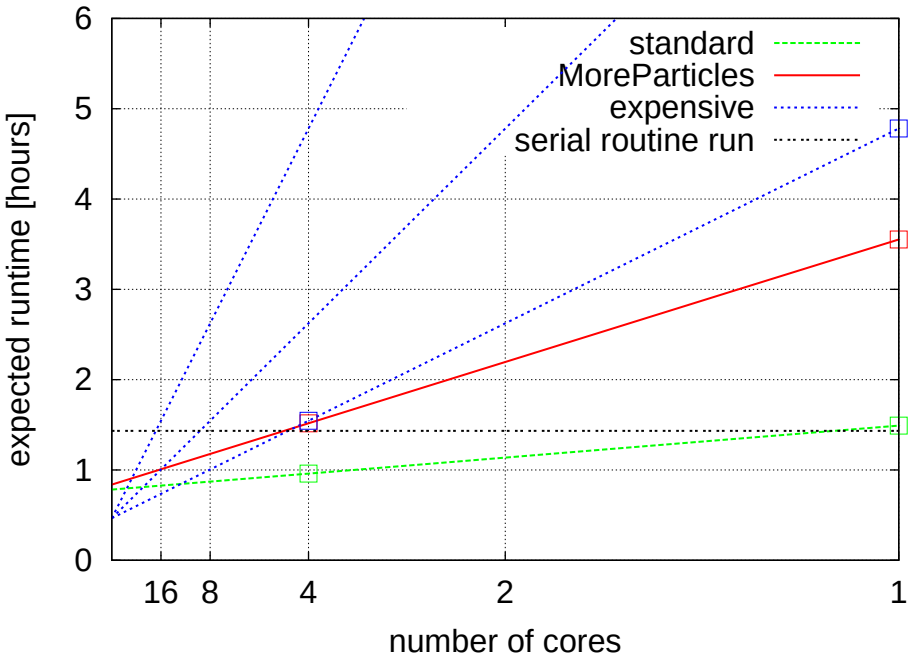


Figure 3.17: Estimated scaling of the total runtime for the *standard*, *MoreParticles* and *expensive* runs. Only the marked data points for 1 and 4 cores have been measured, all other points are predictions based on Amdahl's law (3.2.4). For the expensive run we also show an estimated time for an expensive run with double (four times) the number of particles. The runtime of the original serial routine run is shown by a horizontal line.

output group	1	2	3	4	5	6	7	8	9	10	11	12
	8.3	4.2	4.0	4.0	2.6	2.5	2.5	2.3	2.1	0.0	0.0	0.0

Table 3.8: Time spent in each of the twelve iterations (sorted by time) of `OutputGroupLoop`

section	code version		
	orginal	no `TRIM`	parallel `OuputGroupLoop`
`LPPATs`	963 s	984 s	961 s
`OutputFields()`	2020 s	519 s	472 s
`ProcessFields()`	320 s	320 s	319 s
`OutputAndProcessResults()`	2350 s	839 s	790 s
total runtime	3600 s	2116 s	2040 s

Table 3.9: Time spent in different sections of the code and total runtime for the original version of the code and for two new versions. The timings in column 2 are obtained after removing `TRIM` from `OutputFields()`, in column 3 we show results from a run where in addition `OutputGroupLoop` was parallelised. All timings are taken for a routine run on 4 cores.

Output Processing

From Tabs. 3.6 and 3.7 we see that a significant time is spent in processing and outputting the results, we investigated the subroutine `ProcessAndOutputResults()` and timed the calls to `ProcessFields()` and `OutputFields()` separately. It turns out that for a routine run with a release rate of 1562 particles/hour on four cores around 320 s are spent in `ProcessFields()` whereas 2020 s are spent in `OutputFields()`, in the following we therefore concentrate our efforts on the latter subroutine.

When writing to a file in the subroutine `OutputFields()`, the individual lines are constructed by manipulating a character string `Line`. In every step trailing blanks are removed from `Line` using the `TRIM` function and the new field is appended to the truncated line. This is inefficient as it makes frequent use of the `TRIM` function. Instead, it is much more efficient to use a pointer `CharPos` which points to the current position within `Line` and write directly to a substring of `Line`:

```
Line(CharPos:CharPos+FCW) = NewField
CharPos = CharPos + FCW + 1
```

where the length of the string `NewField` is `FCW+1`.

We checked that these code changes have no impact on the fields that are written to disk by comparing the output fields. For this we investigate the output files at the beginning of the run, which is sufficient, as we have only changed the output routines. Later in the run, the results will not agree bitwise, as for all runs the code has been compiled with `Reprod=.false.` and rounding errors will accumulate differently on multi-core runs.

We also added a timer which measures the time spent in each iteration of the loop `OutputGroupLoop`. The timing results for a representative call to the subroutine `OutputFields()` are presented in Tab. 3.8. As can be seen from this table, the output is balanced reasonably well. In a second step we parallelised the loop over output requirements.

In Tab. 3.9 we show timing results for the old and for the two new versions of the code, in all cases the code was run on 4 cores.

These results show a dramatic reduction of the time spent in the subroutine `OutputFields`, by a factor of around four. The biggest improvement comes from removing calls to the `TRIM` function.

The scheduling strategy we choose for the `OutputGroupLoop` is `DYNAMIC,16`. Setting the chunk size to 16

cores	1		2		4		6	
`LPPATs`	2456 s	—	1410 s	1.7	779 s	3.2	575 s	4.3
`CPRS`	27 s	—	46 s	0.6	61 s	0.4	66 s	0.4
`Release`	86 s		87 s		86 s		85 s	
`NWPMetRead`	35 s		34 s		32 s		31 s	
`NWPMetProcess`	202 s		198 s		190 s		189 s	
`ProcessResults`	280 s		280 s		280 s		280 s	
`OutputResults`	443 s	—	364 s	1.2	329 s	1.3	315 s	1.4
Total runtime	56 m	—	38 m	1.5	27 m	2.1	23 m	2.4

cores	8		12		16		24	
`LPPATs`	462 s	5.3	366 s	6.7	414 s	5.9	359 s	6.8
`CPRS`	64 s	0.4	63 s	0.4	98 s	0.3	74 s	0.4
`Release`	86 s		87 s		88 s		88 s	
`NWPMetRead`	32 s		32 s		33 s		37 s	
`NWPMetProcess`	183 s		187 s		189 s		210 s	
`ProcessResults`	280 s		280 s		292 s		297 s	
`OutputResults`	317 s	1.4	319 s	1.4	320 s	1.4	319 s	1.4
Total runtime	21 m	2.7	20 m	2.8	21 m	2.7	20 m	2.8

Table 3.10: Runtime for a routine run with an emission rate of 1562 particles per hour on the new server `els035` after the code improvements described in section 3.2.2. The dimensionless numbers denote the speedup relative to a run on 1 core.

means that each thread is assigned 16 continuous iterations of the loop and, as there are only 12 output requirements, one thread will process all of them and we would not expect any reduction in runtime. At this stage, running with `DYNAMIC,2` or `DYNAMIC,4`, i.e. with splitting the workload between processors, failed due to a (now fixed) bug in the code (the array `NumList` has to be private to each thread, as opposed to public). After fixing this bug, we compared the runtime (on the new server `els035`) for the scheduling strategies `DYNAMIC,16` (i.e. no splitting of workload) and `DYNAMIC,1`. We find that for `DYNAMIC,16` the time spent in the subroutine `OutputFields()` is 449 s, whereas with `DYNAMIC,1` it is reduced to 326 s, corresponding to a 30% gain in efficiency.

After all these changes the time spent on processing and outputting results is now comparable to that spent in the main particle loop.

Scaling on the new operational server `els035`

With these improvements to the code we study the scaling of the routine run on the new server `els035`. This machine is equipped with two Intel X5680 processors, which have 6 cores each. Hyperthreading is enabled so the the code can be run on up to 24 logical cores.

We measure the runtime on 1, 2, 4, 6, 8, 16 and 24 cores and also scale the number of particles released per hour by a factor of 10 and 20, i.e. the hourly release rates we study are 1562/ h, 15620/ h and 31240/h. The results are collected in Tabs. 3.10 to 3.12. In addition to the sections of the code we timed before, we also measure the time spent in the "particle update" loop (`CPRs`). Due to the small number of output groups (12 in our case) and the likely load imbalance, the `OutputGroupLoop` loop is unlikely to show good scaling, but our results show that parallelisation reduces the time spent in the subroutine `OutputResults` by around $20\% - 40\%$. The dominant section of the code is the particle loop, with its relative importance growing with the number of particles. All other timed parts of the code, such as the `Release()` subroutine, reading and processing of MetData and processing the output results depend only weakly on the number of cores and the hourly release rate.

In Fig. 3.18 we show the scaling of the particle loop (LPPATs). Clearly the scaling improves when the particle number is increased. We observe a slight reduction of the speedup when going from 12 to 16

cores	1		2		4		8		16	
`LPPATs`	23332 s	—	12759 s	1.8	6551 s	3.6	3425 s	6.8	2602 s	9.0
`CPRS`	275 s	—	446 s	0.6	618 s	0.4	638 s	0.4	1076 s	0.3
`Release`	94 s		93 s		108 s		94 s		98 s	
`NWPMetRead`	39 s		38 s		65 s		36 s		39 s	
`NWPMetProcess`	227 s		223 s		218 s		213 s		208 s	
`ProcessResults`	280 s		280 s		281 s		280 s		293 s	
`OutputResults`	445 s	—	362 s	1.2	319 s	1.4	317 s	1.4	317 s	1.4
Total runtime	408 m	—	233 m	1.8	132 m	3.1	80 m	5.1	74 m	5.5

Table 3.11: Runtime for a routine run with an emission rate of 15620 particles per hour. The dimensionless numbers denote the speedup relative to a run on 1 core.

cores	4	8
`LPPATs`	12857 s	6664 s
`CPRS`	1250 s	1287 s
`Release`	103 s	104 s
`NWPMetRead`	47 s	50 s
`NWPMetProcess`	229 s	220 s
`ProcessResults`	281 s	281 s
`OutputResults`	324 s	315 s
Total runtime	248 m	145 m

Table 3.12: Runtime for a routine run with an emission rate of 31240 particles per hour on four and eight cores.

cores. This is most likely due to the layout of the processor, which has 24 logical but only 12 physical cores if hyperthreading is enabled.

We also plot the total runtime in Fig. 3.19. After increasing the emission rate by a factor of ten the runtime is dominated by the parallelised particle loop and the scaling is much improved. Increasing the number of cores is only beneficial up to 8, when the curve flattens out. Again, this is most likely due to the limited number of physical core that are available.

In Fig. 3.20 we show the total runtime as a function of the release rate both on 4 and on 8 cores. The data is very well represented by a linear fit of the form $a + b \cdot x$ where x is the release rate. We show an extrapolation to up to 1,000 times the release rate in the original setup (1562 particles/hour) The offset is comparable in both cases, 15 m on four cores and 16 m on eight cores. For four cores the slope is about 0.2 hours / (1562 particles/hour) and on eight cores about 0.1 hours / (1562 particles/hour).

Scaling of the CPRs loop

From Tabs. 3.10 and 3.11 we observe that, rather surprisingly, the scaling of the CPRs loop is particularly poor, in fact the time spent in this loop *grows* if we increase the number of processors. To check that this is not due to a bug that was introduced when folding the parallelisation into version 5.4.4 we also ran version 5.4a of the code on a simple testcase (`Example_SingleSiteMet_Slow.txt`) which contains particle processing only. We ran the code both on the new server `els035` and on `els034`. We find that on the new server the CPRs loop takes 2.6 s to execute on one core but 12.13 s on four cores. On the other hand we find that on `els034` the loop takes 6.8 s on one core and 17.7 s on four cores, whereas the particle loop scales as expected in both cases. In section 3.1 we performed a detailled timing analysis of version 5.4a of the code for two air quality benchmarks and found that the CPRs loop scales as expected. In both benchmarks a much larger amount of the time was spent in the CPRs loop and we conclude that the poor scaling in the volcanic ash run is most probably due to the short time (per iteration) spent in this loop, overheads from parallelisation dominate and executing this part of the code in parallel is not

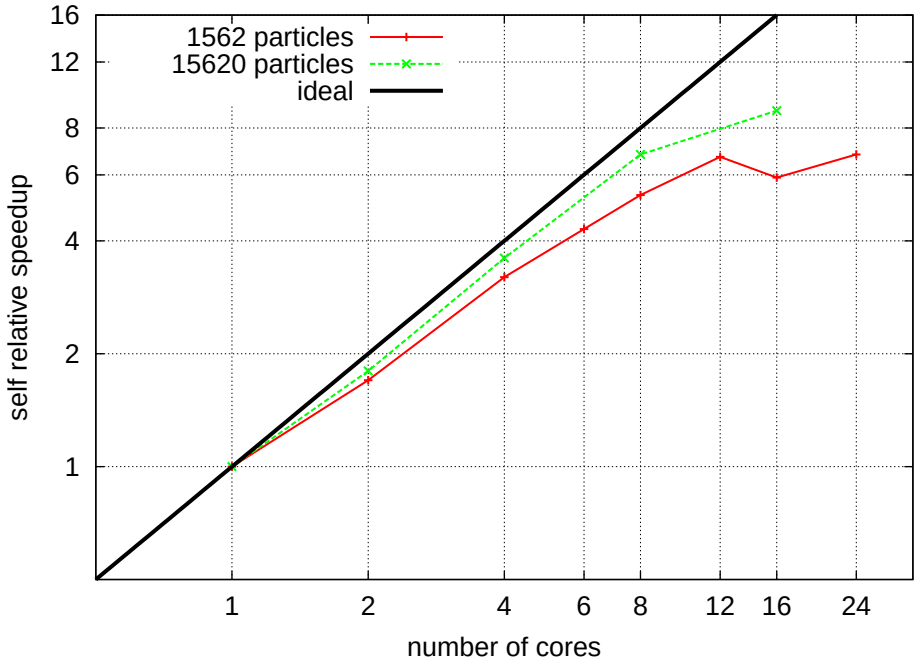


Figure 3.18: Scaling of the particle loop. The scales on both axes are logarithmic.

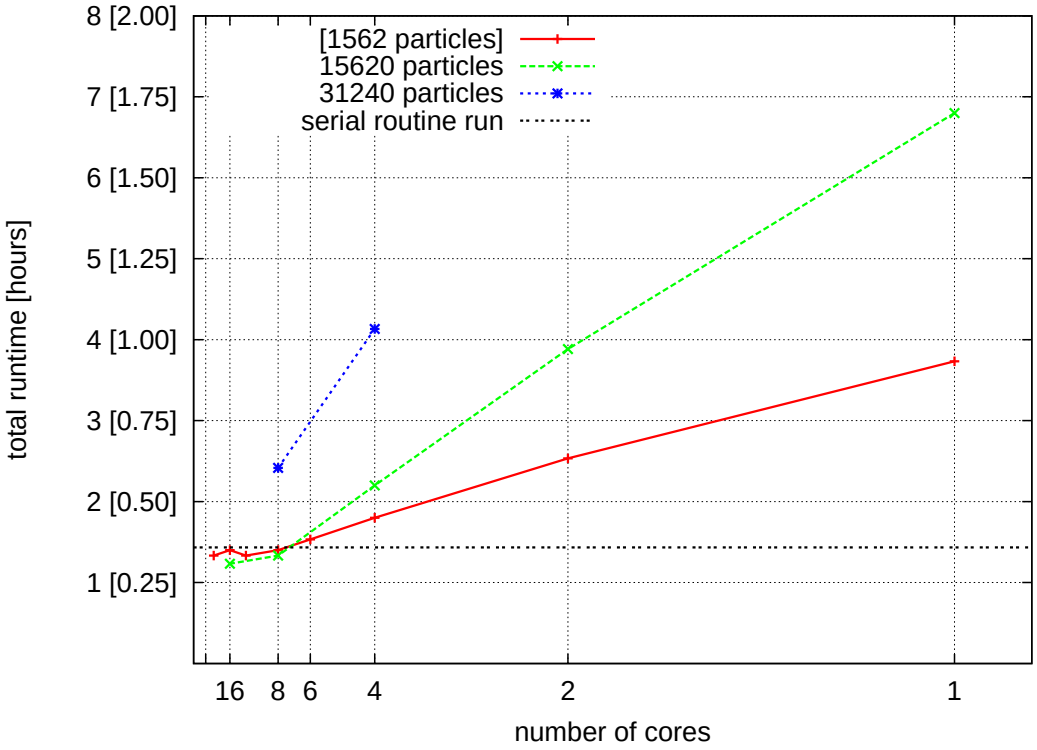


Figure 3.19: Total runtime after the improvements described in section 3.2.2. Note that the horizontal axis is divided by $1/n_{\text{cores}}$ and the scales on the vertical axis differ by a factor of four to make the plot more readable; the numbers in brackets are used for the run with the lower emission rate of 1562 particles/hour. For comparison we also show the runtime of the original serial routine run with 1562 particles/hour as a horizontal line.

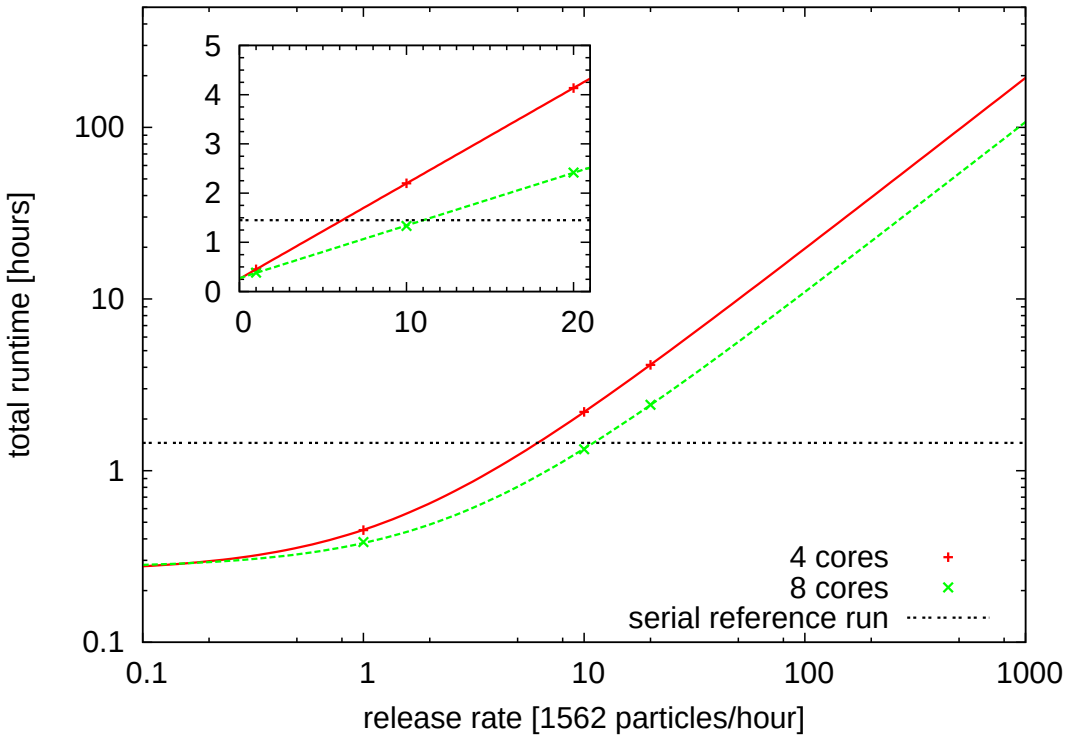


Figure 3.20: Dependence of the total runtime on the particle number. The data points are measured values, with the leftmost corresponding to the routine run with a release rate of 1562 particles/hour and the other two corresponding to release rates of 15620 and 31240 particles/hour. The linear fit of the form $a + b \cdot x$ represents the data very well. Note that both scales in the larger plot are logarithmic whereas they are linear in the smaller inset. We also show the serial reference run with 1562 particles/hour as a horizontal line.

efficient with the current setup.

The number of threads used in separate parallel regions of the code can be controlled by environment variables, in particular the time spent in the `CPRs` loop can be set by `PARTUPDATETHREADS`. We suggest setting this variable to 1 to reduce the overall runtime in future runs, this will reduce the runtime by between 5 m to 13 m on runs with an emission rate of 15620 particles/hour if the code is run on more than four cores. For a run with 1562 particles/hour the saving in runtime is less than one minute.

configuration	runtime (1)	(2)	(3)
`LPPATs`	393 s	396 s	390 s
`Release`	1 s	1070 s	1119 s
`MetRead`	2597 s	765 s	804 s
`MetProcess`	283 s	179 s	180 s
`ProcessResults`	29 s	39 s	39 s
`OutputResults`	1170 s	1160 s	292 s
Total runtime	4554 s	2718 s	1901 s

Table 3.13: Timing results for the ensemble run on ECMWF data. In configuration (1) all ensemble members are contained in one file whereas in (2) they have been split between different files. Configuration (3) is the same as (2) but with the code improvements described in section 3.2.2.

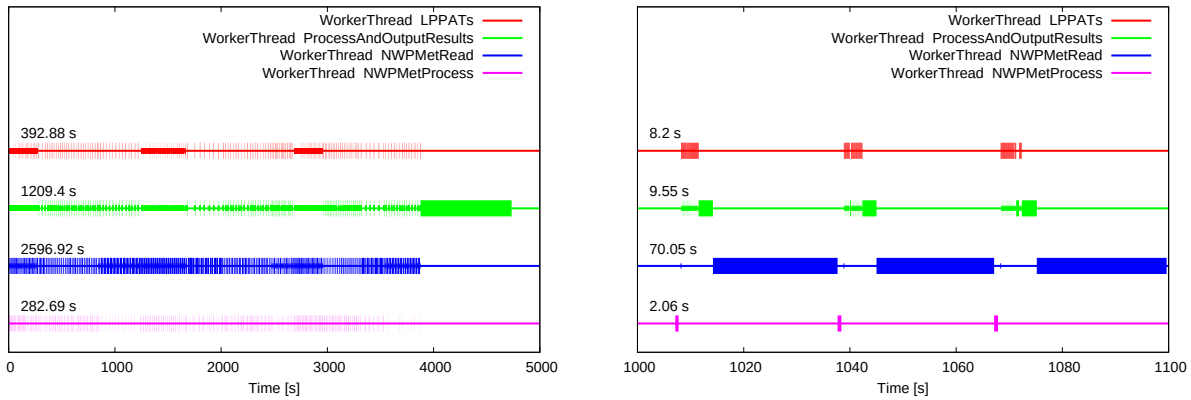


Figure 3.21: Interval timing for an ensemble run, configuration (1) (all ensemble members in one file)

3.2.3 Ensemble runs

We measured the runtime for ensemble runs which are currently carried out on the server `els047`. In the operational configuration the code is run on a set of 25 ensemble members from ECMWF, for test purposes the number of ensemble members was cut down to 3. As the data is available in GRIB format we adapted the parallel code for dynamic linking with the GRIBEX library.

All runs in this section are performed on four cores. We use three different setups: In the first, all ensemble members are contained in one file, wheras in the second and third configuration these files have been split up, so that each file only contains one ensemble member. This reduces the runtime as the code does not have to read spurious data in every iteration over the ensemble. We use the original parallel code without the improvements described in section 3.2.2 for the first two runs. The improved code was used for the third run. Tab. 3.13 contains the timing results for different sections of the code. Note that for the first run Met-on-Demand was switched off, and consequently MetData was read from disk outside the `Release()` subroutine, which explains the much smaller time spent in this subroutine.

We also investigate the time when different subroutines are active in more detail in Figs. 3.21 to 3.23. From Fig. 3.21 we see that for each iteration the time spent reading and processing MetData is much larger than the time spent propagating particles and outputting the results. A significant amount of time is spent on outputting results at the end of the run when the results from different ensemble members are combined. As can be seen from Fig. 3.22, storing the ensemble members in different files reduces the amount of time spent on reading MetData in each iteration. Using the improved version of the code reduces the amount of time spent on combining and outputting the results at the end of the run, see Fig. 3.23.

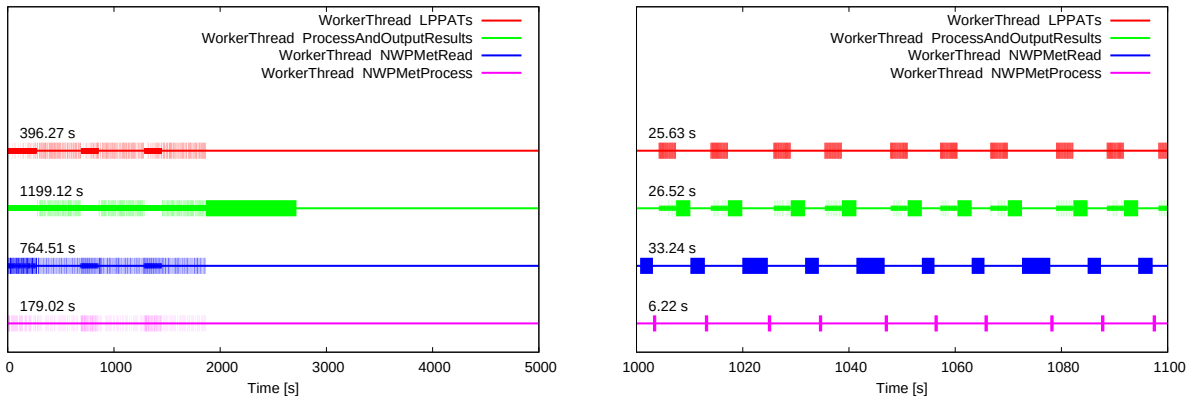


Figure 3.22: Interval timing for an ensemble run, configuration (2) (ensemble members in different files)

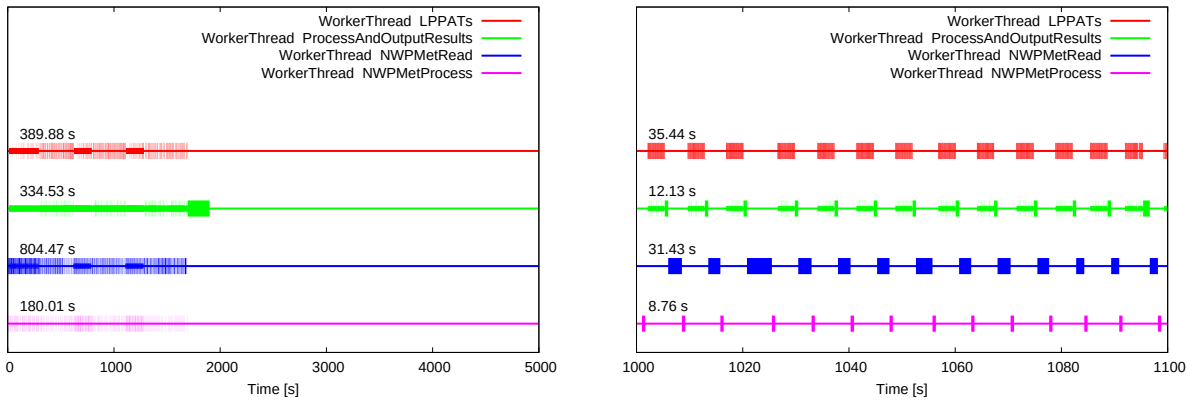


Figure 3.23: Interval timing for an ensemble run, configuration (3) (same as configuration (2), but with the improved `Output()` subroutine descibed in section 3.2.2)

It can be seen that the time spent reading MetData is significantly reduced by splitting up the ensemble members between different files. The time spent in the subroutine which outputs results to disk is reduced by a factor of four, which is in agreement with results from the routine run (see Tab. 3.9). The total runtime is cut down by over factor of two.

3.2.4 Final verification

Runs in `Optimized` mode

To verify that the final version of the parallel code reproduces the results from serial version 5.4.3, we repeated the test runs described in section 3.2.2 but increased the release rate to 15620 particles per hour. The code was then run in the following configurations:

(1) **serial 5.4.3**. Here we used two different executables:

  (1a) We copied the exectuable from the `NameIIILibrary` directory in `apdg`. This code was compiled on 18/05/2010 using version 9.0 of the Intel Fortran compiler (the default version on 64 bit servers).

  (1b) We recompiled the same code with version 11.0 of the Intel Fortran compiler.

(2) **serial 5.4.4**. This is the final parallel code but compiled in serial mode using the `LinuxIntelRelease` script, using version 11.0 of the Intel Fortran compiler.

(3) **parallel 5.4.4**. The same code, but compiled in parallel mode with version 11.0 of the Intel Fortran compiler. We ran the code on two different numbers of cores:

  (3a) 1 thread

  (3b) 4 threads

In all cases the code was compiled in `Optimized` mode. Naively we expect that the results agree between all 1 thread runs but that there are differences in the 4 thread run. These differences should, however, be small and not be discernable in the final red/gray/black plots.

We first compare the output files `Fields_grid1_201005140000.txt` at the end of the run. The data in these files agrees bitwise between runs (1b), (2) and (3a) but the results are significantly different both in run (1a) (the only run where the code was compiled with version 9.0 of the Fotran compiler) and, not surprisingly, in the 4 thread run (3b). We also looked at the file `Fields_grid1_201005081200.txt` at the beginning of the forecast period. Again, the results from runs (1b), (2) and (3a) agree, but those from run (1a) are significantly different. In contrast, the differences between the four thread run (3b) and the one thread run (3a) are relatively small, which is consistent with a different buildup of rounding errors on one and four cores due to a different order of summation.

For all runs we generated red/gray/black plots for the entire forecast period. In Figs. 3.24 to 3.26 we show these plots for flight levels 000-200 at 00:00h on 09/05/2010. The main features agree between all plots, and the results from runs (1b), (2) (3a) and (3b) are identical. Plot (1a), however, differs from these results, for example in that is shows a band of "red" ash over southern France which is not visible in any of the other plots.

Runs in `Debugging` mode

We also used code compiled in `Debugging` mode and performed two runs with a release rate of 1562 particle per hour. Here we used the following two configurations:

(4) **serial 5.4.3**. Again, we copied the executable from the `NameIIILibrary` directory in `apdg`. This code was compiled on 25/05/2010 with version 9.0 of the Intel Fortran compiler.

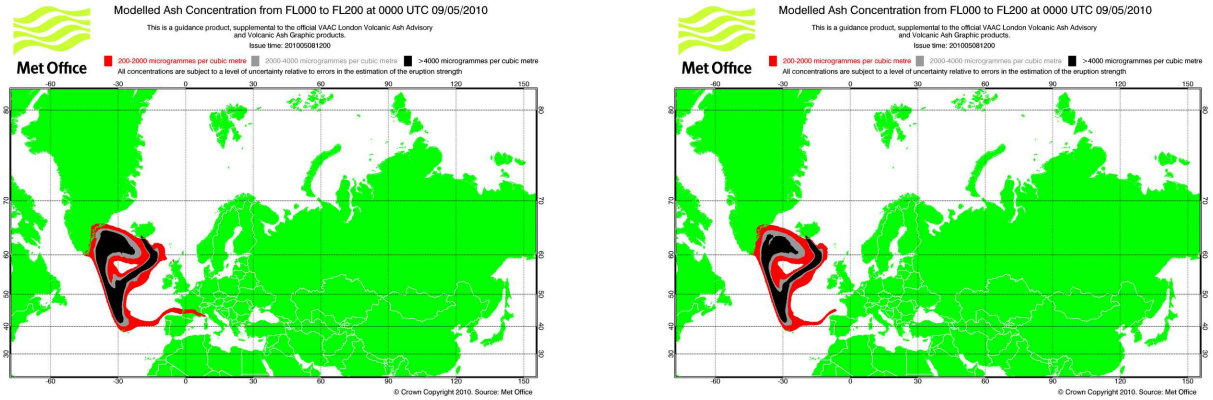


Figure 3.24: Red/black/gray plot for flight levels 000-200 at 00:00h on 09/05/2010. The release rate is 15620 particles per hour. The serial version 5.4.3 of the code, compiled in `Optmized` mode with version 9.0 (left) and 11.0 (right) of the Intel Fortran compiler was used.

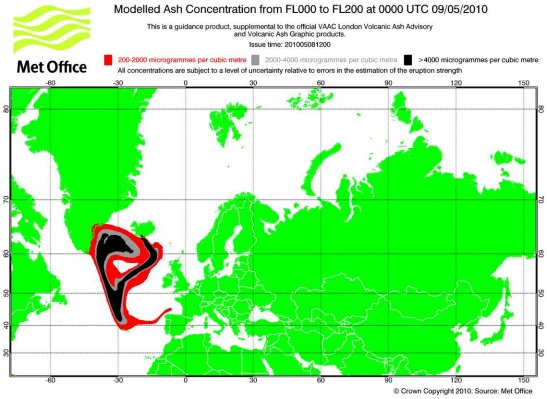


Figure 3.25: Red/black/gray plot for flight levels 000-200 at 00:00h on 09/05/2010. The release rate is 15620 particles per hour. The serial version 5.4.4 of the code, compiled in `Optmized` mode with 11.0 of the Intel Fortran compiler was used.

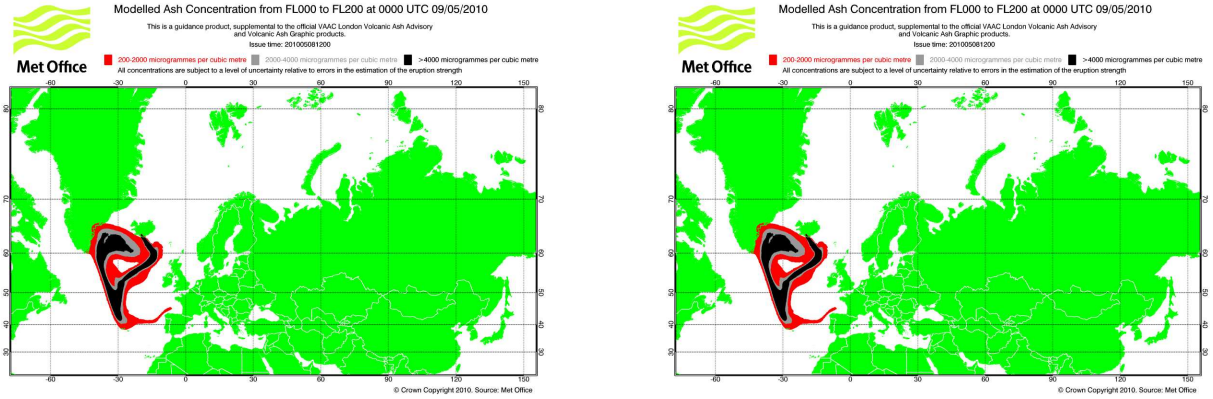


Figure 3.26: Red/black/gray plot for flight levels 000-200 at 00:00h on 09/05/2010. The release rate is 15620 particles per hour. The parallel version 5.4.4 of the code, compiled in `Optmized` mode with 11.0 of the Intel Fortran compiler was used. The results from the one thread run on the left can not be distinguished from those from the four thread run on the right

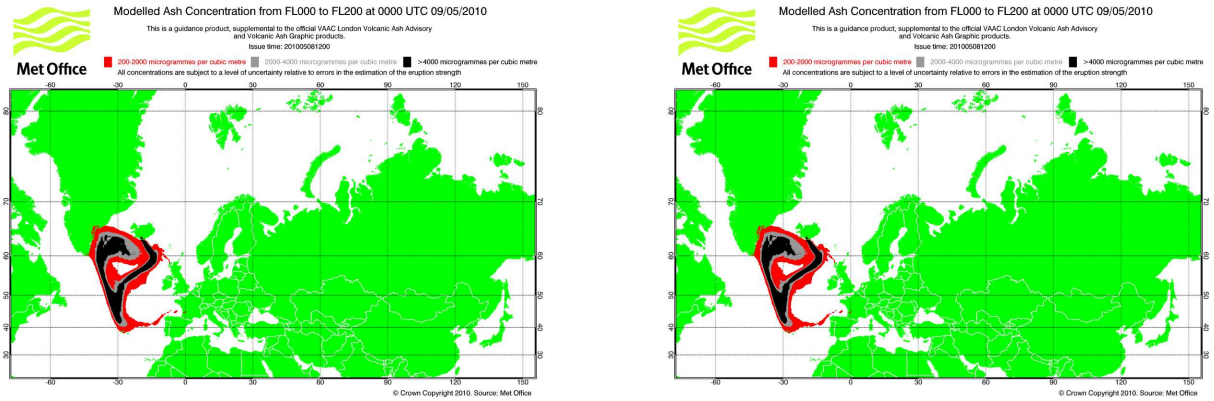


Figure 3.27: Red/black/gray plot for flight levels 000-200 at 00:00h on 09/05/2010. The release rate is 1562 particles per hour. On the left we show results from the serial version 5.4.3, compiled in `Debugging` mode with version 9.0 of the Intel Fortran compiler, wheras the results on the right are from a one thread run with version 5.4.4 of the code, compiled in `Debugging` mode with version 11.0 of the Intel Fortran compiler.

(5) **parallel 5.4.4**. We recompiled the final version of the parallel code, using version 11.0 of the Intel Fortran compiler.

We find that the output fields do not agree bitwise, although the differences are very minor in this case. The corresponding red/black/gray plots are shown in Fig. 3.27.

Note that, although in the "Debugging" runs we used a smaller release rate, the results are consistent with those from the "Optmized" runs if the same compiler version is used, in particular the band of "red" ash over southern France is absent in both serial runs.

Conclusion

We conclude that, when using version 11.0 of the Intel Fotran compiler, the results between the serial version 5.4.3 and the parallel version 5.4.4, run with one thread, are bitwise identical, as expected. Any differences in the output file from runs with more than one thread are due to a different accumulation of rounding errors and show no discernable differences in the final red/black/gray plots. In contrast, when compiling in `Optimized` mode the compiler version has a measurable impact on the results, even if the source code is identical.

Bibliography

[1] Graham Riley and Rupert Ford: *NameIII Parallelisation Study for the MetOffice - Stage 1 Intermediate report, DRAFT v0.1* 28 October 2008

[2] Graham Riley and Rupert Ford: *NameIII Parallelisation Study for the MetOffice - Stage 2 Intermediate report, DRAFT v0.1* 18 March 2009

[3] Graham Riley and Rupert Ford: *NameIII Parallelisation Study for the MetOffice - Stage 2 Final report, v1.0* 4 March 2010

Chapter 4

Further improvements

4.1 Current issues

4.1.1 Chemistry loop

The particularly poor scaling of the parallel Chemistry loop justifies further investigation of this part of the code. One possible bottleneck is memory access: if performance is limited by the speed at which data can be read from memory, increasing the number of compute threads will not lead to any benefit.

On modern processors data is not read directly from memory but stored in a local temporary cache which can be accessed much faster than main memory (see appendix A for details). Usually several levels of cache are used: a fast and small level 1 (L1) cache and a slower but bigger level 2 (L2) cache. Accessing data in the L2 cache is typically a factor ten slower than accessing data in the L1 cache. Reading data from main memory is at least one order of magnitude slower than reading data from the L2 cache. Some architectures have an additional level 3 (L3) cache, see section 2.1.1 for technical information on the systems we used. Caches are updated by reading an entire cacheline of contiguous data from memory, which allows very efficient processing of arrays if they are laid out correctly. On the other hand cache performance is poor and main memory has to be accessed directly if the code operates on non-contiguous data.

Hardware counters

We use the sampling tool of VTune to access the processor's hardware counters during program execution. The following counters and ratios were used [1]:

- `CPU_CLK_UNHALTED.THREAD`: The number of clock ticks while the thread is not halted. This is a measure for the time spent in a specific line/subroutine.
- `CPI`: Clocks per instructions retired. This ratio counts the number of clockticks which are used to execute a specific line of code/subroutine. A high value indicates poor performance.
- `L2_RQSTS.MISS`: Number of L2 cache misses

Note that counters are very processor specific. As a trial version VTune was only installed on a desktop machine (`eld497`) all results with this tool are run on this machine.

	time	clockticks	CPI	L2 cache misses
$y_i = A_{ij}x_j + b_i$	2.095 s	2,206	7.116	1,480
$y_i = A^T_{ji}x_j + b_i$	0.141 s	151	0.683	95

Table 4.1: Performance results for the two different $10,000 \times 10,000$ matrix multiplications. The last three columns count the number of samples of the hardware counter during program execution, clockticks are measured by `CPU_CLK_UNHALTED.THREAD` and cache misses by `L2_RQSTS.MISS`.

Cache misses - a simple test case

We first demostrate the use of these counters with a simple (serial) test code which performs a saxpy operation,

$$y_i = \sum_{j=1}^{n} A_{ij}x_j + b_i \qquad \text{with} \quad i = 1 \ldots n. \tag{4.1.1}$$

In the first case we realise this by

```
Do i = 1, n
  y(i) = b(i)
  Do j = 1, n
    y(i) = y(i) + A(i,j)*x(j)
  End Do
End Do
```

whereas in the second case we multiply by the transpose matrix defined by `Atranspose(i,j)=A(j,i)` and swap the array indices:

```
Do i = 1, n
  y(i) = b(i)
  Do j = 1, n
    y(i) = y(i) + Atranspose(j,i)*x(j)
  End Do
End Do
```

In FORTRAN arrays are laid out in memory with the leftmost index running fastest, i.e. `A(1,1) A(2,1) A(3,1) ... A(n,1) A` When loading data from memory into cache the processor loads an entire cache line, which consists of a contiguous section of the (linearised) array. Consequently the second matrix multiplication significantly reduces the number of L2 cache misses as subsequent iterations use data which is already available in cache. This is reflected in the results collected in Tab. 4.1 which show both a significant reduction of `CPI` and of the number of L2 cache misses.

VTUNE allows sampling of hardware counters for individual source code lines, the results for the number of L2 cache misses are shown in Fig. 4.1. Not very surprisingly bad cache performance in the `saxpyMul` subroutine can be traced back to line 39 in the inner loop: `y(i) = A(i,j)*x(j)`.

We conclude that the hardware counters listed in section 4.1.1 can be used to analyse the L2 cache performance of our test code. In the following section we will use these counters to investigate the main chemistry loop in the NAME code. As mentioned above these counters are very hardware specific and it is worth investigating whether other tools such as PAPI [2] can be used to access more general hardware counters. Discussions with the Linux team revealed that the installation of PAPI would require recompilation of the operating system kernel and is not feasible at the moment.

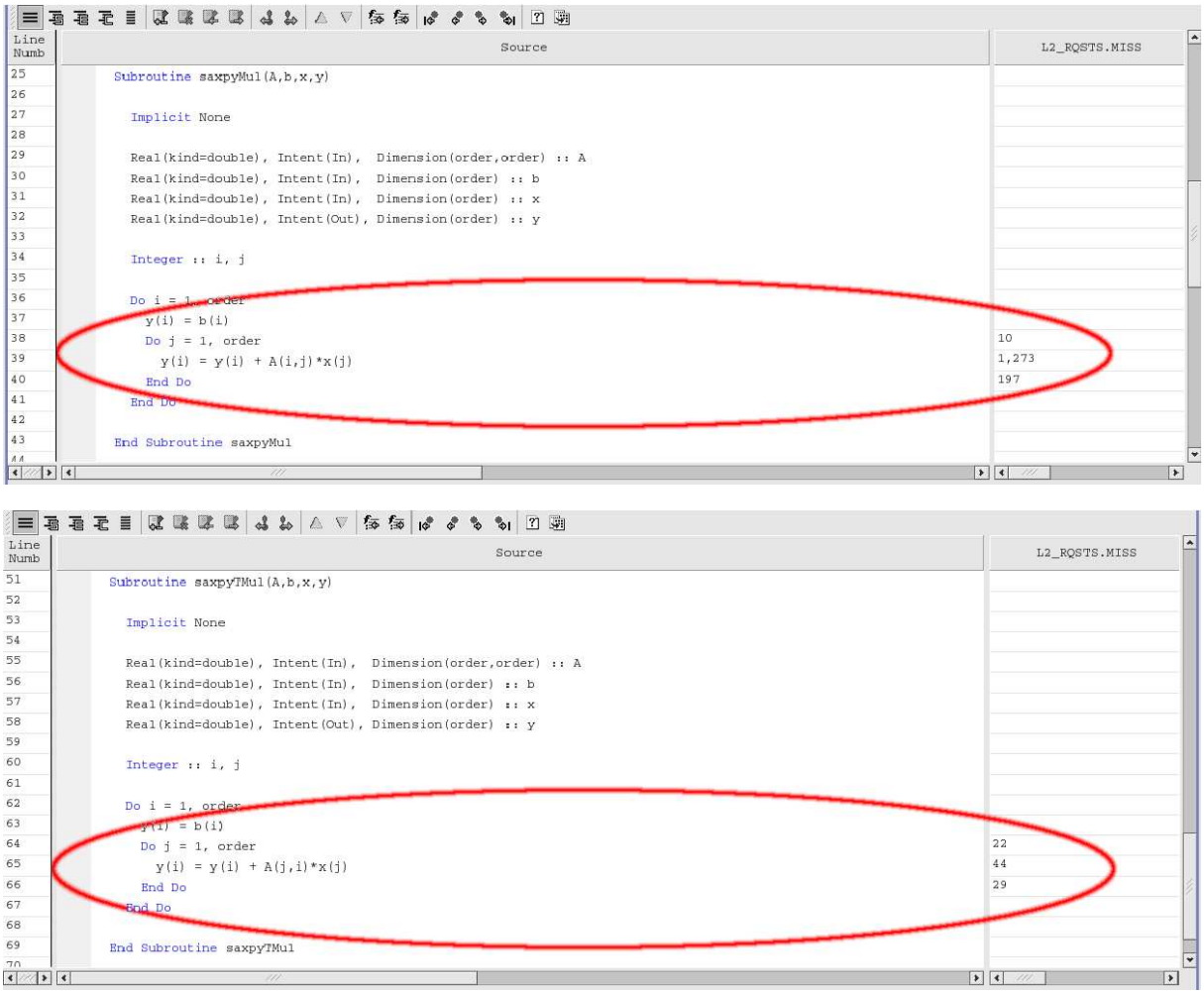


Figure 4.1: Cache misses in the subroutines `saxpyMul` ($y_i = \sum A_{ij}x_j + b_i$, top) and `saxpyTMul` ($y_i = \sum A^T_{ji}x_j + b_i$, bottom) as measured by the hardware counter `L2_RQSTS.MISS`.

Chemistry loop structure

In the subroutine `ChemistryUpdate()` the particle information (stored in an array which contains the position and other characteristics of each particle) is converted to gridbox-specific information, which is stored in a structure of `Type(ChemistryState_)` (see below). This conversion is carried out in `InitChemistryGridboxes()`, which is not parallelised. As demonstrated in section 3.1.4 this part of the code is computationally not very expensise. This, however, does not mean that it is irrelevant, as the way data arrays are initialised may have an effect on the performance of the parallel chemistry loop.

The structure of the parallelised main chemistry loop is given in the following. For each section of the code we also give the total time as measured by the Timer module when running on the server `els034`:

- loop over x coordinate: `Do iX = 1, HGrid%nX`
  - loop over y coordinate: `Do iY = 1, HGrid%nY`
    - first loop over z coordinate: `Do iZ = 1, HGrid%nZ`
      - Calculate position of centre of gridbox and convert to correct chemistry coordinate system: `X2Position()`, `ConvertToH()`, `Position2X()` [2.51 s]
      - Calculate zenith angle: `CalcZenithAngle()` [47.19 s]
      - Get MetData at centre of chemistry gridbox: `GetMetForChemistry()` [15.58 s]
      - Determine gridbox concentrations: `ConvertMassToConc()`, `ConvertChemFieldToConc()`, `SetEffectiveOzoneConc()` [11.36 s]
      - Main chemistry calculation: `MainChemistry()` [23.13 s]
      - Update array of gridbox concentrations [2.21 s]
    - Perform Ozone deposition and mixing: `OzoneColumnDryDepAndMixing()` [0.18 s]
    - second loop over z coordinate: `Do iZ = 1, HGrid%nZ`
      - Convert updated concentrations back to particle array: `ConvertConcToChemField()`, `ConvertConcToMass()`, `UpdateParticleMasses()` [23.50 s]

The timings suggest that most of the time was spent in the subroutine `CalcZenithAngle()`, followed by the main chemistry loop and the set of subroutines that convert the updated concentrations back to the particle array. This has to be compared to the results obtained with the hardware counters sampled by vTune, see Fig. 4.2.

Structure `Type(ChemistryState_)`

The structure `Type(ChemistryState_)` is used to store the information of the chemistry grid such as concentrations and gridbox volumes and plays a central role in the chemistry calculations:

```
Type :: ChemistryState_ ! State of a chemistry calculation.
  ! If changing this type, remember restart read and write routines.
  Logical                      :: Initialised
  Integer,             Pointer :: ParticleIndex(:)
  Type(GridboxState_), Pointer :: GridboxState(:,:,:)
  Real(8),             Pointer :: Volume(:,:,:)
  Real(8),             Pointer :: Mass(:,:,:,:)
  Real(8),             Pointer :: OldMass(:,:,:,:)
  Real(8),             Pointer :: ChemField(:,:,:,:)
  Real(8),             Pointer :: O3LIM(:,:,:)
  Real(8),             Pointer :: H2O2LIM(:,:,:)
  Real(8),             Pointer :: O3DryDeposition(:,:)
  Real(8),             Pointer :: BLOld(:,:)
  ! Initialised     :: Indicates that the chemistry state has been initialised.
  ! ParticleIndex   :: Indices of the particles (grouped by gridbox).
```

```
  ! GridboxState    :: Used with ParticleIndex to identify particles in each grid box.
  ! Volume          :: Volume of each grid box in cm3 (constant throughout model run).
  ! Mass            :: Mass (in g) in each grid box (for species held on particles).
  ! OldMass         :: Initial mass (in g) in each grid box (for species held
  !                    on particles).
  ! ChemField       :: Chemistry field concentrations (in g/m3) (for species held
  !                    on chemistry grid).
  ! O3LIM           :: Background O3 field (in g/m3) (on chemistry grid).
  ! H2O2LIM         :: Background H2O2 field (in g/m3) (on chemistry grid).
  ! O3DryDeposition :: O3 dry deposition field (on 2-d chemistry grid).
  ! BLOld           :: Height of the boundary layer at previous
  !                    synchronisation time (on 2-d chemistry grid).
End Type ChemistryState_
```

In particular it contains the following three- and four-dimensional arrays:

- `GridboxState(x,y,z)`: Information on particle indices in gridbox with integer coordinates (x, y, z)
- `Volume(x,y,z)`: Volume of gridbox (x, y, z)
- `Mass(x,y,z,j)` and `OldMass(x,y,z,j)`: Total mass of species j in gridbox with coordinates (x, y, z)
- `ChemField(x,y,z,j)`: Chemistry field concentrations
- `O3LIM(x,y,z)`, `H2O2LIM(x,y,z)`: Ozone and H_2O_2 concentrations in gridbox (x, y, z).

All arrays are laid out in the order $(x, y, z, \text{Species})$. `O3DryDeposition` and `BLOld` are two dimensional (x, y) arrays.

We ran the code with the short air quality test case (`air_quality_test_short.nif`) as input on the desktop machine `eld497` and analysed the results with vTune. The hardware counters which we monitored during the test run are discussed in section 4.1.1.

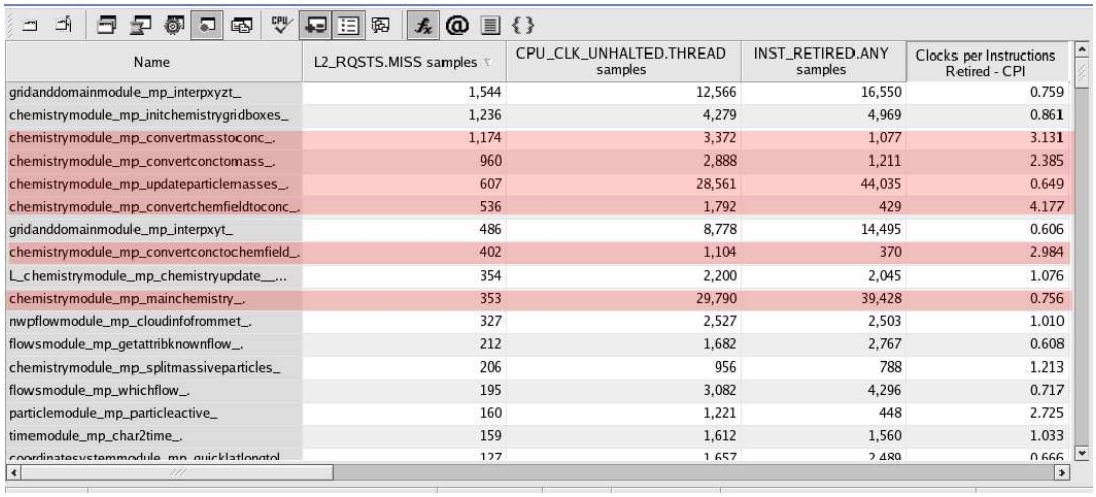
Fig. 4.2 shows vTune sampling results for individual subroutines, sorted by the value of L2 cache misses as measured by `L2_RQSTS.MISS`. Note that the number of cache misses is high in the subroutines `ConvertMassToConc()`, `ConvertConcToMass()`, `UpdateParticleMasses()`, `ConvertChemFieldToConc()`, `ConvertConcToChemField()` and `MainChemistry()`, which are called inside the parallel chemistry loop. Many of these subroutines also have a large `CPI` which indicates bad performance. In Fig. 4.3 we show the number of L2 cache misses in the subroutines `ConvertMassToConc()` and `ConvertConcToMass()` in more detail. The number of cache misses is particularly high for lines of code that access arrays within the structure `Type(ChemistryState_)`.

The largest number of cache misses occurs in the subroutines `InterpXYZT()` and `InitChemistryGridboxes()`. The latter is called from outside the chemistry loop, but it needs to be investigated further as to whether `InterpXYZT` is called indirectly from within the parallel loop.

The time spent in each subroutine, if measured with `CPU_CLK_UNHALTED.THREAD` is particularly large for `UpdateParticleMasses()` and `MainChemistry()` which is not consistent with the timing results reported above for a run on `els034`. This raises questions about the comparability of results on different processor architectures. On the other hand, the quality of all timing results is debatable as a high number of calls (one for every iteration of the loop over chemistry gridboxes) to very fast subroutines is measured.

Loop structure

In the code the arrays in `Type(ChemistryState_)` are looped over in the order $x \to y \to z(\to \text{Species})$:



Name	L2_RQSTS.MISS samples	CPU_CLK_UNHALTED.THREAD samples	INST_RETIRED.ANY samples	Clocks per Instructions Retired - CPI
gridanddomainmodule_mp_interpxyzt_	1,544	12,566	16,550	0.759
chemistrymodule_mp_initchemistrygridboxes_	1,236	4,279	4,969	0.861
chemistrymodule_mp_convertmasstoconc_.	1,174	3,372	1,077	3.131
chemistrymodule_mp_convertconctomass_.	960	2,888	1,211	2.385
chemistrymodule_mp_updateparticlemasses_.	607	28,561	44,035	0.649
chemistrymodule_mp_convertchemfieldtoconc_.	536	1,792	429	4.177
gridanddomainmodule_mp_interpxyt_	486	8,778	14,495	0.606
chemistrymodule_mp_convertconctochemfield_.	402	1,104	370	2.984
L_chemistrymodule_mp_chemistryupdate__...	354	2,200	2,045	1.076
chemistrymodule_mp_mainchemistry_.	353	29,790	39,428	0.756
nwpflowmodule_mp_cloudinfofrommet_.	327	2,527	2,503	1.010
flowsmodule_mp_getattribknownflow_.	212	1,682	2,767	0.608
chemistrymodule_mp_splitmassiveparticles_	206	956	788	1.213
flowsmodule_mp_whichflow_.	195	3,082	4,296	0.717
particlemodule_mp_particleactive_	160	1,221	448	2.725
timemodule_mp_char2time_.	159	1,612	1,560	1.033
coordinatesystemmodule_mp_quicklatlongtol	127	1,657	2,489	0.666

Figure 4.2: Hardware counters for subroutine calls in a `air_quality_test_short.txt` run sorted by the number of L2 cache misses. Subroutines called from inside the parallelised chemistry loop are highlighted in red.

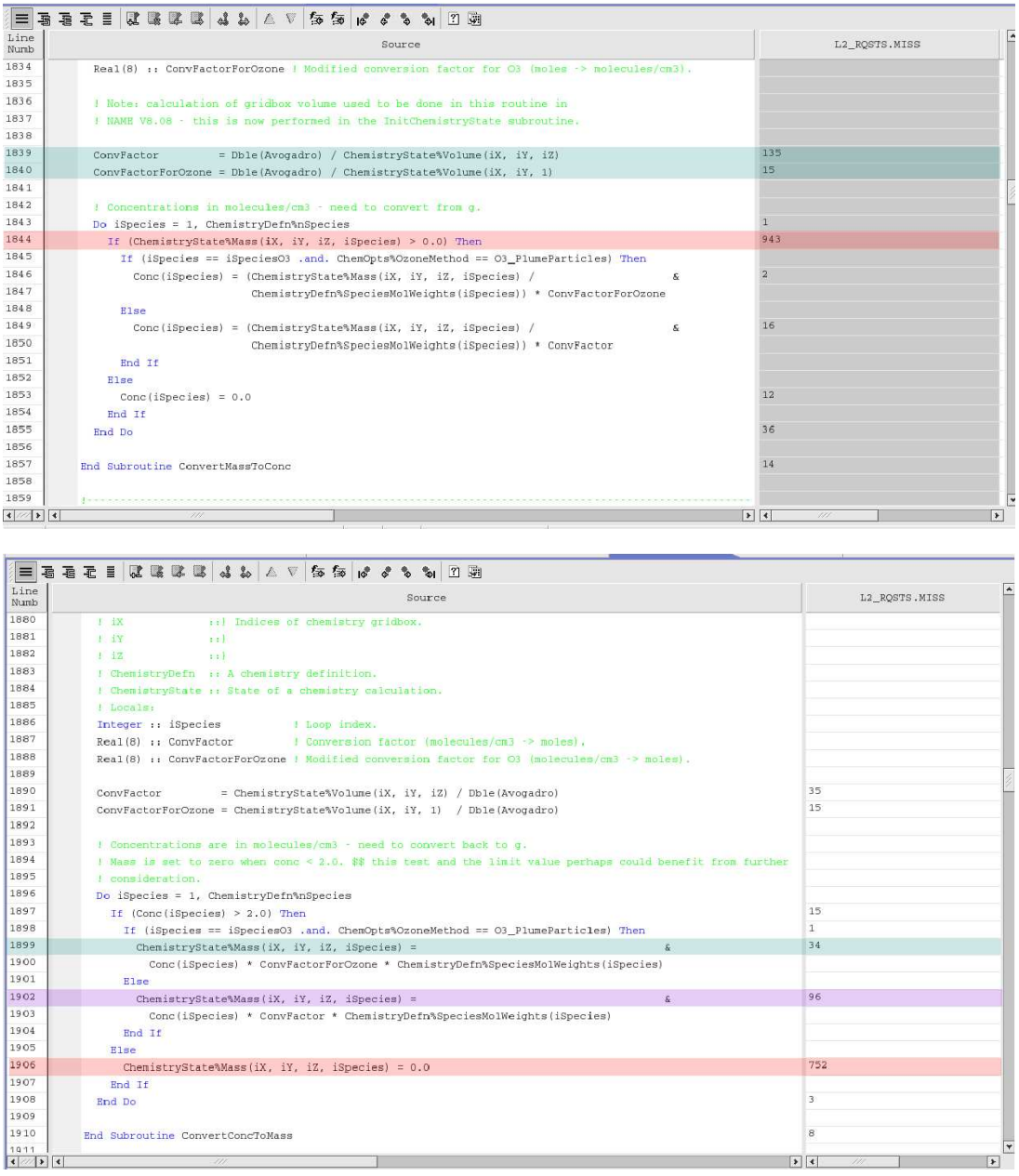
```
Do iX = 1, HGrid%nX
  Do iY = 1, HGrid%nY
    Do iZ = 1, HGrid%nZ
      [ Do iSpecies = 1, ChemistryDefn%nSpecies
        ...
        End Do ]
      ...
    End Do
  End Do
End Do
```

This implies that the index which changes fastest in the loop is the rightmost index in the array structures in `Type(ChemistryState_)` which is inefficient considering the way FORTRAN arrays are laid out in memory.

As it is only possible to swap the order of the x and y loop we decided to change the layout of the following three- and four-dimensional arrays in the structure `ChemistryState_`:

- `GridboxState(z,y,x)`
- `Volume(z,y,x)`
- `Mass(j,z,y,x)`
- `OldMass(j,z,y,x)`
- `ChemField(j,z,y,x)`
- `O3LIM(z,y,x)`
- `H2O2LIM(z,y,x)`

With this new layout the *leftmost* index changes fastest as it is looped over in the innermost loop and we expect this to reduce the number of cache misses.



```
Line Numb | Source                                                                                                  | L2_RQSTS.MISS
1834      |   Real(8) :: ConvFactorForOzone ! Modified conversion factor for O3 (moles -> molecules/cm3).           |
1835      |                                                                                                         |
1836      |   ! Note: calculation of gridbox volume used to be done in this routine in                              |
1837      |   ! NAME V8.08 - this is now performed in the InitChemistryState subroutine.                            |
1838      |                                                                                                         |
1839      |   ConvFactor         = Dble(Avogadro) / ChemistryState%Volume(iX, iY, iZ)                               | 135
1840      |   ConvFactorForOzone = Dble(Avogadro) / ChemistryState%Volume(iX, iY, 1)                                | 15
1841      |                                                                                                         |
1842      |   ! Concentrations in molecules/cm3 - need to convert from g.                                           |
1843      |   Do iSpecies = 1, ChemistryDefn%nSpecies                                                               | 1
1844      |     If (ChemistryState%Mass(iX, iY, iZ, iSpecies) > 0.0) Then                                           | 943
1845      |       If (iSpecies == iSpeciesO3 .and. ChemOpts%OzoneMethod == O3_PlumeParticles) Then                  |
1846      |         Conc(iSpecies) = (ChemistryState%Mass(iX, iY, iZ, iSpecies) /                    &              | 2
1847      |                           ChemistryDefn%SpeciesMolWeights(iSpecies)) * ConvFactorForOzone               |
1848      |       Else                                                                                              |
1849      |         Conc(iSpecies) = (ChemistryState%Mass(iX, iY, iZ, iSpecies) /                    &              | 16
1850      |                           ChemistryDefn%SpeciesMolWeights(iSpecies)) * ConvFactor                       |
1851      |       End If                                                                                            |
1852      |     Else                                                                                                |
1853      |       Conc(iSpecies) = 0.0                                                                              | 12
1854      |     End If                                                                                              |
1855      |   End Do                                                                                                | 36
1856      |                                                                                                         |
1857      | End Subroutine ConvertMassToConc                                                                        | 14
1858      |                                                                                                         |
1859      | !-------------------------------------------------------------------------------------------------------|
```

```
Line Numb | Source                                                                                                  | L2_RQSTS.MISS
1880      |   ! iX             :: Indices of chemistry gridbox.                                                     |
1881      |   ! iY             ::                                                                                   |
1882      |   ! iZ             ::                                                                                   |
1883      |   ! ChemistryDefn  :: A chemistry definition.                                                           |
1884      |   ! ChemistryState :: State of a chemistry calculation.                                                 |
1885      |   ! Locals:                                                                                             |
1886      |   Integer :: iSpecies          ! Loop index.                                                            |
1887      |   Real(8) :: ConvFactor        ! Conversion factor (molecules/cm3 -> moles).                            |
1888      |   Real(8) :: ConvFactorForOzone ! Modified conversion factor for O3 (molecules/cm3 -> moles).           |
1889      |                                                                                                         |
1890      |   ConvFactor         = ChemistryState%Volume(iX, iY, iZ) / Dble(Avogadro)                               | 35
1891      |   ConvFactorForOzone = ChemistryState%Volume(iX, iY, 1)  / Dble(Avogadro)                               | 15
1892      |                                                                                                         |
1893      |   ! Concentrations are in molecules/cm3 - need to convert back to g.                                    |
1894      |   ! Mass is set to zero when conc < 2.0. $$ this test and the limit value perhaps could benefit from further |
1895      |   ! consideration.                                                                                      |
1896      |   Do iSpecies = 1, ChemistryDefn%nSpecies                                                               |
1897      |     If (Conc(iSpecies) > 2.0) Then                                                                      | 15
1898      |       If (iSpecies == iSpeciesO3 .and. ChemOpts%OzoneMethod == O3_PlumeParticles) Then                  | 1
1899      |         ChemistryState%Mass(iX, iY, iZ, iSpecies) =                                      &              | 34
1900      |           Conc(iSpecies) * ConvFactorForOzone * ChemistryDefn%SpeciesMolWeights(iSpecies)               |
1901      |       Else                                                                                              |
1902      |         ChemistryState%Mass(iX, iY, iZ, iSpecies) =                                      &              | 96
1903      |           Conc(iSpecies) * ConvFactor * ChemistryDefn%SpeciesMolWeights(iSpecies)                       |
1904      |       End If                                                                                            |
1905      |     Else                                                                                                |
1906      |       ChemistryState%Mass(iX, iY, iZ, iSpecies) = 0.0                                                   | 752
1907      |     End If                                                                                              |
1908      |   End Do                                                                                                | 3
1909      |                                                                                                         |
1910      | End Subroutine ConvertConcToMass                                                                        | 8
1911      |                                                                                                         |
```

Figure 4.3: L2 cache misses in the subroutines `ConvertMassToConc()` and `ConvertConcToMass()` for the run in Fig. 4.2. Lines with a high number of cache misses are highlighted.

	subroutine	L2 misses before	L2 misses after	clockticks before	clockticks after	CPI before	CPI after
	`InterpXYZT()`	1,544	1,231	12,566	11,891	0.759	0.707
□	`InitChemistryGridboxes()`	1,236	782	4,279	3,894	0.861	0.755
⋆	`ConvertMassToConc()`	1,174	125	3,372	1,000	3.131	1.021
⋆	`ConvertConcToMass()`	960	45	2,888	617	2.385	0.506
⋆	`UpdateParticleMasses()`	607	403	28,561	26,980	0.649	0.617
⋆	`ConvertChemFieldsToConc()`	536	43	1,792	459	4.177	1.070
	`InterpXZT()`	486	399	8,778	8,797	0.606	0.618
⋆	`ConvertConcToChemField()`	402	4	1,104	99	2.984	0.476
	`ChemistryUpdate()`	354	304	2,200	2,175	1.076	0.994
⋆	`MainChemistry()`	353	241	29,790	29,718	0.756	0.757

Table 4.2: Values of hardware counters before and after restructuring the layout of arrays in `Type(ChemistryState_)`, sorted by number of L2 cache misses as measured by `L2_RQSTS.MISS`. Clockticks are measured by `CPU_CLK_UNHALTED.THREAD`. Subroutines called directly from the parallel chemistry loop are marked by a ⋆. Subroutines marked with a □ are called from within the subroutine `ChemistryUpdate()` but outside the parallel region.

Improved array layout

After verifing that changing the array layout does not alter the results (the code was compiled in `Debugging` mode in this case and the output fields were compared) we sampled the hardware counters again[1]. The results are shown in Fig. 4.5 and are compared to those in Fig. 4.2 and Tab. 4.2. In all cases the number of L2 cache misses is reduced by the new array layout, in some cases this reduction is dramatic, as for example in the subroutines `ConvertMassToConc()` and `ConvertConcToMass()`. The value of the counter `L2_RQSTS.MISS` for these subroutines is shown in Fig. 4.4 which has to be compared to Fig. 4.3.

For these subroutines the `CPI` is reduced, which implies an improvement in efficiency. Note, however, that the performance gain in the computationally most expensive subroutines `UpdateParticleMasses()` and `MainChemistry()` is not very dramatic. Further efforts should concentrate on these two subroutines.

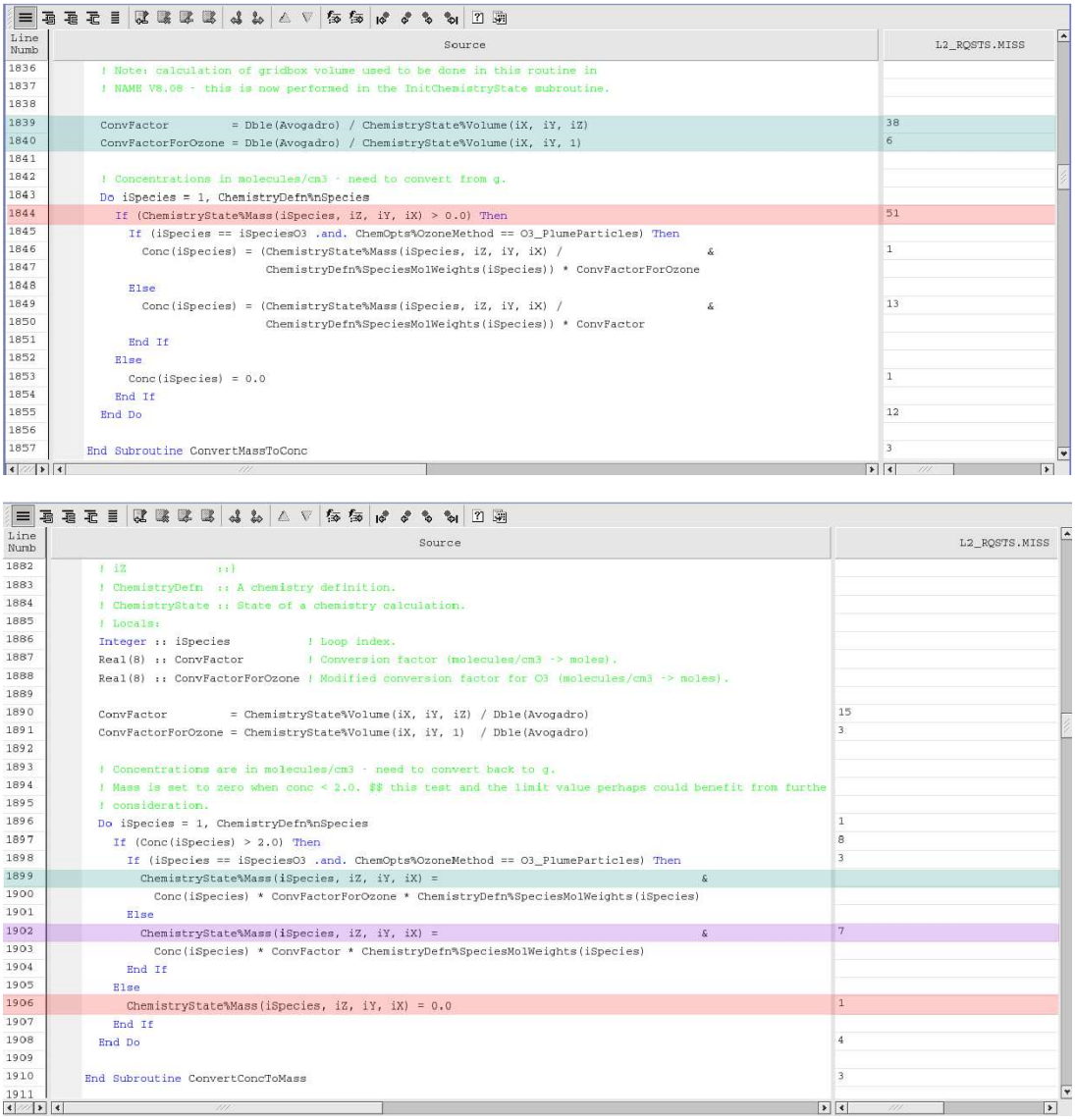
Runtime and speedup

We measured the time spent in the parallelised chemistry loop before and after restructuring the arrays in `Type(ChemistryState_)`. The results obtained on `els034` are collected in Tab. 4.3.

The self-relative speedup for all parallelised loops is plotted in Figs. 4.6 and 4.7. The scheduling strategy for the particle loop (LPPATs) was hardwired to `"static,16"`, the value reported to give good results in [4]. For the particle update (CPRs) and chemistry loop the scheduling strategy was set at runtime. The results shown in this section were obtained with the default value `static`.

From Tab. 4.3 it can be seen that reordering the index structure of the arrays in `ChemistryState_` leads to a significant increase of performance of the order of 10% with the benefit increasing with the number of threads used. Scaling is not improved significantly but the plateau is shifted to slightly larger number of threads.

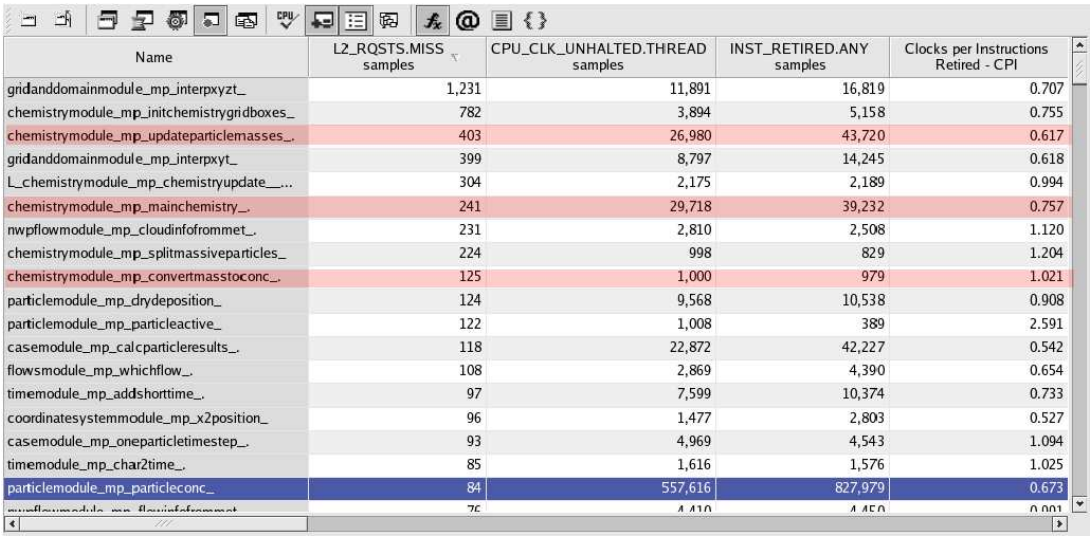
[1] The code with the original array layout includes some further changes made to the `apdg` directory between 19/03/2010 and 25/03/2010. We checked that these changes do not affect the timings of the chemistry loop. Also, after measuring the times we fixed a minor bug in the subroutine `SetBackgroundFieldsFromSTOCHEM()` (the arrays `O3LIM` and `H2O2LIM` were accessed in the wrong order). This subroutine is not called from within the parallel loop so this will have no impact on the timing results.



Line Numb	Source	L2_RQSTS.MISS
1836	! Note: calculation of gridbox volume used to be done in this routine in	
1837	! NAME V8.08 - this is now performed in the InitChemistryState subroutine.	
1838		
1839	ConvFactor = Dble(Avogadro) / ChemistryState%Volume(iX, iY, iZ)	38
1840	ConvFactorForOzone = Dble(Avogadro) / ChemistryState%Volume(iX, iY, 1)	6
1841		
1842	! Concentrations in molecules/cm3 - need to convert from g.	
1843	Do iSpecies = 1, ChemistryDefn%nSpecies	
1844	If (ChemistryState%Mass(iSpecies, iZ, iY, iX) > 0.0) Then	51
1845	If (iSpecies == iSpeciesO3 .and. ChemOpts%OzoneMethod == O3_PlumeParticles) Then	
1846	Conc(iSpecies) = (ChemistryState%Mass(iSpecies, iZ, iY, iX) / &	1
1847	ChemistryDefn%SpeciesMolWeights(iSpecies)) * ConvFactorForOzone	
1848	Else	
1849	Conc(iSpecies) = (ChemistryState%Mass(iSpecies, iZ, iY, iX) / &	13
1850	ChemistryDefn%SpeciesMolWeights(iSpecies)) * ConvFactor	
1851	End If	
1852	Else	
1853	Conc(iSpecies) = 0.0	1
1854	End If	
1855	End Do	12
1856		
1857	End Subroutine ConvertMassToConc	3

Line Numb	Source	L2_RQSTS.MISS
1882	! iZ ::)	
1883	! ChemistryDefn :: A chemistry definition.	
1884	! ChemistryState :: State of a chemistry calculation.	
1885	! Locals:	
1886	Integer :: iSpecies ! Loop index.	
1887	Real(8) :: ConvFactor ! Conversion factor (molecules/cm3 -> moles).	
1888	Real(8) :: ConvFactorForOzone ! Modified conversion factor for O3 (molecules/cm3 -> moles).	
1889		
1890	ConvFactor = ChemistryState%Volume(iX, iY, iZ) / Dble(Avogadro)	15
1891	ConvFactorForOzone = ChemistryState%Volume(iX, iY, 1) / Dble(Avogadro)	3
1892		
1893	! Concentrations are in molecules/cm3 - need to convert back to g.	
1894	! Mass is set to zero when conc < 2.0. ## this test and the limit value perhaps could benefit from furthe	
1895	! consideration.	
1896	Do iSpecies = 1, ChemistryDefn%nSpecies	1
1897	If (Conc(iSpecies) > 2.0) Then	8
1898	If (iSpecies == iSpeciesO3 .and. ChemOpts%OzoneMethod == O3_PlumeParticles) Then	3
1899	ChemistryState%Mass(iSpecies, iZ, iY, iX) = &	
1900	Conc(iSpecies) * ConvFactorForOzone * ChemistryDefn%SpeciesMolWeights(iSpecies)	
1901	Else	
1902	ChemistryState%Mass(iSpecies, iZ, iY, iX) = &	7
1903	Conc(iSpecies) * ConvFactor * ChemistryDefn%SpeciesMolWeights(iSpecies)	
1904	End If	
1905	Else	
1906	ChemistryState%Mass(iSpecies, iZ, iY, iX) = 0.0	1
1907	End If	
1908	End Do	4
1909		
1910	End Subroutine ConvertConcToMass	3
1911		

Figure 4.4: L2 cache misses in the subroutines `ConvertMassToConc()` and `ConvertConcToMass()` for the run in Fig. 4.5 after restructuring the array layout in the structure `Type(ChemistryState_)`. These results have to be compared to those in Fig. 4.3, the number of cache misses in the highlighted lines is significantly reduced.



Name	L2_RQSTS.MISS samples	CPU_CLK_UNHALTED.THREAD samples	INST_RETIRED.ANY samples	Clocks per Instructions Retired - CPI
gridanddomainmodule_mp_interpxyzt_	1,231	11,891	16,819	0.707
chemistrymodule_mp_initchemistrygridboxes_	782	3,894	5,158	0.755
chemistrymodule_mp_updateparticlemasses_.	403	26,980	43,720	0.617
gridanddomainmodule_mp_interpxyt_	399	8,797	14,245	0.618
L_chemistrymodule_mp_chemistryupdate__...	304	2,175	2,189	0.994
chemistrymodule_mp_mainchemistry_.	241	29,718	39,232	0.757
nwpflowmodule_mp_cloudinfofrommet_.	231	2,810	2,508	1.120
chemistrymodule_mp_splitmassiveparticles_	224	998	829	1.204
chemistrymodule_mp_convertmasstoconc_.	125	1,000	979	1.021
particlemodule_mp_drydeposition_	124	9,568	10,538	0.908
particlemodule_mp_particleactive_	122	1,008	389	2.591
casemodule_mp_calcparticleresults_.	118	22,872	42,227	0.542
flowsmodule_mp_whichflow_.	108	2,869	4,390	0.654
timemodule_mp_addshorttime_.	97	7,599	10,374	0.733
coordinatesystemmodule_mp_x2position_	96	1,477	2,803	0.527
casemodule_mp_oneparticletimestep_.	93	4,969	4,543	1.094
timemodule_mp_char2time_.	85	1,616	1,576	1.025
particlemodule_mp_particleconc_	84	557,616	827,979	0.673
[illegible]	76	4,410	4,450	0.991

Figure 4.5: Hardware counters for subroutine calls in a `air_quality_test_short.txt` run *after* changing the array layout in the structure `Type(ChemistryState_)`.

# threads	time [before]	time [after]	gain	speedup [before]	speedup [after]
1	110.820 s	101.120 s	9.6%	—	—
2	73.190 s	70.110 s	4.4%	1.51	1.44
3	65.800 s	64.470 s	2.1%	1.68	1.57
4	67.510 s	62.000 s	8.9%	1.64	1.63
5	69.040 s	61.340 s	12.6%	1.61	1.65
6	73.030 s	65.160 s	12.1%	1.52	1.55
7	80.170 s	68.810 s	16.5%	1.38	1.47
8	88.340 s	73.510 s	20.2%	1.25	1.38

Table 4.3: Total time spent in the parallelised chemistry loop before and after changing the array layout in `Type(ChemistryState_)` from (x, y, z, j) to (j, z, y, x). In the third column we show the performance gain (defined as $\frac{\text{time[before]}}{\text{time[after]}} - 1$), in the last two columns we show the self-relative speedup in both cases.

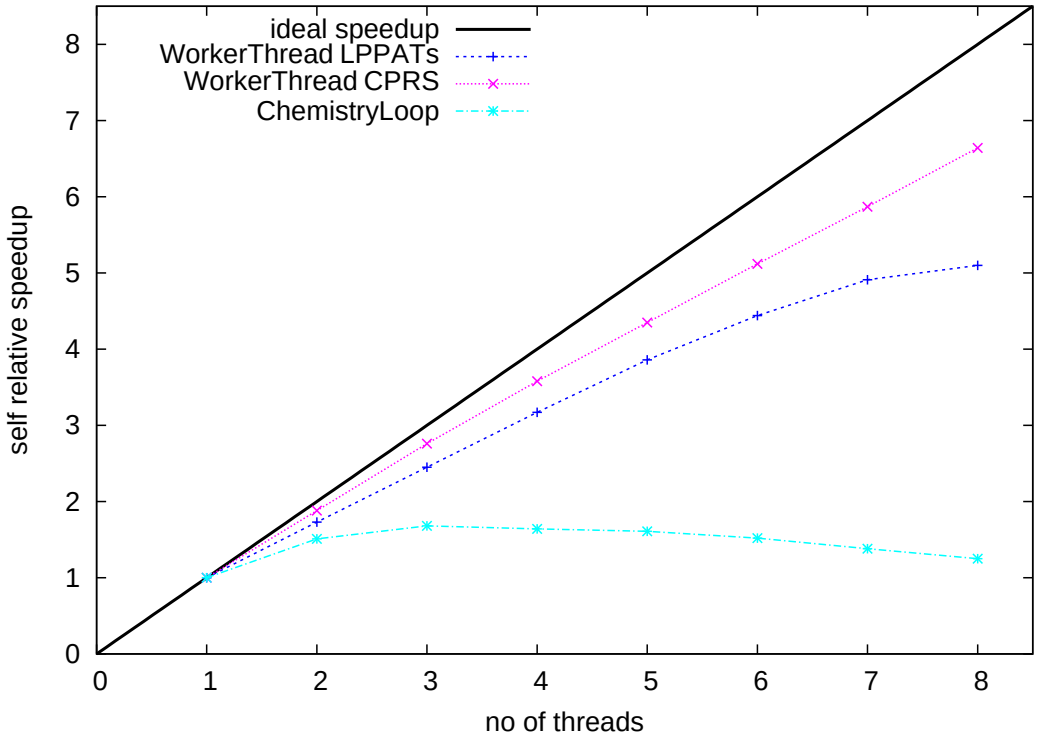


Figure 4.6: Speedup of parallel loops *before* restructuring arrays in `ChemistryState_`
.

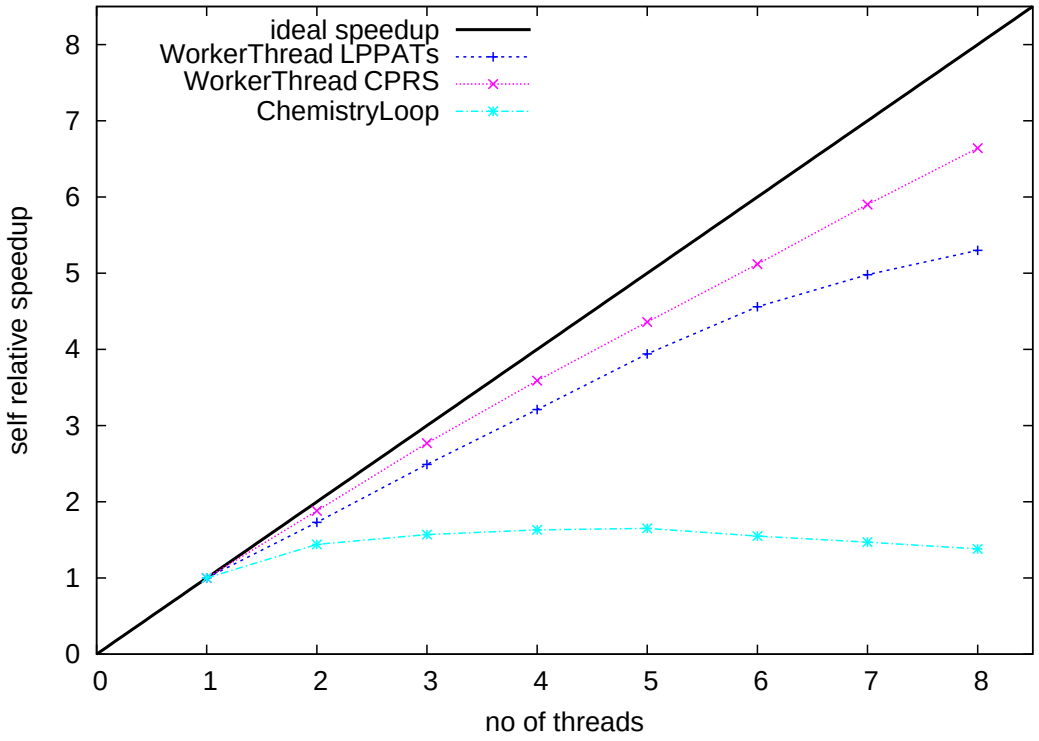


Figure 4.7: Speedup of parallel loops *after* restructuring arrays in `ChemistryState_`
.

benchmark	LPPATs	CPRs	ChemistryUpdate	total
coarse	15%	26%	58%	253 m
hires	11%	5%	71%	669 m

Table 4.4: Total runtime and relative time spent in the parallelised sections of the code for a 4 thread run of the new AQ benchmark before restructuring of the array layout in `Type(ChemistryState_)`

# threads	coarse	hires
1	674 m	1278 m
2	382 m	827 m
4	247 m	638 m
6	209 m	642 m

Table 4.5: Total runtime for the new air quality benchmark. All results are obtained with the new array layout in `Type(ChemistryState_)`.

Air quality benchmarks

We tested the impact of changes in array layout on the air quality benchmarks described in section 3.1.1. Since the runs reported there, the parameters of these benchmarks have been adjusted (see [3]), so that the runtimes in this section can not be compared directly to those in section 3.1.1. However, a reference run on 4 cores was performed with the new set of parameters. In Tab. 4.4 we show the total runtime for this run and the relative amount of time spent in the parallelised sections of the code. All runs were performed with the scheduling strategy `"dynamic,16"` for the chemistry loop, we also repeated the 4 thread run with `"static,1"` but find that this has no impact on the runtime. With the new set of parameters the chemistry loop takes up a significant amount of time in both benchmarks. It should be kept in mind, however, that this is a 4 core run, i.e. the relative proportion of time spent in the chemistry loop in a one core run is likely to be smaller.

Tab. 4.5 lists the new timing results after changing the layout of the arrays in the structure `Type(ChemistryState_)`. Comparing to the 4 core runs (last column in Tab. 4.4), the total increase in performance is 2.4% for the coarse run and 4.6% for the hires benchmark. In Tab. 4.6 we also list more detailled timing results for the hires benchmark. We give results for the entire subroutine `ChemistryUpdate()` and for the parallelised Chemistry loop alone, the latter data was not available for the run before array restructuring. From these results we conclude that restructuring the arrays in `Type(ChemistryState_)` leads to a speedup of 8% of the subroutine `ChemistryUpdate()`. This has to be compared to the results obtained with the simplified air quality benchmark (`air_quality_test_short.nif`): according to Tab. 4.3 the chemistry loop is 8.9% faster with array restructuring.

Total speedup curves for both benchmarks are shown in Fig. 4.8. As chemistry processing takes up a significant of time, even in the coarse benchmark, scaling is poor in both cases.

Fig. 4.9 shows speedup for different parallelised sections of the code on the hires benchmark. As in previous runs the bad overall scaling can be traced back to the chemistry loop. Even though restructuring the array layout has reduced the overall runtime it has not resolved the scaling issue and further efforts should concentrate on the chemistry loop.

4.1.2 Scheduling strategies

Tab. 4.7 shows timing results for the short air quality testrun `air_quality_test_short.nif` using different scheduling strategies. All runs were carried out on `els034` with 4 processors. It should be noted that these values are likely to be dependent on the problem. If, for example, the chemistry grid is well balanced over the gridboxes, the scheduling strategy of the chemistry loop will only have a small influence on the runtime.

# threads	LPPATs	CPRs	Chemistry Loop	ChemistryUpdate	total
1	242 m	104 m	794 m	842 m	1278 m
2	137 m	65 m	487 m	535 m	827 m
4	73 m	39 m	388 m	435 m	638 m
4⋆	74 m	32 m	—	473 m	669 m
6	52 m	28 m	423 m	471 m	642 m

Table 4.6: Total runtime and relative time spent in the parallelised sections of the code for a 4 thread run of the new hires AQ benchmark. The run 4⋆ is a reference run on 4 cores before restructuring the arrays in `Type(ChemistryState_)`, all other runs were performed after array restructuring

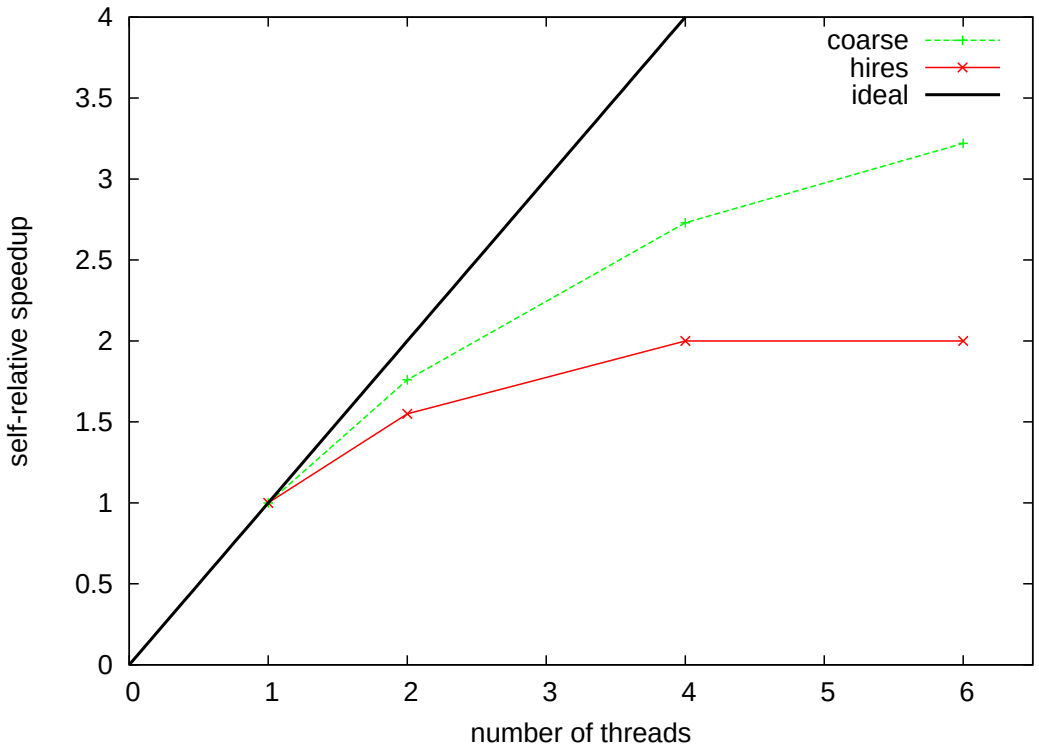


Figure 4.8: Self-relative speedup of the coarse and hires benchmark after reordering the structures in `Type(ChemistryState_)`.

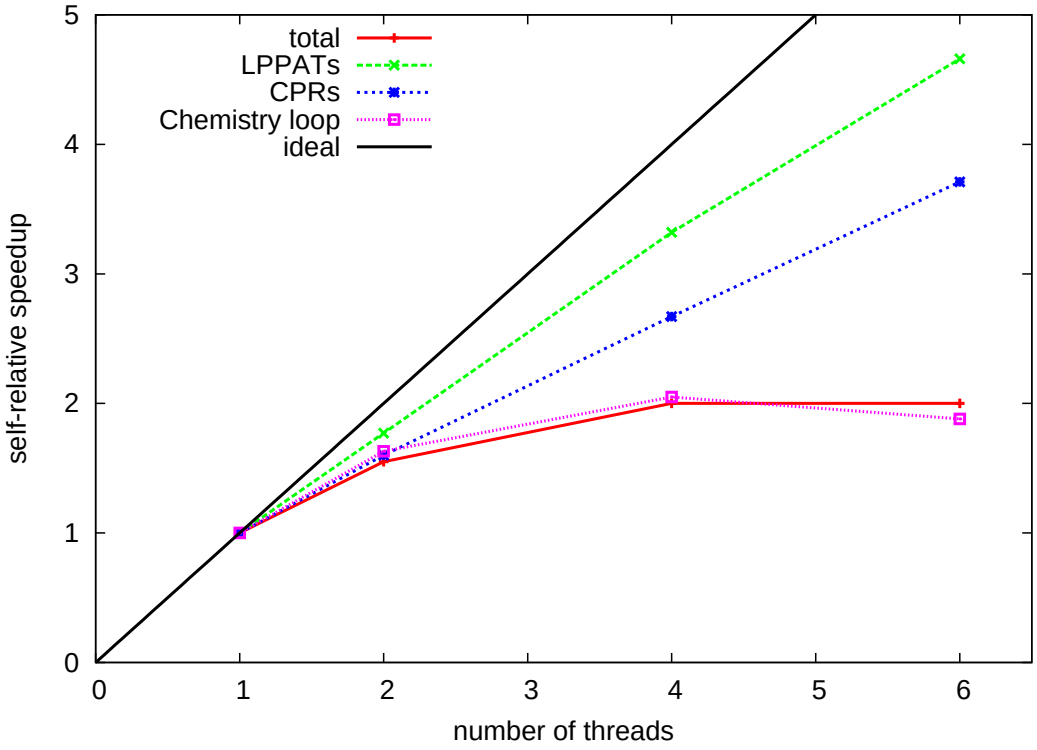


Figure 4.9: Self-relative speedup of different parallel sections of the code for the hires benchmark after reordering the structures in `Type(ChemistryState_)`. Overall scaling is still poor due to the dominance of the chemistry loop in the runtime.

	LPPATs	CPRs	ChemistryLoop
static,1	21.58 s	149.33 s	61.94 s
static,4	20.81 s	149.11 s	68.13 s
static,16	20.28 s	149.05 s	65.07 s
static,32	19.96 s	149.03 s	76.91 s

	LPPATs	CPRs	ChemistryLoop
dynamic,1	25.07 s	148.58 s	63.03 s
dynamic,4	21.84 s	148.14 s	62.96 s
dynamic,16	20.52 s	148.03 s	65.04 s
dynamic,32	20.17 s	147.63 s	75.74 s

	LPPATs	CPRs	ChemistryLoop
guided,1	19.94 s	147.58 s	68.07 s
guided,4	20.17 s	147.64 s	66.63 s
guided,16	20.07 s	147.73 s	63.55 s
guided,32	19.95 s	147.44 s	76.87 s

Table 4.7: Timing results of the three parallel loops in the code for different scheduling strategies

For the "particle" loop the strategy does not seem to be important but the time spent on this loop can be slightly reduced by using larger chunk sizes. We decided to keep the value of `"static,16"` which was reported to be good in [4].

For the "particle update" loop the dependence on the chunksize is very weak for all three strategies. `"guided"` seems to be the optimal strategy but the difference to `"dynamic"` is extremely small. We will keep the hardwired value of `"dynamic,16"` in the code.

The performance of the "chemistry" loop get significantly worse for chunksizes larger than 16. The optimal strategy seems to be `"static,1"` but we will leave the strategy to be specified at runtime.

4.1.3 Hyperthreading

On a multicore processor each physical core consists of a collection of specialised compute units to carry out different tasks such as floating point- and integer arithmetic. Usually each core can only execute one thread at a time. Threads can be switched between cores in rapid succession, which gives the illusion of parallel execution but each switch requires the thread specific memory to be overwritten. The hyperthreading technology allows to run multiple threads simultaneously on one physical core. For this the elements which store the state of a thread, such as registers, are duplicated and multiple threads can use different compute resources simultaneously, for example one thread might perform floating point operations while another uses the integer arithmetic unit.

Fig. 4.10 shows the layout of a typical system, in which each node consists of two CPUs, and each processor contains two cores, which in turn provide two logical cores each. When labelling logical cores, we use a cenvention where the first index refers to the physical core and the second to the logical subcore.

Does my CPU support hyperthreading? To find out whether a specific machine supports hyperthreading, look at the processor information in `/proc/cpuinfo`. CPUs that support hyperthreading have the flag `ht` in `/proc/cpuinfo` set, e.g.

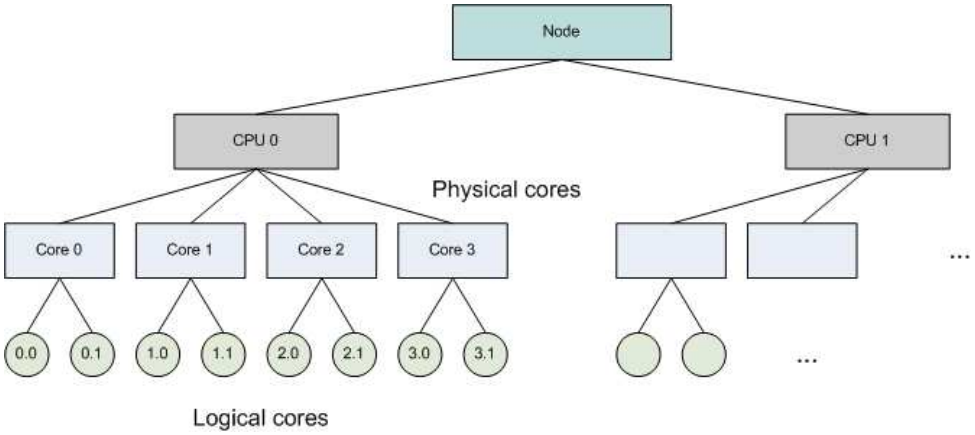


Figure 4.10: Processor layout

```
apem@eld497:> grep flags /proc/cpuinfo
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
                  pge mca cmovpat pse36 clflush dts acpi mmx fxsr
                  sse sse2 ss ht tm pbe nx rdtscp lm constant_tsc
                  pni monitor ds_cpl est tm2 xtpr popcnt
                  ...
```

Hyperthreading can be enabled in the BIOS (ask the Linux team by submitting a remedy request) and requires a Linux kernel which has been compiled with appropriate support. To check whether hyperthreading is enabled, print out the number of different core ids in `/proc/cpuinfo` and compare it to the total number of core ids:

```
apem@eld497:> # number of different core ids
apem@eld497:> grep 'core id' /proc/cpuinfo | sort | uniq | wc -l
4

apem@eld497:> # total number of core ids
apem@eld497:> grep 'core id' /proc/cpuinfo | sort | wc -l
8
```

Hyperthreading is enabled if the latter is larger than the first. The number of physical CPUs can be determined with

```
apem@eld497:> grep 'physical id' /proc/cpuinfo | sort | uniq | wc -l
1
```

The output above was obtained on the desktop machine eld497, which contains an Intel Xeon E5520 quad core processor. With hyperthreading enabled, each physical core supports is split into two logical cores. On contrast, the server X5680 contains two Intel Xeon X5680 processors, which contain 6 cores each, giving 24 logical cores in total.

KMP_AFFINITY no. of threads	none	scatter	compact
1	107.71 s	107.83 s	107.72 s
2	55.47 s	55.64 s	87.94 s
3	37.82 s	37.77 s	59.19 s
4	28.84 s	28.76 s	44.92 s
5	31.30 s	36.01 s	36.23 s
6	28.72 s	30.32 s	30.31 s
7	28.20 s	26.10 s	26.13 s
8	23.19 s	23.24 s	24.49 s

Table 4.8: Time spent in the main particle loop for different numbers of threads and values of `KMP_AFFINITY`.

Binding threads to individual cores

Usually, threads are not bound to individual cores and are moved around according to the available resources. Potentially, this can have an impact on the runtime as data that is stored locally on a specific core is lost whenever this happens. For code compiled with the Intel Fortran compiler, threads can be bound to individual cores by setting the environment variable `KMP_AFFINITY` (see [5], which also described a low level interface for thread binding).

Of particular interest are the following three settings for `KMP_AFFINITY`:

- `none`: Threads are not bound to individual cores
- `compact`: Threads are bound to logical cores such that successive threads are kept as close as possible, i.e. when distributing four threads between the logical cores in Fig. 4.10, the first two are assigned to physical core 0 (i.e. logical cores 0.0 and 0.1) and the last two to the next physical core (i.e. logical cores 1.0 and 1.1).
- `scatter`: In contrast to `compact` successive threads are kept as far apart as possible. When distributing 4 threads on a single CPU, each thread is assigned to a separate physical core. For a node with four CPUs, each thread would be assigned to the first logical core on the separate CPUs.

In addition, the keyword `verbose` can be added to print out information on thread binding, for example

```
export KMP_AFFINITY=verbose,scatter
```

Threads can also be explicitly bound to specific logical cores, but this is not discussed in this article.

Scaling

Individual run We investigate the scaling of the particle loop LPPATs in a simple testcase (`Example_SingleSite_Quick.nif`, see appendix 4.1.3 for the input file). Meteorological data is provided from the `SingleSite` flow module and the particles are propagated for 12 hours. The runtime is dominated by the particle loop (over 95% of the runtime). All runs were performed on the desktop `eld497`, which has an Intel Xeon E5520 quadcore processor and can be used to run up to eight threads simultaneously if hyperthreading is enabled.

In Tab. 4.8 we list the time spent in this loop for runs with three different settings of `KMP_AFFINITY`. For the `scatter` run the logical cores are filled in the order $0.0 \to 1.0 \to 2.0 \to 3.0 \to 0.1 \to 1.1 \to 2.1 \to 3.1$ whereas for the `compact` run they are filled in the order $0.0 \to 0.1 \to 1.0 \to 1.1 \to 2.0 \to 2.1 \to 3.0 \to 3.1$. For 7 and 8 threads the distribution between different logical cores is identical in the `scatter` and `compact`

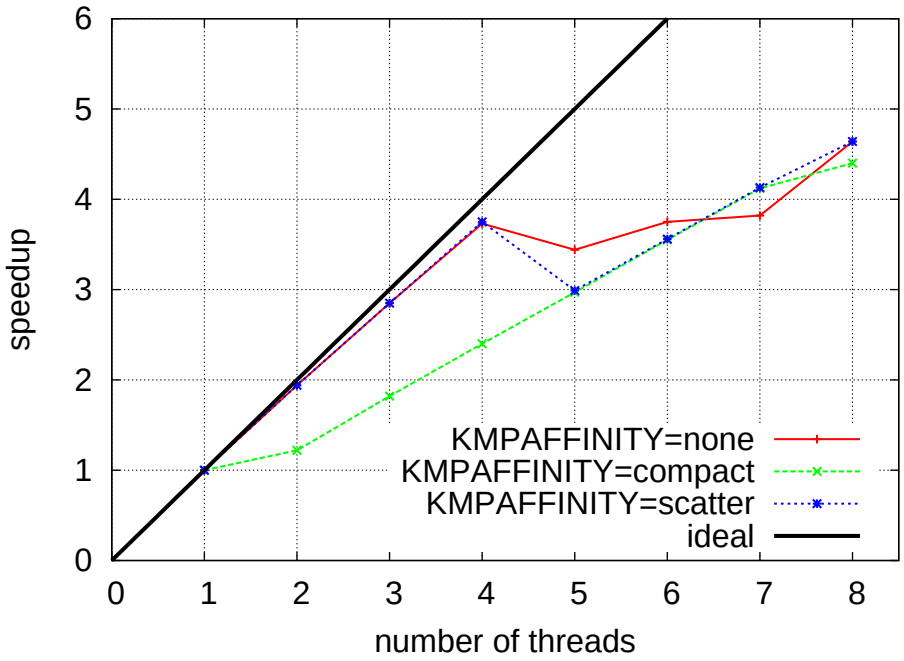


Figure 4.11: Scaling of the particle loop. We show the self relative speedup as a function of the number of threads for different values of `KMP_AFFINITY`.

runs. The timing results were obtained on a quiet machine, only a single terminal window was open. If other applications, such as firefox or an email client are run at the same time, the performance decreases significantly for the 4 and 8 thread runs.

The self relative speedup is shown in Fig. 4.11.

We also increased the number of particles released into the atmosphere by a factor of 10. Tab. 4.9 and Fig. 4.12 show that this increases the runtime by one order of magnitude but has no impact on the scalability.

As can be seen from these results, for up to four threads it it much better to scatter the threads as wide apart as possible and this seems to be the default strategy is `KMP_AFFINITY` is set to `none`. This result is not very surprising as running two threads on the same physical core will be slower than running them on two separate physical cores. If more threads than physical cores are started, the runtime of the `scatter`

`KMP_AFFINITY` no. of threads	none	scatter	compact
1	1076.53 s	1075.40 s	1077.71 s
2	556.08 s	557.36 s	878.83 s
3	377.30 s	377.59 s	591.37 s
4	287.35 s	287.12 s	447.61 s
5	311.96 s	360.15 s	361.40 s
6	267.44 s	303.36 s	302.21 s
7	252.58 s	260.36 s	260.48 s
8	229.57 s	230.01 s	229.47 s

Table 4.9: Time spent in the main particle loop for different numbers of threads and values of `KMP_AFFINITY`. Compared to Tab. 4.8 the number of particles released into the atmosphere was increased by a factor of 10.

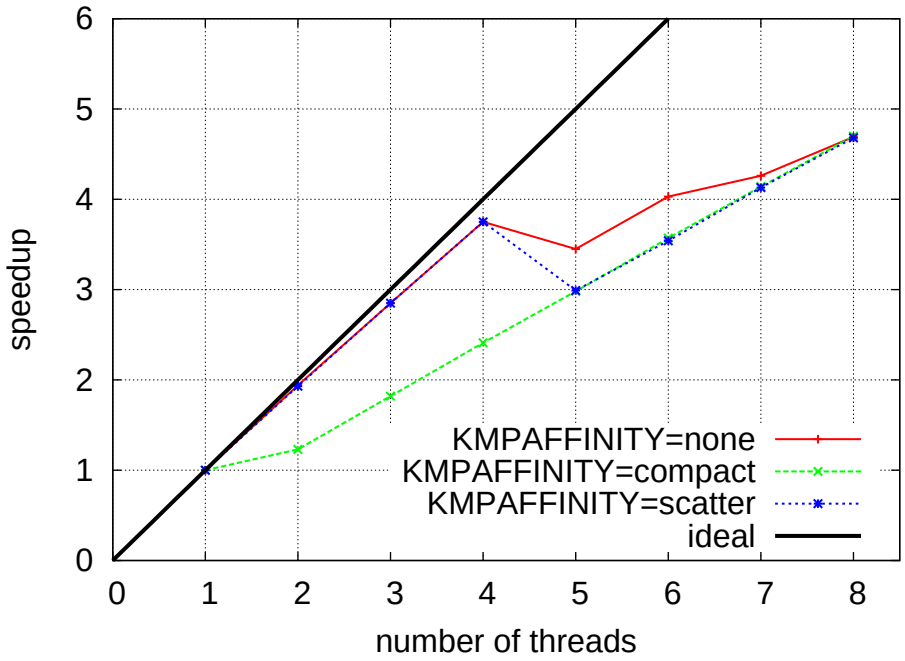


Figure 4.12: Scaling of the particle loop. We show the self relative speedup as a function of the number of threads for different values of `KMP_AFFINITY`. Compared to 4.11 the number of particles released into the atmosphere was increased by a factor of 10.

and `compact` runs is identical, and they both scale at the reduced rate, observed for the `compact` runs with four or less threads. Not binding the threads seems to reduce the runtime in this case, except on 7 cores. This could, however, be explained by background processes which might disturb the timing results. For the longer run we find that not binding the threads to individual cores gives consistently better results. We conclude that, when running a single NAME job on a quiet machine, the best results are obtained by not binding threads to cores or by scattering them as wide as possible. Using more threads than physical cores reduces the runtime further, but the scalability is reduced significantly.

Simultaneous runs Next, we ran two identical copies of the code simultaneosly on the same machine. The number of threads requested for each individual run was varied between 1 and 4. The total time spent in the main particle loop is shown in Tab. 4.10. It is important to note that when binding threads to cores, this will be done independently for the two processes. For example, using `KMP_AFFINITY=compact` and running with two threads, each process will assign the two threads to the same physical core and this physical core will run four threads whereas the remaining three cores are left idle. This is clearly not desirable. Again, the best results are obtained by not binding threads to individual cores. For the 4 thread run, however, using `KMP_AFFINITY=none` and `KMP_AFFINITY=scatter` gives the same results because each physical core runs two separate instances of the code which leads to the best load balancing as it is likely that at a given point in time the two processes use different compute units on the core.

It is also instructive to compare the runtime of two simultaneous four thread runs to that of two consecutive runs on 8 cores. As can be seen from Tab. 4.10 the first takes around 45 s whereas the latter would take about the same time, namely $2 \times 23\,\mathrm{s} = 46\,\mathrm{s}$ (see the last row in Tab. 4.8).

Input file

For future reference, we show the input file used for the runs in this report in Fig. 14:

KMP_AFFINITY no. of threads	none	scatter	compact
1	107.63 s / 107.92 s	176.25 s / 175.72 s	176.44 s / 175.66 s
2	55.78 s / 55.57 s	89.34 s / 89.33 s	178.18 s / 183.06 s
3	53.77 s / 53.82 s	59.96 s / 59.93 s	130.26 s / 129.16 s
4	45.30 s / 45.20 s	45.32 s / 45.37 s	87.74 s / 85.96 s

Table 4.10: Timing results for two simultaneous NAME runs on the same machine. In each case we show the time spent in the main particle loop for both processes.

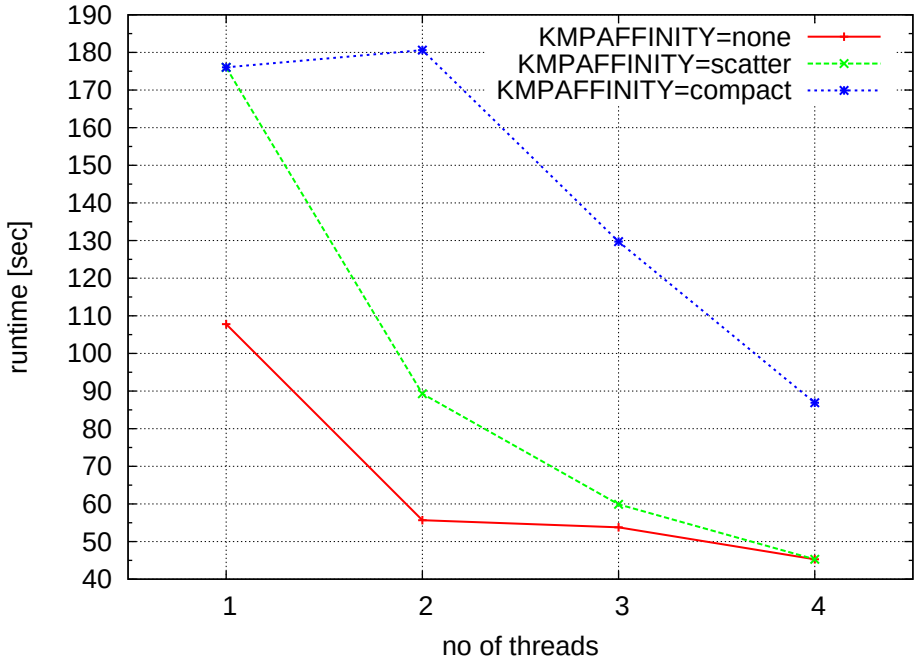


Figure 4.13: Runtime for two simultaneous NAME runs. We show the average runtime for the two processes, which is very close to the runtime of each individual process (see Tab. 4.12).

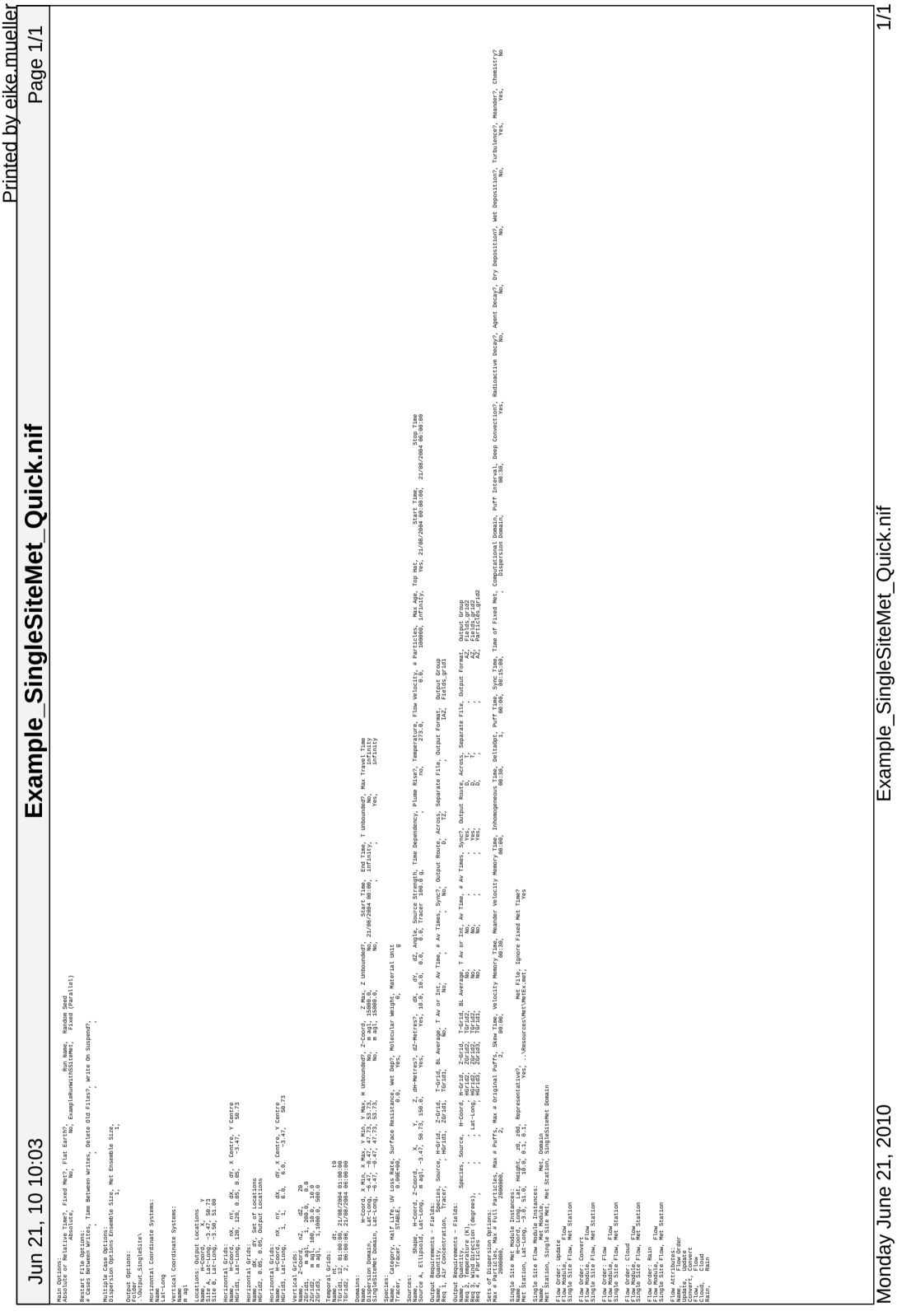


Figure 14: Input file used for NAME runs in section 4.1.3

Bibliography

[1] http://software.intel.com/en-us/articles/using-intel-vtune-performance-analyzer-events-ratios-optimizing-applications/

[2] Performance Application Programming Interface http://icl.cs.utk.edu/papi/

[3] Lucy S. Davis *Potential improvement areas in* NAME *air quality through using parallelised code* Met Office internal note

[4] Graham Riley and Rupert Ford: *NameIII Parallelisation Study for the MetOffice - Stage 2 Final report, v1.0* 4 March 2010

[5] Documentation of KMP_AFFINITY on http://www.ncsa.illinois.edu/UserInfo/Resources/Software/Intel/Compilers/10.0old/main_cls/mergedProjects/optaps_cls/common/optaps_openmp_thread_affinity.htm

Summary and outlook

In this document we described the parallelisation of the current development version on NAME using OpenMP. The performance of the parallel code was tested for two realistic air quality benchmarks and found to give a significant speedup when run on multiple cores. The poor scaling of the chemistry loop was identified as the main weakness of the current implementation. This is likely due to memory access bottlenecks. In an exploratory study with Intel vTune we showed that this is at least partially due to bad cache utilisation. vTune allows to identify subroutines and lines in the source code which are responsible for this behaviour. By restructuring some of the arrays in the main chemistry data structure `Type(ChemistryState_)` we managed to speed up the chemistry loop by around 10%. This applies both to the serial and the parallel version. Most likely there is significant potential for further improvement as we have not yet analysed the two most time consuming subroutines called from inside the chemistry loop.

We find that vTune is very valuable tool for performance analysis, partially because it is very easy to use and does not require any modifications to the source code. We feel that during the four week test period, for which the trial version was installed, we have only used a small fraction of its functionality. It is, however, worth considering other tools such as the freely available papi library.

Two of the new servers have CPUs with eight cores which support hyperthreading, which effectively doubles the number of available cores to 16. Running the parallel code on these machines will allow for further insights into the scalability of the code.

Most of the work reported here was carried out on the Linux system. We have also compiled and run the code with OpenMP support under the Visual Studio environment on windows and plan to adapt the Makefile for cross platform compatibility. No detailed performance and scaling analysis has been carried out on the windows platform so far.

Appendix A

CPU Caches

Following the huge increase in clockspeed of modern processors over the last years, the main performance bottleneck is now the rate with which data can be read from main memory. In particular this seems to be the case for the chemistry loop in the NAME code. In modern computer architectures data is not read directly from main memory but stored in intermediate *cache* memory which can be accessed much faster by the CPU. As memory access is often local in time, i.e. subsequent lines of code are likely to operate on the same data, this reduces the number of main memory accesses.

In the following we give a brief overview of the most important concepts related to caches, for more details see, for example, [1, 2, 3].

A.1 Cache structure

Caches are usually organised in levels with a small. The most common layout consists of a small fast cache close to the CPU (usually called direct cache or L1 cache) and a slower, larger L2 cache between the L1 cache and main memory (see Fig. A.1). The L1 cache is usually split into a data cache and an instruction cache. In multicore systems. L1 caches are often local to a particular core.

A.2 Cache lines

Cache memory is organised in rows, which contain the data (the *cache line*, which typically has a length of 16 to 128 bytes) and its address in main memory. As most programs operate on data which is contiguous in memory (this is particularly true for scientific code which typically processes large arrays), loading entire cache lines instead if individual bytes reduces the number of main memory accesses significantly.

A.3 Associativity

There are several ways of deciding where to store data from main memory in a cache.

If the cache is fully associative, each datum in memory can be mapped to an arbitrary cache row. This reduces the number of cache misses but has the disadvantage is that all cache rows have to be searched when deciding whether data for a particular memory address is already available in the cache.

The other extreme would be a directly mapped cache, in which each memory address is mapped to exactly one cache row. In set associative caches the memory address is split into three parts: If the size of main memory is 2^M, the cache has 2^R rows and the size of a cacheline is 2^L, the L least significant bits are stored as the *offset* of this data. The next R bits determine the cache row the data is stored in

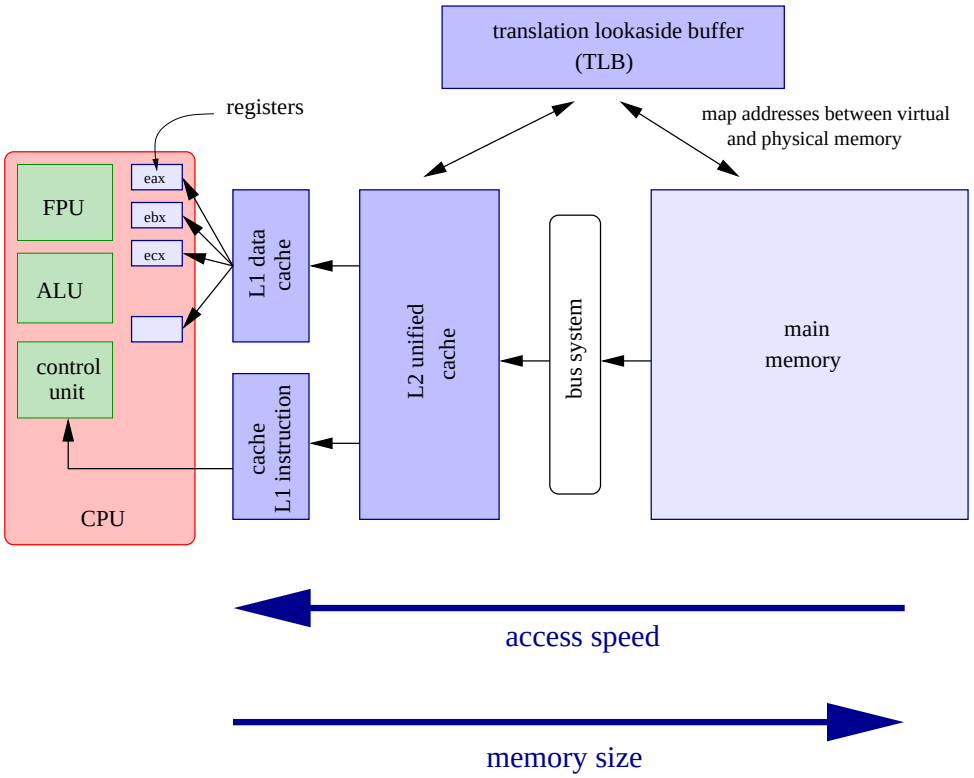


Figure A.1: Multilevel cache system of a modern processor

(the *index*). The remaining $M - R - L$ bits are stored as the *tag*. To determine whether a requested datum is held in cache, the CPU splits its address into offset o, index i and tag t and compares the tag stored in cache row i with t. If they agree, the data is stored in the cache line of this row, otherwise it has to be fetched from main memory. Obviously this is much faster than scanning all cache rows in a fully associative cache but has the disadvantage of an increased number of cache misses (see [2] for an example). An n-set associative cache allows to have n cache rows with the same index. This reduces the number of cache misses as a given memory address can be stored in n different cache rows.

Note that addressing data is complicated by the fact that most machines address memory indirectly by allowing each process to have its own, contiguous block of virtual memory which is then mapped to the physical memory addresses. These mappings can be stored in a separate cache, the translation lookaside buffer (TLB).

A.4 Replacement strategy

When fetching new data from main memory the cache has to decide which old data to overwrite. There are several strategies for this, for example it could decide to overwrite the oldest data in cache, this policy is known as *least recently used* (LRU). Note that a directly mapped cache does not need a replacement strategy as each address in main memory maps to exactly once cache row which will be evicted when new data is read from main memory. Obviously programs with a regular and predictable memory access pattern, such as many scientific codes, will have a better cache performance than applications with random data access.

A.5 Write back strategy

Most of the time data is read from cache and writes are quite rare (they do not occur at all for the instruction cache). However, whenever data is written to the cache, main memory has to be updated at some point. In write-through caches this happens whenever the data in the cache is written whereas in copy-back (or write-back) caches data is only written back to main memory when it is evicted from the cache. This requires additional housekeeping and an additional *dirty* flag is associated with each cache row to indicate that it differs from what is stored in main memory. This is particularly important if different cores share data: If core 1 reads data from memory but core 2 has modified this data in its cache, core 2 has to ensure that it is written back to main memory before it can be used by core 1. In OpenMP the `!$OMP ATOMIC` directive can be used to protect memory accesses to avoid this problem.

Bibliography

[1] Ulrich Drepper `http://lwn.net/Articles/252125/`

[2] Ruud van der Pas *Memory Hierarchy in Cache-Based Systems*, 2002, Sun Microsystems

[3] Paul Genua, P.E. *A Cache Primer*, 2004, Freescale Semiconductor

[4] Suely Oliveira and David Steward *Writing Scientific Software - a Guide to Good Style* Cambridge University Press 2006

Appendix B

Input files

This appendix lists the input files for the testruns described in this document. Due to the small fontsize they are probably not readable in the printed version but might be a useful reference when viewing the document with sufficient magnification on a computer screen.

Printed by eike.mueller

Mar 11, 10 17:27 **Example_EMARC.nif** **Page 1/1**

```
Main Options:
Absolute or Relative Time?, Fixed Met?, Flat Earth?,          Run Name,  Random Seed
                  Absolute,        No,          No,  EMARC_Test01,       Fixed (Parallel)

Restart File Options:
# Cases Between Writes, Time Between Writes, Delete Old Files?,
                      ,                    ,                  ,

Multiple Case Options:
Dispersion Options Ensemble Size, Met Ensemble Size,
                               1,                  1,

Output Options:
Folder
./Output_EMARC/

Input Files:
File names
../Resources/Defns/MetDefnUM6G.txt

../Resources/Defns/MetDefnUM5MDt1.txt

Horizontal Coordinate Systems:
Name
Lat-Long

Vertical Coordinate Systems:
Name
m agl

Horizontal Grids:
Name,           H-Coord,   nX,    nY,    dx,    dy,       X Centre,     Y Centre
OutPut_HGrid1, Lat-Long,  180, 150,  0.4,  0.3,  -1.0,   53.0

Vertical Grids:
Name,           Z-Coord,  nZ,     dZ,    Z0
OutPut_ZGrid1,   m agl,   4, 5000.0, 2500.0

Temporal Grids:
Name,    nt,          dt,                t0
TGrid1,  6,     3:00:00,  01/04/2008 04:00:00

TGrid1,  20,     6:00:00,  01/04/2008 00:00:00

Domains:
Name,                H Unbounded?,  H-Coord,     X Min,     X Max,  Y Min,  Y Max, Z Unbounded?, Z-Coord,   Z Max, T Unbounded?,          Start Time,  End Time, Max Travel Time
Dispersion Domain,             No, Lat-Long,     -44.2,      42.2,   26.0,   80.0,           No,   m agl, 20000.0,           No, 01/01/2005 00:00:00,  infinity,        infinity

Locations: Source Locations
Name,           H-Coord,      X,          Y
SomeWhere,      Lat-Long,   -1.0,   53.0

Species:
Name,                 Category,        Half Life, UV Loss Rate, Surface Resistance, Deposition Velocity, Wet Type, Molecular Weight, Material Unit
CAESIUM-137,       RADIONUCLIDE,         9.46E+08,     0.00E+00,           2.00E+03,                   ,        1,                0,            Bq

Sources:
Name,      Set of Locations,           Location,       Shape,  H-Coord, Z-Coord, dH-Metres?, dZ-Metres?,    dX,    dY,      Z,     dZ,  Angle,         Source Strength, Time Dependency, Plume Rise?, Temperature, Flow Velocity,  # Particles,  Max Age, Top Hat,           Start Time,      Stop Time
Source A,  Source Locations,          SomeWhere,   Ellipsoid, Lat-Long,   m agl,        Yes,        Yes,  50.0,  50.0,  250.0,  500.0,    0.0, CAESIUM-137   1.0E12 Bq,                ,          No,       273.0,           0.0,       200000, infinity,     Yes,  01/04/2008 01:00:00,  01/04/2008 04:00:00

Source A,  Source Locations,          SomeWhere,   Ellipsoid, Lat-Long,   m agl,        Yes,        Yes,  50.0,  50.0,  250.0,  500.0,    0.0, CAESIUM-137   1.0E12 Bq,                ,          No,       273.0,           0.0,       120000, infinity,     Yes,  01/04/2008 00:00:00,  06/04/2008 00:00:00

Output Requirements - Fields:
Quantity,                Species,   Source,        H-Grid,        Z-Grid,  T-Grid, BL Average, T Av or Int, Av Time,  # Av Times, Sync?, Graph?, Screen?, Disk?,  Across, Separate File,  Output Format,  Output Group
Air Concentration,    CAESIUM-137,         , OutPut_HGrid1,              ,  TGrid1,        Yes,          Av,    6:00,          36,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1
Air Concentration,    CAESIUM-137,         , OutPut_HGrid1, OutPut_ZGrid1,  TGrid1,           ,          Av,    6:00,          36,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1
Air Concentration,    CAESIUM-137,         , OutPut_HGrid1,              ,  TGrid1,        Yes,         Int,   24:00,         144,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1
Dry deposition rate,  CAESIUM-137,         , OutPut_HGrid1,              ,  TGrid1,           ,         Int,   24:00,         144,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1
Wet deposition rate,  CAESIUM-137,         , OutPut_HGrid1,              ,  TGrid1,           ,         Int,   24:00,         144,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1
Deposition Rate,      CAESIUM-137,         , OutPut_HGrid1,              ,  TGrid1,           ,         Int,   24:00,         144,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1

Air Concentration,    CAESIUM-137,         , OutPut_HGrid1,              ,  TGrid1,        Yes,          Av,    6:00,          36,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1
Air Concentration,    CAESIUM-137,         , OutPut_HGrid1, OutPut_ZGrid1,  TGrid1,           ,          Av,    6:00,          36,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1
Air Concentration,    CAESIUM-137,         , OutPut_HGrid1,              ,  TGrid1,        Yes,         Int,  120:00,         720,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1
Dry deposition rate,  CAESIUM-137,         , OutPut_HGrid1,              ,  TGrid1,           ,         Int,  120:00,         720,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1
Wet deposition rate,  CAESIUM-137,         , OutPut_HGrid1,              ,  TGrid1,           ,         Int,  120:00,         720,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1
Deposition Rate,      CAESIUM-137,         , OutPut_HGrid1,              ,  TGrid1,           ,         Int,  120:00,         720,    No,     No,      No,   Yes,      TZ,             T,           IA2,  Fields_grid1


Sets of Dispersion Options:
Max # Particles, Max # Full Particles, Max # Puffs, Max # Original Puffs, Skew Time, Velocity Memory Time, Meander Velocity Memory Time, Inhomogeneous Time, DeltaOpt, Puff Time, Sync Time, Time of Fixed Met, Computational Domain, Puff Interval, Deep Convection?, Radioactive Decay?, Agent Decay?, Dry Deposition?, Wet Deposition?, Turbulence?, Meander?, Chemistry?
        4100000,               200000,          10,                   10,     00:00,                00:00,                        00:00,              00:00,        1,     00:00,  00:07:30,                  ,     Dispersion Domain,         00:30,             Yes,                Yes,           No,             Yes,             Yes,         Yes,      Yes,         No

NWP Met Module Instances:
Met Definition Name,     Name,   Min B L Depth, Max B L Depth, Use NWP BL Depth?, Restore Met Script, Delete Met?,                        Met Folder,          Topography Folder
UM6G,                Met_data,            50.0,        4000.0,               Yes,                   ,          No,       /data/local/apem/ParallelMet,   /home/h03/apdg/NameIII/NameIIILibrary/Data/UMTopogData

UM5M,            Mes_Met_data,            50.0,        4000.0,               Yes,                   ,          No,       /data/local/apem/ParallelMet,   /home/h03/apdg/NameIII/NameIIILibrary/Data/UMTopogData

UM6G,                Met_data,            50.0,        4000.0,               Yes,                   ,          No,       ../Resources/UMMet,   ../Resources/UMTopogData
UM5M,            Mes_Met_data,            50.0,        4000.0,               Yes,                   ,          No,       ../Resources/UMMet,   ../Resources/UMTopogData

NWP Flow Module Instances:
Name,                     Met Module,          Met,   Domain
Glo_Met_data_flow,           NWP Met,     Met_data,   Dispersion Domain

Mes_Met_data_flow,           NWP Met,  Mes_Met_data,   Dispersion Domain

Flow Order: Update
Flow Module, Flow
NWP Flow,    Glo_Met_data_flow

NWP Flow,    Mes_Met_data_flow

Flow Order: Convert
Flow Module, Flow
NWP Flow,    Glo_Met_data_flow

NWP Flow,    Mes_Met_data_flow

Flow Order: Flow
Flow Module, Flow
NWP Flow,    Glo_Met_data_flow

NWP Flow,    Mes_Met_data_flow

Flow Order: Cloud
Flow Module, Flow
NWP Flow,    Glo_Met_data_flow

NWP Flow,    Mes_Met_data_flow

Flow Order: Rain
Flow Module, Flow
NWP Flow,    Glo_Met_data_flow

NWP Flow,    Mes_Met_data_flow

Flow Attributes:
Name,    Flow Order
Update,  Update
Convert, Convert
Flow,    Flow
Cloud,   Cloud
Rain,    Rain
```

Monday March 29, 2010 Example_EMARC.nif 1/1

Figure B.1: Input file for the EMARC benchmark described in section 2.3.4.

Printed by eike.mueller

Mar 19, 10 10:17 **air_quality_test_short.nif** Page 1/1

```
------------------------------------------------------------
! Project: NAQF
! File: NAME III AQ run with uk and emep 2003 data
! Author: Claire Witham
! Date: 8/04/2007
! For NAME III version: 5.0c
------------------------------------------------------------

Main Options:
Absolute or Relative Time?, Fixed Met?, Flat Earth?,       Max # Sources, Run Name,  Random Seed
                  Absolute,         No,          No,               10000,  AQ_test,     Fixed (Parallel)

Restart File Options:
# Cases Between Writes, Time Between Writes, Delete Old Files?, Write On Suspend?,
                      ,                    ,                  ,               No,

Multiple Case Options:
Dispersion Options Ensemble Size, Met Ensemble Size,
                               1,                  1,

Output Options:
Folder
./Output_AQ

! Using REGUM6

Input Files:
File names
../Resources/Defns/MetDefnUM6R.txt
../Resources/Defns/NAMEChemistryGridDefn_2km.txt
../Resources/Defns/NAMEChemistrySpecies.txt
../Resources/Defns/STOCHEMGridDefn.txt
../Resources/traffic_time_dep.txt
../Resources/AQ_tslocs_20071017.txt
../Resources/chem_uk_2003_nameIII.txt
../Resources/chem_emep_2003_nameIII_nouk.txt

Horizontal Coordinate Systems:
Name
Lat-Long
UK National Grid (100m)
EMEP 50km Grid

Vertical Coordinate Systems:
Name
m agl

Horizontal Grids:
Name,      H-Coord,  nx,   ny,   dx,     dy,     x min,   y min
HGridAll, Lat-Long,  50,  100,  0.4550,  0.1309, -10.7725, 47.06545

! 100 x  50 grid
HGridAll, Lat-Long, 100,   50,  0.2275,  0.2618, -10.88625, 47.1309

!  50 x 100 grid
HGridAll, Lat-Long,  50,  100,  0.4550,  0.1309, -10.7725, 47.06545

! Original grid
HGridAll, Lat-Long,  91,   77,  0.25,  0.17,  -10.875, 47.085

Horizontal Grids:
Name,          dx,       dy,    Set of Locations
Hsites,       0.25,    0.17,    TimeSeries Locations

Vertical Grids:
Name,    Z-Coord,  nz,   dz,   z0
Layer1,    m agl,   1,  100.0, 50.0
Layer21,   m agl,   1, 3000.0, 3500.0
Layer22,   m agl,   1, 5000.0, 7500.0

! Hourly output for time series, three-hourly output for fields

Temporal Grids:
Name,            nt,       dt,                 t0
Hrly,             9, 01:00:00, 08/05/2008 01:00:00
ThreeHrly,        3, 03:00:00, 08/05/2008 03:00:00

Domains:
Name,                      H-Coord,     X Min,    X Max,     Y Min,    Y Max, H Unbounded?, Z-Coord,   Z Max, Z Unbounded?,      Start Time,          End Time, T Unbounded?, Max Travel Time
ComputationalDomain,      Lat-Long,     -11.0,     12.0,      47.0,   60.26,           No,   m agl, 20000.0,           No,  08/05/2008 00:00,         Infinity,           no,          10 day

Chemistry Options:
Chemistry Folder,          Ozone Modelling Method
../Resources/Stochem/,             Plume Particles

! These particle mass limits are calibrated for the 2002 AQ source files.

Particle Mass Limits:
Species Name,    Particle Mass Limit
SULPHUR-DIOXIDE,             30000.0
AMMONIA,                      8750.0
NO,                           8750.0
NO2,                          1750.0
O3,                         400000.0
CO,                          35000.0
HCHO,                         2500.0
C2H4,                         1750.0
C3H6,                         1500.0
C5H8,                          100.0
OXYL,                          700.0
TOLUEN,                       2800.0
BD,                            210.0
PM10,                        15000.0

Output Requirements - Fields:
Quantity,                  Species, Source,        H-Grid,  Z-Grid,  T-Grid, BL Average, T Av Or Int, Av Time, # Av Times, Sync?, Output Route, Across, Separate File, Output Format, Output Group
Air Concentration,  SULPHUR-DIOXIDE,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Air Concentration,               CO,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Air Concentration,              NO2,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Air Concentration,               NO,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Chemistry Field,                 O3,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Air Concentration,             PM10,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Air Concentration,         SULPHATE,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Air Concentration,          NH42SO4,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Air Concentration,           NH4NO3,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Air Concentration,             NAER,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Air Concentration,              NO3,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Air Concentration,             HNO3,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Chemistry Field,                 OH,       ,     HGridAll,  Layer1, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid1
Air Concentration,  SULPHUR-DIOXIDE,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Air Concentration,               CO,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Air Concentration,              NO2,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Air Concentration,               NO,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Chemistry Field,                 O3,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Air Concentration,             PM10,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Air Concentration,         SULPHATE,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Air Concentration,          NH42SO4,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Air Concentration,           NH4NO3,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Air Concentration,             NAER,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Air Concentration,              NO3,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Air Concentration,             HNO3,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Chemistry Field,                 OH,       ,     HGridAll, Layer21, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid2
Air Concentration,  SULPHUR-DIOXIDE,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Air Concentration,               CO,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Air Concentration,              NO2,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Air Concentration,               NO,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Chemistry Field,                 O3,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Air Concentration,             PM10,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Air Concentration,         SULPHATE,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Air Concentration,          NH42SO4,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Air Concentration,           NH4NO3,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Air Concentration,             NAER,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Air Concentration,              NO3,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Air Concentration,             HNO3,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Chemistry Field,                 OH,       ,     HGridAll, Layer22, ThreeHrly,       No,          Av,   01:00,          4,   Yes,            D,    TZ,             T,          IA2, Fields_grid3
Air Concentration,  SULPHUR-DIOXIDE,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,               CO,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,              NO2,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,               NO,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Chemistry Field,                 O3,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,             PM10,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,         SULPHATE,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,          NH42SO4,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,           NH4NO3,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,             NAER,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,          AMMONIA,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,              NO3,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,             N2O5,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,             HNO3,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,             HCHO,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,             C2H4,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,             C3H6,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,             C5H8,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,             OXYL,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,           TOLUEN,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,               BD,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,           CH3CHO,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,              PAN,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1
Air Concentration,             HONO,       ,       Hsites,  Layer1,      Hrly,       No,          Av,   01:00,          4,   Yes,            D,   XYZ,           XYN,          AZ2, Time_series_grid1


Output Requirements - Fields:
Quantity,    Species, Source, T-Grid, H-Coord, Z-Coord, T Av Or Int, Sync?,  Output Route, Output Format, Output Group
# Particles,        ,       ,   Hrly,        ,        ,          No,   Yes,             D,             A, Numbers


Sets of Dispersion Options:
Max # Particles, Max # Full Particles, Max # Puffs, Max # Original Puffs, Skew Time, Velocity Memory Time, Inhomogeneous Time, Meander Velocity Memory Time, DeltaOpt, Puff Time, Sync Time,  Computational Domain, Puff Interval, Deep Convection?, Radioactive Decay?, Agent Decay?, Dry Deposition?, Wet Deposition?, Turbulence?, Meander?, Chemistry?
8000000,                                2,           2,                    2,     00:00,                00:00,              00:00,                        00:00,        1,     00:00,  00:15:00,   ComputationalDomain,      00:15:00,               No,                 No,           No,             Yes,             Yes,         Yes,      Yes,      Yes

NWP Met Module Instances:
Name,      Min B L Depth, Max B L Depth, Use NWP BL Depth?,             Restore Met Script,     Delete Met?,             Met Folder,          Topography Folder, Met Definition Name
Regional,          100.0,        4000.0,               Yes,                              ,              No,  /data/local/apem/ParallelMet,      ../Resources/Topog,               UM6R

NWP Flow Module Instances:
Name,      Met Module,       Met, Domain
Regional,     NWP Met,  Regional, UM6R Whole

Flow Order: Update
Flow Module, Flow
NWP Flow,    Regional

Flow Order: Convert
Flow Module, Flow
NWP Flow,    Regional

Flow Order: Flow
Flow Module, Flow
NWP Flow,    Regional

Flow Order: Cloud
Flow Module, Flow
NWP Flow,    Regional

Flow Order: Rain
Flow Module, Flow
NWP Flow,    Regional

Flow Attributes:
Name,    Flow Order
Update,  Update
Convert, Convert
Flow,    Flow
Cloud,   Cloud
Rain,    Rain
```

Wednesday April 07, 2010 1/1

Figure B.2: Input file `air_quality_test_short.txt` for the benchmark described in section 3.1.3.

Printed by eike.mueller

Mar 26, 10 13:44 **coarse.txt** Page 1/1

```
----------------------------------------------------------
! Project: AQ coarse run over Europe
! File: NAME III AQ run with emep 2006 data on airq
! Author: Claire Witham
! Date: 24/03/2009
! For NAME III version: 5.3 modified
----------------------------------------------------------

Main Options:
Absolute or Relative Time?, Fixed Met?, Flat Earth?,           Run Name,  Random Seed, Max # Sources
                  Absolute,         No,          No,     Parallel_Par1_coarse,        Fixed,      6000

Restart File Options:
# Cases Between Writes, Time Between Writes, Delete Old Files?, Write On Suspend?,
                      ,              720:00,                No,               yes,


Multiple Case Options:
Dispersion Options Ensemble Size, Met Ensemble Size,
                               1,                  1,

Output Options:
Folder
/data/local/apdl/Parallel/Par1/output_coarse

! Using UMG_Mk6_L59PT2

Input Files:
File names
/home/h06/apdl/NameIII/Version5_4a/Resources/Defns/MetDefnUMG_Mk6_L52PTpp.txt
/home/h06/apdl/NameIII/Version5_4a/Resources/Defns/STOCHEMGridDefn.txt
/home/h06/apdl/Parallel/input/NAMEChemistrySpecies_BackgroundPart.txt
/home/h06/apdl/Parallel/input/traffic_time_dep_2003.txt
/home/h06/apdl/Parallel/input/NAMEChemistryGridDefn_UKAQFC_coarse.txt
/home/h06/apdl/AQ_Emissions/Emission_files/chem_2007_nameIII_emep_POCP.txt
/data/local/apdl/Parallel/Par1/input/boundary_sources_coarse_full.txt
/data/local/apdl/Parallel/Par1/input/InitialConditions_coarse_02

Horizontal Coordinate Systems:
Name
Lat-Long
UK National Grid (100m)
EMEP 50km Grid

Vertical Coordinate Systems:
Name
m agl

Horizontal Grids:
Name,       H-Coord,  nx,   ny,    dx,    dy,    x min,   y min
HGrid1,    Lat-Long,  80,   50,   0.5,   0.5,   -14.75,   40.25

Horizontal Grids:
Name,       H-Coord,  nx,   ny,     dx,     dy, x centre,   y centre
HGridNW,   Lat-Long,   1,    1,   3.74,   0.14,    -7.35,     61.04
HGridNC,   Lat-Long,   1,    1,   3.74,   0.14,    -3.61,     61.04
HGridNE,   Lat-Long,   1,    1,   3.74,   0.14,     0.13,     61.04
HGridEN,   Lat-Long,   1,    1,   0.22,   4.13,     2.11,     59.045
HGridECN,  Lat-Long,   1,    1,   0.22,   2.03,     2.11,     55.965
HGridECS,  Lat-Long,   1,    1,   0.22,   2.03,     2.11,     53.935
HGridESN,  Lat-Long,   1,    1,   0.22,   2.03,     2.11,     51.905
HGridESS,  Lat-Long,   1,    1,   0.22,   2.03,     2.11,     49.875
HGridSEE,  Lat-Long,   1,    1,   1.87,   0.14,    1.065,     48.93
HGridSEW,  Lat-Long,   1,    1,   1.87,   0.14,   -0.805,     48.93
HGridSCE,  Lat-Long,   1,    1,   1.87,   0.14,   -2.675,     48.93
HGridSCW,  Lat-Long,   1,    1,   1.87,   0.14,   -4.545,     48.93
HGridSW,   Lat-Long,   1,    1,   3.74,   0.14,    -7.35,     48.93
HGridWS,   Lat-Long,   1,    1,   0.22,   4.06,    -9.33,     50.89
HGridWC,   Lat-Long,   1,    1,   0.22,   4.06,    -9.33,     54.95
HGridWN,   Lat-Long,   1,    1,   0.22,   4.06,    -9.33,     59.045
HGridTOP,  Lat-Long,   1,    1,  11.66,  12.25,    -3.61,     54.985

Array: VertZL3
Array Values
1250.0
3750.0

Array: VertZ123
Array Values
500.0
1750.0
3750.0

Array: VertdZL3
Array Values
2500.0
2500.0

Array: VertdZ123
Array Values
1000.0
1500.0
2500.0

Vertical Grids:
Name,    Z-Coord,  Z-Array,  Av Z-Array
LayersL3,  m agl,  VertZL3,  VertdZL3
Layers123, m agl, VertZ123,  VertdZ123

Vertical Grids:
Name,    Z-Coord,  nz,    dz,     z0
LayerSurf, m agl,   1, 100.0,   50.0
Top,       m agl,   1, 500.0, 5250.0

! Hourly output for verification fields, SixHourly for boundary conditions

Temporal Grids:
Name,                nt,         dt,                    t0
SixHrly,         24,   06:00:00,  11/02/2010 06:00:00
Hrly,             144,   01:00:00,  11/02/2010 01:00:00

Domains:
Name,                          H-Coord,      X Min,    X Max,    Y Min,   Y Max, H Unbounded?, Z-Coord,   Z Max, Z Unbounded?,          Start Time,    End Time, T Unbounded?, Max Travel Time
MainDomain,                   Lat-Long,     -15.00,    25.00,    40.00,   65.00,           No,   m agl,  6000.0,           No,  11/02/2010  00:00,    Infinity,          no,          10 day
BiggerXYZSmallerTravelT,      Lat-Long,     -16.00,    26.00,    39.00,   66.00,           No,   m agl,  7000.0,           No,  11/02/2010  00:00,    Infinity,          no,        00:07:30
Unbounded,                    Lat-Long,     -16.00,    26.00,    39.00,   66.00,           No,   m agl,  7000.0,           No,  11/02/2010  00:00,    Infinity,          no,        infinity

Chemistry Options:
Chemistry Folder,                                  Ozone Modelling Method
/home/h06/apdl/AQ/ozone_casestudy/input/Stochem_MH/,       Background Particles

! These particle mass limits are calibrated for chem_2007_nameIII_emep_POCP.txt on the coarse run grid

Particle Mass Limits:
Species Name,    Particle Mass Limit
O3,                          4000000.0
SULPHUR-DIOXIDE,              600000.0
AMMONIA,                      225000.0
NO,                           450000.0
NO2,                           76000.0
CO,                         14000000.0
HCHO,                          52000.0
C2H4,                         142000.0
C3H6,                          70000.0
C5H8,                           2000.0
OXYL,                          46000.0
TOLUEN,                       112000.0
BD,                            15000.0
PM10,                         140000.0

! Air Concentrations for Time-series

Output Requirements - Fields:
Name,          Quantity,                                                     Species, Source,        H-Grid,    Z-Grid,    T-Grid, BL Average, T Av Or Int, Av Time, # Av Times, Sync?, Output Route, Across, Separate File, Output Format, Output Group
NW,  Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,        HGridNW,  LayersL3,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridNW
NC,  Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,        HGridNC,  LayersL3,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridNC
NE,  Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,        HGridNE,  LayersL3,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridNE
EN,  Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,        HGridEN,  LayersL3,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridEN
ECN, Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,       HGridECN, Layers123,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridECN
ECS, Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,       HGridECS, Layers123,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridECS
ESN, Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,       HGridESN, Layers123,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridESN
ESS, Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,       HGridESS, Layers123,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridESS
SEE, Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,       HGridSEE, Layers123,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridSEE
SEW, Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,       HGridSEW, Layers123,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridSEW
SCE, Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,       HGridSCE, Layers123,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridSCE
SCW, Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,       HGridSCW, Layers123,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridSCW
SW,  Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,        HGridSW,  LayersL3,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridSW
WS,  Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,        HGridWS,  LayersL3,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridWS
WC,  Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,        HGridWC,  LayersL3,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridWC
WN,  Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,        HGridWN,  LayersL3,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridWN
TOP, Air Concentration, CO; O3; PM10; SULPHATE; NH42SO4; NH4NO3; NAER; NO; NO2; HCHO; HNO3,       ,       HGridTOP,       Top,   SixHrly,         No,          Av,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridTOP

Output Requirements - Fields:
Quantity,                  Species, Source,        H-Grid,    Z-Grid,    T-Grid, BL Average, T Av Or Int, Av Time, # Av Times, Sync?, Output Route, Across, Separate File, Output Format, Output Group
# Particles,                      ,       ,        HGridNW,  LayersL3,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridNW
# Particles,                      ,       ,        HGridNC,  LayersL3,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridNC
# Particles,                      ,       ,        HGridNE,  LayersL3,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridNE
# Particles,                      ,       ,        HGridEN,  LayersL3,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridEN
# Particles,                      ,       ,       HGridECN, Layers123,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridECN
# Particles,                      ,       ,       HGridECS, Layers123,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridECS
# Particles,                      ,       ,       HGridESN, Layers123,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridESN
# Particles,                      ,       ,       HGridESS, Layers123,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridESS
# Particles,                      ,       ,       HGridSEE, Layers123,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridSEE
# Particles,                      ,       ,       HGridSEW, Layers123,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridSEW
# Particles,                      ,       ,       HGridSCE, Layers123,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridSCE
# Particles,                      ,       ,       HGridSCW, Layers123,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridSCW
# Particles,                      ,       ,        HGridSW,  LayersL3,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridSW
# Particles,                      ,       ,        HGridWS,  LayersL3,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridWS
# Particles,                      ,       ,        HGridWC,  LayersL3,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridWC
# Particles,                      ,       ,        HGridWN,  LayersL3,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridWN
# Particles,                      ,       ,       HGridTOP,       Top,   SixHrly,         No,          No,   06:00,         24,   Yes,            D,    XYZ,           XYN,           AZ2, Time_series_gridTOP


! Air Concentration on Fields

Output Requirements - Fields:
Quantity,                  Species, Source,        H-Grid,   Z-Grid,   T-Grid, BL Average, T Av Or Int, Av Time, # Av Times, Sync?, Output Route, Across, Separate File, Output Format, Output Group
# Particles,                      ,       ,        HGrid1, LayersL3,  SixHrly,         No,          No,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,  SULPHUR-DIOXIDE,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,               CO,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,              NO2,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,               NO,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,               O3,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,             PM10,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,         SULPHATE,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,          NH42SO4,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,           NH4NO3,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,             NAER,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,             HCHO,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1
Air Concentration,             HNO3,       ,        HGrid1, LayersL3,  SixHrly,         No,          Av,   06:00,         24,   Yes,            D,     TZ,             T,           IA2, Fields_grid1

! Air Concentration on Fields - time-averaged outputs

Output Requirements - Fields:
Quantity,                  Species, Source,        H-Grid, Z-Grid,  T-Grid, BL Average, T Av Or Int, Av Time, # Av Times, Sync?, Output Route, Across, Separate File, Output Format, Output Group
Air Concentration,  SULPHUR-DIOXIDE,       ,        HGrid1, LayerSurf, Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,               CO,       ,        HGrid1, LayerSurf, Hrly,         No,          Av,   08:00,         32,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,              NO2,       ,        HGrid1, LayerSurf, Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,               NO,       ,        HGrid1, LayerSurf, Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,               O3,       ,        HGrid1, LayerSurf, Hrly,         No,          Av,   08:00,         32,   Yes,            D,     Tz,             T,          ZIA2, Fields_grid2
Air Concentration,             PM10,       ,        HGrid1, LayerSurf, Hrly,         No,          Av,   24:00,         96,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,         SULPHATE,       ,        HGrid1, LayerSurf, Hrly,         No,          Av,   24:00,         96,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,          NH42SO4,       ,        HGrid1, LayerSurf, Hrly,         No,          Av,   24:00,         96,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,           NH4NO3,       ,        HGrid1, LayerSurf, Hrly,         No,          Av,   24:00,         96,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,             NAER,       ,        HGrid1, LayerSurf, Hrly,         No,          Av,   24:00,         96,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2


! Total number of particles

Output Requirements - Fields:
Quantity,    Species, Source, T-Grid, H-Coord, Z-Coord, T Av Or Int, Sync?, Output Route,  Output Format, Output Group
# Particles,        ,       ,   SixHrly,        ,        ,          No,   Yes,            D,              A, Numbers

Sets of Dispersion Options:
Max # Particles, Max # Full Particles, Max # Puffs, Max # Original Puffs, Skew Time, Velocity Memory Time, Inhomogeneous Time, Meander Velocity Memory Time, DeltaOpt, Puff Time, Sync Time,  Computational Domain, Puff Interval, Deep Convection?, Radioactive Decay?, Agent Decay?, Dry Deposition?, Wet Deposition?, Turbulence?, Meander?, Chemistry?
10000000,                          2,           2,                    2,     00:00,                00:00,              00:00,                        00:00,        1,     00:00,  00:15:00,             Unbounded,      00:15:00,               No,                 No,           No,              Yes,             Yes,         Yes,      Yes,      Yes

NWP Met Module Instances:
Name,      Min B L Depth, Max B L Depth, Use NWP BL Depth?,      Restore Met Script,   Delete Met?,  Met Folder,                         Topography Folder,   Met Definition Name
UMG_Mk6_L59PT2,           100.0,        4000.0,               Yes,  ,            No,    /data/local/apdl/AQtest/met,  /home/h03/apdg/NameIII/NameIIILibrary/Data/UMTopogData,   UMG_Mk6_L52PT2pp

NWP Flow Module Instances:
Name,                            Met Module,       Met, Domain
REGMainDomain,                      NWP Met,    UMG_Mk6_L59PT2, MainDomain
REGBiggerXYZSmallerTravelT,         NWP Met,    UMG_Mk6_L59PT2, BiggerXYZSmallerTravelT

Flow Order: Update
Flow Module, Flow
NWP Flow,    REGMainDomain
NWP Flow,    REGBiggerXYZSmallerTravelT

Flow Order: Convert
Flow Module, Flow
NWP Flow,    REGMainDomain
NWP Flow,    REGBiggerXYZSmallerTravelT

Flow Order: Flow
Flow Module, Flow
NWP Flow,    REGMainDomain
NWP Flow,    REGBiggerXYZSmallerTravelT

Flow Order: Cloud
Flow Module, Flow
NWP Flow,    REGMainDomain
NWP Flow,    REGBiggerXYZSmallerTravelT

Flow Order: Rain
Flow Module, Flow
NWP Flow,    REGMainDomain
NWP Flow,    REGBiggerXYZSmallerTravelT

Flow Attributes:
Name,    Flow Order
Update,  Update
Convert, Convert
Flow,    Flow
Cloud,   Cloud
Rain,    Rain
```

Friday March 26, 2010 1/1

Figure B.3: Input file for the coarse AQ benchmark described in section 3.1.1.

Printed by eike.mueller

Mar 26, 10 13:52 **hires.txt** Page 1/1

```
--------------------------------------------------------
! Project: AQ hires run over UK
! File: NAME III AQ run with uk and emep 2006 data on airq
! Author: Claire Witham
! Date: 24/03/2009
! For NAME III version: 5.3 modified
--------------------------------------------------------

Main Options:
Absolute or Relative Time?, Fixed Met?, Flat Earth?,       Run Name,  Random Seed, Max # Sources
                  Absolute,         No,          No,     Parallel_Par1_hires,        Fixed,  15000

Restart File Options:
# Cases Between Writes, Time Between Writes, Delete Old Files?, Write On Suspend?,
                      ,               24:00,                No,                No,


Multiple Case Options:
Dispersion Options Ensemble Size, Met Ensemble Size,
                               1,                 1,

Message controls:
Name,                  Action, # of Messages Before Suppression
Zero source strength, Warning,                                2

Output Options:
Folder
/data/local/apdl/Parallel/Par1/output_hires

! Using UMG_Mk6_L59PT2

Input Files:
File names
/home/h06/apdl/NameIII/Version5_4a/Resources/Defns/MetDefnUMG_Mk6_L52PTpp.txt
/home/h06/apdl/NameIII/Version5_4a/Resources/Defns/STOCHEMGridDefn.txt
/home/h06/apdl/Parallel/input/NAMEChemistrySpecies_BackgroundPart.txt
/home/h06/apdl/Parallel/input/traffic_time_dep_2003.txt
/home/h06/apdl/Parallel/input/AQ_timeseries_sites.txt
/home/h06/apdl/Parallel/input/NAMEChemistryGridDefn_UKAQFC_hires.txt
/home/h06/apdl/AQ_Emissions/Emission_files/chem_2007_nameIII_0.11x0.07_noentec_POCP.txt
/data/local/apdl/Parallel/Par1/input/InitialConditions_hires_02


Input Files:
File names
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_NW.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_NC.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_NE.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_EN.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_ECN.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_ECS.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_ESN.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_ESS.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_SEE.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_SEW.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_SCE.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_SCW.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_SW.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_WS.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_WC.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_WN.txt
/data/local/apdl/Parallel/Par1/input/BoundaryConditions_TOP.txt

Horizontal Coordinate Systems:
Name
Lat-Long
UK National Grid (100m)
EMEP 50km Grid

Vertical Coordinate Systems:
Name
m agl

Horizontal Grids:
Name,      H-Coord,  nx,   ny,    dx,    dy,    x min,   y min
HGrid1,   Lat-Long, 102,  171,  0.11,  0.07,   -9.165,  49.035

Horizontal Grids:
Name,           dx,      dy,    Set of Locations
Hsites,       0.11,    0.07,    TimeSeries Locations

Vertical Grids:
Name,    Z-Coord,  nz,    dz,   z0
Layer1,    m agl,   1, 100.0, 50.0

Vertical Grids:
Name,           Z-Coord,  nz,     dz,    z0
MetGrid10m   ,   m agl,   1,    1.0,    10.0
MetGrid2m    ,   m agl,   1,    0.5,    2.0
MetGridSurface,  m agl,   1,    0.0,    0.0


! Hourly output for time series and fields

Temporal Grids:
Name,             nt,         dt,                   t0
Hrly,       144,   01:00:00, 11/02/2010 01:00:00

Domains:
Name,                           H-Coord,       X Min,    X Max,     Y Min,   Y Max, H Unbounded?, Z-Coord,   Z Max, Z Unbounded?,        Start Time,    End Time, T Unbounded?, Max Travel Time
MainDomain,                    Lat-Long,       -9.22,     2.00,     49.00,   60.97,           No,   m agl,  5000.0,           No, 11/02/2010 00:00,    Infinity,           no,         10 day
BiggerXYZSmallerTravelT,       Lat-Long,       -9.44,     2.22,     48.86,   61.11,           No,   m agl,  6000.0,           No, 11/02/2010 00:00,    Infinity,           no,       00:05:00
Unbounded,                     Lat-Long,       -9.44,     2.22,     48.86,   61.11,           No,   m agl,  6000.0,           No, 11/02/2010 00:00,    Infinity,           no,       infinity

Chemistry Options:
Chemistry Folder,                                     Ozone Modelling Method
/home/h06/apdl/AQ/ozone_casestudy/input/Stochem_MH/,       Background Particles


! These particle mass limits are calibrated for chem_2007_nameIII_0.11x0.07_POCP.txt on the hires run grid

Particle Mass Limits:
Species Name,    Particle Mass Limit
O3,                        400000.0
SULPHUR-DIOXIDE,            60000.0
AMMONIA,                    28000.0
NO,                         46000.0
NO2,                        13000.0
CO,                       1400000.0
HCHO,                        7800.0
C2H4,                       21000.0
C3H6,                       10000.0
C5H8,                         200.0
OXYL,                        6800.0
TOLUEN,                     16000.0
BD,                          2200.0
PM10,                       14000.0

! Air Concentrations for Time-series

Output Requirements - Fields:
Quantity,                   Species, Source,        H-Grid, Z-Grid,  T-Grid, BL Average, T Av Or Int, Av Time, # Av Times, Sync?, Output Route, Across, Separate File, Output Format, Output Group
Air Concentration,  SULPHUR-DIOXIDE,       ,        Hsites, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,    XYZ,            XY,           AZ2, Time_series_grid1
Air Concentration,               CO,       ,        Hsites, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,    XYZ,            XY,           AZ2, Time_series_grid1
Air Concentration,              NO2,       ,        Hsites, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,    XYZ,            XY,           AZ2, Time_series_grid1
Air Concentration,               NO,       ,        Hsites, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,    XYZ,            XY,           AZ2, Time_series_grid1
Air Concentration,               O3,       ,        Hsites, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,    XYZ,            XY,           AZ2, Time_series_grid1
Air Concentration,             PM10,       ,        Hsites, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,    XYZ,            XY,           AZ2, Time_series_grid1
Air Concentration,         SULPHATE,       ,        Hsites, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,    XYZ,            XY,           AZ2, Time_series_grid1
Air Concentration,          NH42SO4,       ,        Hsites, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,    XYZ,            XY,           AZ2, Time_series_grid1
Air Concentration,           NH4NO3,       ,        Hsites, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,    XYZ,            XY,           AZ2, Time_series_grid1
Air Concentration,             NAER,       ,        Hsites, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,    XYZ,            XY,           AZ2, Time_series_grid1
# Particles,                      ,       ,        Hsites, Layer1,    Hrly,         No,          No,   01:00,          4,   Yes,            D,    XYZ,            XY,           AZ2, Time_series_grid1


! Air Concentration on Fields - hourly averages

Output Requirements - Fields:
Quantity,                   Species, Source,        H-Grid, Z-Grid,  T-Grid, BL Average, T Av Or Int, Av Time, # Av Times, Sync?, Output Route, Across, Separate File, Output Format, Output Group
Air Concentration,  SULPHUR-DIOXIDE,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid1
Air Concentration,               CO,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid1
Air Concentration,              NO2,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid1
Air Concentration,               NO,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid1
Air Concentration,               O3,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     Tz,             T,          ZIA2, Fields_grid1
Air Concentration,             PM10,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid1
Air Concentration,         SULPHATE,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid1
Air Concentration,          NH42SO4,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid1
Air Concentration,           NH4NO3,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid1
Air Concentration,             NAER,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid1

! Air Concentration on Fields - time-averaged outputs

Output Requirements - Fields:
Quantity,                   Species, Source,        H-Grid, Z-Grid,  T-Grid, BL Average, T Av Or Int, Av Time, # Av Times, Sync?, Output Route, Across, Separate File, Output Format, Output Group
Air Concentration,  SULPHUR-DIOXIDE,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,               CO,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   08:00,         32,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,              NO2,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,               NO,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,               O3,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   08:00,         32,   Yes,            D,     Tz,             T,          ZIA2, Fields_grid2
Air Concentration,             PM10,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   24:00,         96,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,         SULPHATE,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   24:00,         96,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,          NH42SO4,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   24:00,         96,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,           NH4NO3,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   24:00,         96,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
Air Concentration,             NAER,       ,        HGrid1, Layer1,    Hrly,         No,          Av,   24:00,         96,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2
# Particles,                      ,       ,        HGrid1, Layer1,    Hrly,         No,          No,   01:00,          4,   Yes,            D,     TZ,             T,          ZIA2, Fields_grid2

!Met Output

Output Requirements - Fields:
Name,           Quantity,   Species,   Source,   H-Coord,         H-Grid,        Z-Grid,  T-Grid, BL Average, T Av or Int,        Av Time,          # Av Times,   Sync?, Output Route,  Across,    Separate File,  Output Format,  Output Group
Met1,      Temperature (K),        ,         ,          ,         HGrid1,     MetGrid2m,    Hrly,          ,          No,              ,                   ,     Yes,            D,      TZ,                T,           EIAZ,  Met_output
Met2, Wind Direction (degrees),    ,         ,  Lat-Long,         HGrid1,    MetGrid10m,    Hrly,          ,          No,              ,                   ,     Yes,            D,      TZ,                T,           EIAZ,  Met_output
Met3, Wind Speed           ,       ,         ,          ,         HGrid1,    MetGrid10m,    Hrly,          ,          No,              ,                   ,     Yes,            D,      TZ,                T,           EIAZ,  Met_output
Met4, Boundary layer depth,        ,         ,          ,         HGrid1,MetGridSurface,    Hrly,          ,          No,              ,                   ,     Yes,            D,      TZ,                T,           EIAZ,  Met_output
Met5, Cloud Amount (oktas),        ,         ,          ,         HGrid1,MetGridSurface,    Hrly,          ,          No,              ,                   ,     Yes,            D,      TZ,                T,           EIAZ,  Met_output
Met6, Precipitation Rate (mm/hr),  ,         ,          ,         HGrid1,MetGridSurface,    Hrly,          ,          No,              ,                   ,     Yes,            D,      TZ,                T,           EIAZ,  Met_output

! Total number of particles

Output Requirements - Fields:
Quantity,    Species, Source, T-Grid, H-Coord, Z-Coord, T Av Or Int, Sync?, Output Route,  Output Format, Output Group
# Particles,        ,       ,   Hrly,        ,        ,          No,   Yes,            D,              A, Numbers

Sets of Dispersion Options:
Max # Particles, Max # Full Particles, Max # Puffs, Max # Original Puffs, Skew Time, Velocity Memory Time, Inhomogeneous Time, Meander Velocity Memory Time, DeltaOpt, Puff Time, Sync Time,  Computational Domain, Puff Interval, Deep Convection?, Radioactive Decay?, Agent Decay?, Dry Deposition?, Wet Deposition?, Turbulence?, Meander?, Chemistry?
10000000,                          2,           2,                    2,     00:00,                00:00,              00:00,                        00:00,        1,     00:00,  00:10:00,             Unbounded,      00:15:00,               No,                 No,           No,             Yes,              Yes,         Yes,      Yes,        Yes

NWP Met Module Instances:
Name,           Min B L Depth, Max B L Depth, Use NWP BL Depth?, Restore Met Script,   Delete Met?,  Met Folder,                       Topography Folder,   Met Definition Name
UMG_Mk6_L59PT2,            100.0,        4000.0,               Yes, ,             No,    /data/local/apdl/AQtest/met,  /home/h03/apdg/NameIII/NameIIILibrary/Data/UMTopogData,   UMG_Mk6_L52PT2pp

NWP Flow Module Instances:
Name,                           Met Module,       Met, Domain
REGMainDomain,                      NWP Met,    UMG_Mk6_L59PT2, MainDomain
REGBiggerXYZSmallerTravelT,         NWP Met,    UMG_Mk6_L59PT2, BiggerXYZSmallerTravelT

Flow Order: Update
Flow Module, Flow
NWP Flow,    REGMainDomain
NWP Flow,    REGBiggerXYZSmallerTravelT

Flow Order: Convert
Flow Module, Flow
NWP Flow,    REGMainDomain
NWP Flow,    REGBiggerXYZSmallerTravelT

Flow Order: Flow
Flow Module, Flow
NWP Flow,    REGMainDomain
NWP Flow,    REGBiggerXYZSmallerTravelT

Flow Order: Cloud
Flow Module, Flow
NWP Flow,    REGMainDomain
NWP Flow,    REGBiggerXYZSmallerTravelT

Flow Order: Rain
Flow Module, Flow
NWP Flow,    REGMainDomain
NWP Flow,    REGBiggerXYZSmallerTravelT

Flow Attributes:
Name,    Flow Order
Update,  Update
Convert, Convert
Flow,    Flow
Cloud,   Cloud
Rain,    Rain
```

Friday March 26, 2010 1/1

Figure B.4: Input file for the hires AQ benchmark described in section 3.1.1