PROCEEDINGS, Fire and Evacuation Modeling Technical Conference (FEMTC) 2014
Gaithersburg, Maryland, September 8-10, 2014

# PERFORMANCE ANALYSIS AND SHARED MEMORY PARALLELISATION OF FDS

DANIEL HAARHOFF, LUKAS ARNOLD

Jülich Supercomputing Centre
Institute for Advanced Simulation, Forschungszentrum Jülich GmbH
52425 Jülich, Germany
e-mail: l.arnold@fz-juelich.de

## ABSTRACT

Fire simulation is a complex issue due to the large number of physical and chemical processes involved. The code of FDS covers many of these using various models and is extensively verified and validated, but lacks support for modern multicore hardware.

This article documents the efforts of providing an Open Multi-Processing (OpenMP) parallelised version of the Fire Dynamics Simulator (FDS), version 6, that also permits hybrid use with the Message Passing Interface (MPI). As FDS does not allow for arbitrary domain decomposition to be used with MPI, the amount of computational resources is limited. An OpenMP parallelisation does not have these restrictions, but it is not able to use the resources as efficient as MPI does.

Prior to parallelising the code, FDS was profiled using various measurement systems. To allow parallelisation the radiation solver as well as the tophat filter for LES equation where altered. The achieved parallelisation and speedup for various architectures and problem sizes were measured.

A speedup of two is now attainable for common simulation cases on modern four-core processors and requires no additional setup by the user. Timings for various combinations of simultaneous usage of OpenMP and MPI are presented. Finally recommendations for further optimisation efforts are given.

## PARALLELISATION

Parallelism in software has a long history that has both driven and been dependent upon the available computational hardware. The rapid rate of hardware development provides an ever-changing playing field upon which software emerges and vanishes at an even quicker pace. Trying to utilise the given resources effectively while keeping software functional and maintainable is a challenge.

This section provides a compact overview of shared memory computer systems, the OpenMP approach and the measurement of parallel performance.

### Shared Memory Approach

The oldest, most established tool for parallelisation is most likely the message passing interface (MPI, [mpi]), a distributed memory approach. With MPI, the goal is to decompose the work to be performed in its entirety and spread it to multiple processes. Whenever data or results are needed from other processes, MPI allows these to be exchanged using explicit messages.

The orthogonal approach to this are shared memory approaches, the most prominent framework being Open Multi-Processing (OpenMP or OMP, [openmp]). Here all processes have access to the same memory and data can therefore be shared without message passing. This of course necessitates that measures are taken to prevent two processes from accessing or modifying memory simultaneously. As a result OpenMP requires a more fine grained parallelism than MPI.

Since the epoch of ever-rising processor speeds is over [Geer 2005], Moore's law has only held true due to the introduction of multi-core processors. This of course has led to a prevalence of shared memory systems using shared-memory parallelisation.

With the even longer running trend of building high-performance systems by clustering commodity hardware, a situation has emerged where the hardware exhibits nested levels of parallelisation. A schematic overview of a cluster setup is given in Figure 1. This structure of densely-packed shared memory nodes lends itself to being mirrored in software by using a hybrid of MPI and OpenMP; using MPI to spread processes (ranks) across nodes which communicate via a network, and OpenMP threads to further parallelise the execution of each MPI rank within a shared memory space. In general it is best practise to distribute the MPI ranks on the individual sockets to ensure an uniform memory access, as in common computer architectures the memory is split among the sockets and local to an individual socket.

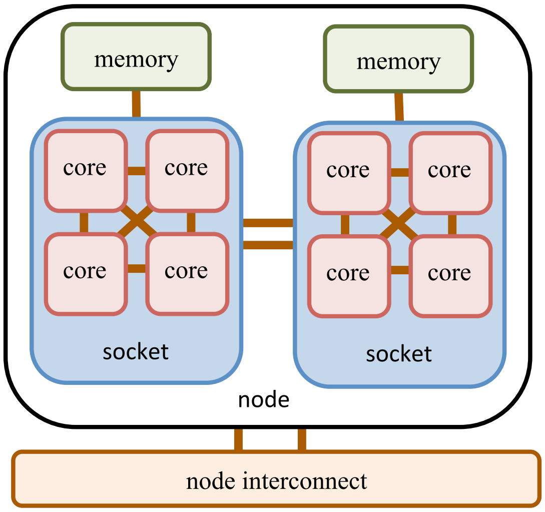

**Figure 1:** *Schematic outline of a compute node. It contains two sockets, each hosting four compute cores. Although the memory is attached to each socket, it can be accessed by all sockets; this results in general in a non-uniform memory access (NUMA) and therefore lower performance. All nodes are connected via a node interconnect. Good practice is to distribute the MPI ranks over the sockets and spawn an OpenMP thread on each of the socket's cores.*

While initially promising, studies have shown the added complexity of the hybrid approach only pays off in a limited set of use cases [Rabenseifner et al., 2009]. And while the pragma-based model of OpenMP allows for relatively easy parallelisation of existing software, the coarser parallelisation offered by domain decomposition and MPI will usually be well worth the additional programming effort.

It should be noted that MPI 3.0 supports shared memory constructs. At the time of writing, the support for this feature is not readily available in the compilers and MPI libraries and therefore has not been able to prove itself, which is why it was not an option for shared memory parallelisation during this work.

## OpenMP

The basic concept of OpenMP is to identify execution tasks (groups of instructions), which are independent of each other and therefore suitable for parallel execution. These tasks are then forked on the available $n$ OpenMP threads (this number includes the master thread and is specified in the `OMP_NUM_THREADS` environment variable). A simplified execution example in serial and parallel is given in Figure 2. In general each of these threads runs on a computational core and does not share hardware resources with the other threads. However, modern architectures require multiple threads – so called hyperthreading or Symmetric Mutlithreading (SMT) – to run on the same hardware to utilize all processing pipelines. After the computation the threads are joined. The execution continues with tasks depending on the joined results.

A characteristic situation for OpenMP parallelisation is a for-loop. If each iteration step is independent, every thread computes $1/n$ iterations. Benefit can only be gained if the overhead (task fork, non-uniform memory access, task join) is smaller than the serial execution. Thus very short loops, in terms of execution time, don't scale well.

In contrast to MPI, where the communication must be explicitly formulated, the parallelisation of a FORTRAN loop may take the following simple form[1], see Listing 1. The implicit OpenMP instructions are passed to the compiler via pragmas starting with `!$OMP`.

**Listing 1:** *FORTRAN code snippet of an OpenMP loop.*

```
!$OMP PARALLEL DO
do i = 1, length
  r(i) = a(i) + b(i)
end do
!$OMP END PARALLEL DO
```

In this example, the for-loop iterates over two arrays `a` and `b` to compute the element-wise sum into the array `r`. OpenMP would detect that all iterations are independent and distribute the work.

[1] The code examples in this article are adjusted extractions from the FDS source code to illustrate the main idea and are not stand-alone examples.

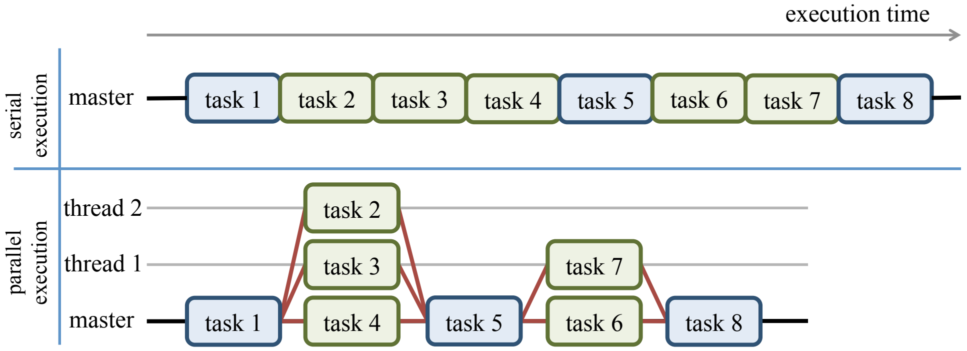

**Figure 2:** *Fork-join model. Shown are two execution modes – serial and parallel – of a sequence of tasks. Each green task group contains tasks that have no dependencies on each other, i.e. they can be executed at the same time; the blue tasks can not be overlapped with other tasks. Therefore during parallel execution green tasks can be distributed on other available threads to speedup computation by overlapping task execution. The overhead in this fork-join modell (red lines) is due to the task dispatch and the join of results.*

The pragma-based approach of OpenMP makes it easy to incrementally parallelise loops in existing code. With a lot of the heavy lifting being hidden behind these simple pragmas, the problem lies in ensuring that the compiler is actually doing what the software developer expects. Thus most of the effort after identifying the appropriate loops lies in verifying that they are parallelised, well load-balanced and not introducing any dataraces.

An example for a simple loop that was parallelised in FDS can be seen in Listing 2. The loop performs a standard finite difference where DZDX is calculated for each cell (line 7) and then used to update the density RHO_D_DZDX (this is repeated in all three dimensions) in line 9. Since DZDX and its brethren are local variables that are written during each loop cycle, they need to be declared as private (line 2) so as to ensure correct results and avoid dataraces.

**Listing 2:** *Example for an OpenMP parallel do-loop to compute field derivatives in parallel, adopted from divg.f90.*

```
1  ! Compute rho*D del Z
2  !$OMP PARALLEL DO PRIVATE(DZDX, DZDY, DZDZ)
       SCHEDULE (STATIC)
3  DO K=0,KBAR
4  DO J=0,JBAR
5  DO I=0,IBAR
6
7  DZDX = (ZZP(I+1,J,K,N) - ZZP(I,J,K,N)) *
8    RDXN(I)
9  RHO_D_DZDX(I,J,K) = .5_EB *
10   (RHO_D(I+1,J,K) + RHO_D(I,J,K)) * DZDX
11
12 [...]
13 !$OMP END PARALLEL DO
```

As both examples represent the ideal cases, in practical applications some issues require more attention. The following section illustrates some issues and challenges of using OpenMP, based on the applied parallelisation strategies in FDS.

## APPLIED PARALLELISATION

This section documents selected parallelisation techniques applied to some routines of FDS. It covers the tophat filter used in the flow solver, wall loops and the radiation solver. These cases illustrate common issues with shared memory parallelisation: loop reshape, data races and loop carried dependencies.

### Tophat Filter

The tophat filter used for the LES computations consumes roughly one percent of runtime. This does not make it one of the costliest functions, but it is one that lends itself to parallelisation as one can see in the degree of parallelisation that was achieved (see Table 5).

In the original version (Listing 3), the filter is applied by calling the 1D tophat filter routine (line 17) along each dimension of the mesh. This incurs a strided memory access which is detrimental to performance. This is why the serial version made copies to two work arrays (lines 16 and 20).

**Listing 3:** *Original structure of the* `test_filter` *function, adopted form a previous version of source file* `turb.f90`*.*

```
SUBROUTINE TEST_FILTER
  (PHIBAR,PHI,PHI_MIN,PHI_MAX)

REAL(EB), INTENT(IN) :: PHI_MIN,PHI_MAX
REAL(EB), INTENT(IN) ::
     PHI(0:IBP1,0:JBP1,0:KBP1)
REAL(EB), INTENT(OUT) ::
     PHIBAR(0:IBP1,0:JBP1,0:KBP1)
REAL(EB), POINTER, DIMENSION(:) :: PHI1,PHI2
INTEGER I,J,K

PHI1 => TURB_WORK11
PHI2 => TURB_WORK12

! filter in x:
DO K = 0,KBP1
  DO J = 0,JBP1
    PHI1(0:IBP1) = PHI(0:IBP1,J,K)
    CALL TOPHAT_FILTER_1D(PHI2(0:IBP1),
                      PHI1(0:IBP1),0,IBP1,
                      PHI_MIN,PHI_MAX)
    PHIBAR(0:IBP1,J,K) = PHI2(0:IBP1)
  ENDDO
ENDDO
[...]
END SUBROUTINE TEST_FILTER

SUBROUTINE TOPHAT_FILTER_1D
  (UBAR,U,N_LO,N_HI,U_MIN,U_MAX)

INTEGER, INTENT(IN) :: N_LO,N_HI
REAL(EB), INTENT(IN) ::
     U(N_LO:N_HI),U_MIN,U_MAX
REAL(EB), INTENT(OUT) :: UBAR(N_LO:N_HI)
INTEGER :: J

! Filter the u field to obtain ubar
DO J=N_LO+1,N_HI-1
  UBAR(J) = 0.5_EB*U(J) +
       0.25_EB*(U(J-1)+U(J+1))
ENDDO

! set boundary values (not ideal, but fast
     and simple)
UBAR(N_LO) = MIN( U_MAX, MAX( U_MIN,
    2._EB*UBAR(N_LO+1)-UBAR(N_LO+2)) )
UBAR(N_HI) = MIN( U_MAX, MAX( U_MIN,
    2._EB*UBAR(N_HI-1)-UBAR(N_HI-2)) )

END SUBROUTINE TOPHAT_FILTER_1D
```

**Listing 4:** *Restructured loop structure of the* `test_filter` *function to suite OpenMP parallelisation, taken from* `turb.f90`*.*

```
SUBROUTINE TEST_FILTER(HAT,ORIG)

REAL(EB), INTENT(IN) ::
     ORIG(0:IBP1,0:JBP1,0:KBP1)
REAL(EB), INTENT(OUT) ::
     HAT(0:IBP1,0:JBP1,0:KBP1)
INTEGER :: I, J, K, L, M, N
REAL(EB), PARAMETER :: K1D(3) = (/1.0, 2.0,
     1.0/)
REAL(EB), PARAMETER :: K3D(-1:1, -1:1,
     -1:1) = RESHAPE( (/ (((K1D
(I)*K1D(J)*K1D(K)/64.0,I=1,3),J=1,3),K=1,3)
     /), (/ 3,3,3 /) )

! Traverse bulk of mesh
!$OMP PARALLEL
!$OMP DO SCHEDULE(static)
DO K = 1,KBP1-1
  DO J = 1,JBP1-1
    DO I = 1,IBP1-1
      ! Apply 3x3x3 Kernel; this is faster
           than
      ! intrinsic array multiplication.
      HAT(I,J,K) = 0._EB
      DO N = -1,1
        DO M = -1,1
          DO L = -1,1
            HAT(I,J,K) = HAT(I,J,K) +
                 ORIG(I+L,J+M,K+N) * K3D(L,M,N)
          ENDDO
        ENDDO
       ENDDO
    ENDDO
  ENDDO
ENDDO

!$OMP END DO
! Traverse faces, edges and corners
!$OMP DO SCHEDULE(static)
DO K = 0,KBP1
  DO J = 0,JBP1
    HAT(0,J,K) = 2._EB * HAT(0+1,J,K) -
         HAT(0+2,J,K)
    HAT(IBP1,J,K) = 2._EB * HAT(IBP1-1,J,K)
         - HAT(IBP1-2,J,K)
  END DO
END DO
!$OMP END DO
[...]
END SUBROUTINE TEST_FILTER
```

To avoid these copy operations as well as to simplify the code, the three 1D filter operations were replaced with a single loop using a 3x3 filter kernel (Listing 4, lines 19 - 25). This produces a single big loop which is, due to its simple structure, straightforward to parallelise with OpenMP (lines 12 and 32).

The application of the tophat filter kernel for the LES equations requires an element-wise multiplication. This can be implemented using a Fortran intrinsic for element-wise multiplication or by manually decomposing the operation into three loops. The results of a micro-benchmark show a speedup of four for the latter method, see Table 1. The filter operation was applied (10.000 times) to the bulk (no outer shell) of a cubic 3D array with an edge length of 80.

| elementwise multiplication | runtime (s) |
|---|---|
| FORTRAN intrinsic | 84 |
| manual decomposition | 20 |

**Table 1:** *Timing of two ways to compute the filter kernel.*

The shell (faces, edges, and corners) of the mesh, to which the kernel cannot be applied, is computed as it was in the serial code. In the serial code, the shell for each dimension was processed after the bulk had been calculated in that dimension. Since the filter of the bulk is completed in all three dimensions prior to the shell operation, the results of the filter operation are not numerically identical – in general on the scale of machine precision. Due to the change in the order of the floating point operations, the results would differ numerically anyway, but the differences in the shell will be larger than in the bulk.

## Dataraces in Wall Loops

Dataraces are an inherent danger of shared memory parallelisation. Since all threads have access to the same memory they can read and write to the same location. While read-only access is not a problem and provides one of the benefits of shared memory parallelisation, namely not having to duplicate all memory for the workers, read-write access is difficult. For example when the body of a loop requires a temporary variable. If more than one thread uses the same memory to store this variable they will accidentally overwrite each other causing faulty results.

The wall loops in FDS all follow a similar structure. An example is given in Listing 5. They iterate over a set of wall loop indices (IW). Each IW identifies a wall cell WC which is a derived data type that includes the indices to the neighbouring cells in the solid (II, JJ, KK – not in the example) and the gaseous phase (IIG, JJG, KKG). The variable IOR indicates the orientation of the wall cell to the gaseous phase. The use of the neighbouring cells gives rise to detectable dataraces (for example with the Intel Threadchecker). The reason for this is that in corners several wall cells can share the same neighbour. While these would also overwrite each other in the serial execution of the wall loop, during a multi-threaded execution these writes might occur simultaneously (line 17) which could lead to data corruption. To overcome this issue, OpenMP critical or atomic statements were used (lines 16 and 18). This ensures that the write (or updates) of data occurs atomically using the appropriate hardware instructions.

**Listing 5:** *Example of a wall loop. The potential data races caused by multiple wall cells (`WC`) sharing the same neighbouring gas phase cell are solved using OpenMP atomic instructions. Adopted from `divg.f90`.*

```
1  WALL_LOOP2:
2  DO IW=1, N_EXT_WALL_CELLS+N_INT_WALL_CELLS
3
4    WC => WALL(IW)
5    [...]
6    IIG = WC%ONE_D%IIG
7    JJG = WC%ONE_D%JJG
8    KKG = WC%ONE_D%KKG
9    IOR = WC%ONE_D%IOR
10   [...]
11   RHO_D_DZDN = 2._EB*WC%RHODW(N) *
12     (ZZP(IIG,JJG,KKG,N)-WC%ZZ_F(N)) * WC%RDN
13   SELECT CASE(IOR)
14     CASE( 1)
15       !$OMP ATOMIC WRITE
16       RHO_D_DZDX(IIG-1,JJG,KKG) = RHO_D_DZDN
17       !$OMP END ATOMIC
18   [...]
```

## Loop Carried Dependency in the Radiation Solver

Dependencies of a loop iteration on previous iterations prevent a straight forward parallelisation. In the very general case such loops can not be paralellised as they are in fact a sequence of dependent instructions. However, sometimes there might be underlying structures which allow for loop transformations that remove, or reduce, the loop carried dependencies.

One of the largest runtime contributors is the function of the radiation solver `compute_radiation`. The big loop in this function is the radiation propagation. One of the settings for FDS is the number of angles over which the radiation propagation is discretised. Another is the number of spectral bands to use when computing the absorption. Both of these settings are reflected in loops in the radiation solver.

The radiation propagates for each angle in an individual loop. Depending upon the orientation with regard to the mesh, the propagation begins from any of the eight corners of the mesh. This is reflected in the propagation loop by setting the start, end and step direction of three nested loops used to traverse the mesh.

Within the propagation loop (Listing 6), three lines (6-8) give rise to a loop-carried dependency. During the propagation, the values for the current cell depend on the values calculated for previous cells. Figure 3 (left) illustrates the issue for a 2D mesh. The dependencies spread through the mesh like a wave, leading to the term loop-carried wavefront dependency.

**Listing 6:** *Abbreviated loop from the radiation solver to illustrate the loop-carried dependency. Adopted from* `radi.f90`*.*

```
1  SLICELOOP: DO IJK = 1, M_IJK
2    I = IJK_SLICE(1,IJK)
3    J = IJK_SLICE(2,IJK)
4    K = IJK_SLICE(3,IJK)
5
6    ILXU = IL(I-ISTEP,J,K)
7    ILYU = IL(I,J-JSTEP,K)
8    ILZU = IL(I,J,K-KSTEP)
9
10   AIU_SUM = AX*ILXU + AY*ILYU + AZ*ILZU
11   IL(I,J,K) = MAX(0._EB, RAP * AIU_SUM)
12 ENDDO SLICELOOP
```

While there exists solutions for many loop-carried dependencies [Hager & Wellein, 2010], only few cover the wavefront dependency [Anvik et al., 2001] and none are applicable to a 3D mesh in all eight directions of propagation. On that account, a new approach is proposed.

The idea is to parallelise within each wavefront, since the cells within a wavefront are not dependent upon each other. To do this, the indices of the cells in a wavefront need to be determined and stored in a list which can then be iterated over. The basic principle for this is illustrated in Figure 3 (right).

An explanatory implementation of this algorithm is given in Listing 7. The first loop (line 1) iterates over all possible wavefront sums. In the following, all cells belonging to the currently iterated wavefront are collected into the `slice` structure (lines 3 - 11). The actual computation (line 18) loops over all matching indices (line 14).

**Listing 7:** *Algorithm for removing loop carried dependency. This code only works for the case where the propagation runs in positive direction in all dimensions.*

```
1  do ind_sum = imin+jmin+kmin, imax+jmax+kmax
2    ! Determine the cells in the slice
3    do k = kmin, kmax
4      do j = jmin, jmax
5        i = ind_sum - k - j
6        if (i >= imin .and. i <= imax) then
7          cell_count = cell_count+1
8          slice(:, cell_count) = (/i,j,k/)
9        end if
10     enddo
11   enddo
12
13   ! Do the work
14   do ijk = 1, cell_count
15     i = slice(1,ijk)
16     j = slice(2,ijk)
17     k = slice(3,ijk)
18     a(i,j,k) = (a(i-1,j,k) + a(i,j-1,k) +
          a(i,j,k-1)) / 3
19   enddo
20 enddo
```

The next step is to generalise this for the other directions. This can be achieved by introducing a stepping direction into the calculation of the index sums and the index subtraction. This solution can easily be expanded for three dimensions and is implemented in the current version of FDS.

With this code, the propagation can be parallelised using OpenMP. The issue is that while the propagation is now parallel, the calculation of the indices introduces a new costly serial loop. The reason this loop is so costly is the branching within the inner loop caused by the bounds checks.

By calculating the point where the inner loop enters and exits the bounds, these checks can be omitted. Due to the necessity of running in eight directions, this requires some case selection. Also the loops used for calculation now always run in the positive direction (to ease the externalisation of the bounds calculation). Depending on the direction of propagation, the new bounds need to be calculated differently.

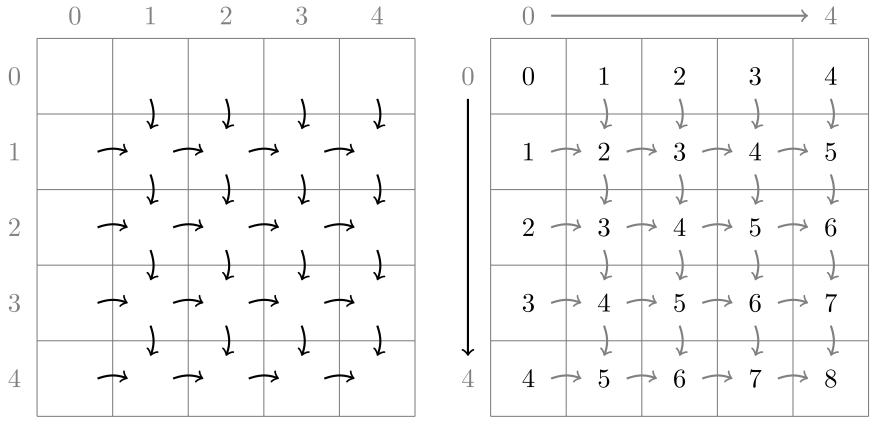

**Figure 3:** *The propagation of radiation in the radiation solver has a loop-carried dependency. The dependencies structure is illustration in a simplified 2D example on the left. Here the information flow is indicated by arrows. One can observe the wavefront of independent cells that moves through the mesh. The right depicts the sum of the x- and y-indices for each cell. These conform with the wavefronts and can be parallelised over.*

## ACHIEVED PERFORMANCE

It is important to distuingish speedup from performance. Speedup is a relative measure and depends upon the baseline that is used, usually the runtime of the serial executable. Bad serial performance will therefore usually lead to better speedups [Hager & Wellein, 2010].

This is why it is important to also have some form of absolute measure of performance. One way of doing this is calculating the number of cells that have been updated per second. This allows the comparison of various problem sizes as well as numbers of threads used.

Such a metric can be calculated thanks to the information provided by the FDS out files. These report the runtime required by the time steps as well as the mesh dimensions of the simulation. Taking these number we can calculate the cell calculations per second. Two things may distort this metric in FDS: the pressure solver and the stability checks. The pressure solver might take several or even hundreds of iterations before it achieves results below the allowed velocity error. And whenever the stability checks fail the predictor step is rerun with a smaller time step. Which means that the metric is not only dependant upon the computational cost of the solvers but also upon the case being simulated.

The benchmark case used for this work only requires a single pressure iteration and does not run into stability issues. The number of pressure iterations for each timestep is reported but since the pressure solver only constitus a part of the predictor corrector scheme one can not simply normalise the metric with this number. Which means that benchmarks with other cases need to be checked with regard to these two issues prior to using this metric.

### Benchmark Setup

As FDS is a very flexible software and is applied to a wide range of scenarios, it is difficult to find a representative benchmark scenario. Yet, the `bench2` input file, which is shipped with FDS, provides a reasonable compromise.

The slightly modified input consists of

- a computational domain 1.6m x 1.6m x 3.2m,
- equally discretised with varying number of grid cells, whereas in the following the sizes are abbreviated by $1k = 1024$ and $1M = 1k \cdot 1k$,
- a simple polyurethane burner with a power of 180kW, and
- soot particle tracing.

The following measurements are based on FDS version 6.1, compiled with version 13.1 of the Intel compiler. The computer systems used for benchmarking are outlined in Table 2.

As the measurement of execution time is essential during software parallelisation. The main performance measuring tools used in this work are scalasca [scalasca] and VTune [vtune]. A characteristic profile

| | **workstation** | **juropa2** | **juropa3** |
|---|---|---|---|
| processor(s) | i7-2600 | 2x Xeon X5570 | 2x Xeon E5-2650 |
| clockspeed | 3.4 GHz | 2.93 GHz | 2.0 GHz |
| cores (threads) | 4 (8) | 8 (16) | 16 (32) |
| cachesize | 8 MB | 8 MB | 20 MB |
| memory bandwidth | 21 GB/s | 32 GB/s | 51.2 GB/s |

**Table 2:** *Hardware specifications of the systems used for profiling and benchmarking. Juropa3 has a variety of different node types with various accelerators, this information is not pertinent since we did not utilise them. The thread number given above indicates the number of treads in hyperthreading mode.*

of the serial code is listed in Table 3 in a top view, i.e. the time spent in functions directly called by the main function.

| **function** | **t[s]** | **t[%]** |
|---|---|---|
| divergence_part_1 | 48.4 | 33.7 |
| compute_velocity_flux | 20.0 | 13.9 |
| mass_finite_differences | 15.9 | 11.1 |
| compute_radiation | 13.4 | 9.3 |
| update_particles | 7.4 | 5.2 |
| dump_mesh_outputs | 7.0 | 4.9 |
| pressure_iteration_scheme | 5.7 | 4.0 |

**Table 3:** *Topdown profile of the* bench2 *scenario in serial execution. Only the seven costliest functions are listed.*

The Table 4 shows the bottomup view of the execution. It allows to determine costly parts (loops or functions) of the source code which do not branch into other parts. The costliest lowest routines and loops are presented.

| **function** | **t[s]** | **t[%]** |
|---|---|---|
| scalar_face_value | 14.8 | 15.9 |
| get_sensible_enthalpy_diff | 4.0 | 4.3 |
| loop,l.1012,radiation_fvm | 3.7 | 4.0 |
| loop,l.151,div._part_1 | 2.4 | 2.6 |
| loop,l.672,velocity_flux | 2.4 | 2.6 |
| heat_transfer_bc | 2.3 | 2.5 |
| loop,l.732,velocity_flux | 2.3 | 2.4 |

**Table 4:** *Bottomup profile of the* bench2 *scenario in serial execution. Only the seven costliest loops and functions are listed.*

## OpenMP Speedup

With OpenMP being the focus of this work, at first the speedup achieved using only OpenMP is analysed. To be able determine the effect of the mesh size, additional versions of the benchmark setup were created with progressively larger meshes.

Looking at the speedup for pure OpenMP on these three systems in Figure 4, one can observe several interesting effects.

First of all the speedup continuously improves with increased mesh size. This is to be expected since larger meshes mean that more time is spent inside the parallelised region leading to a greater parallel fraction and ergo a greater speedup.

The second interesting effect is that hyperthreading is detrimental to performance. The workstation has four hardware threads, juropa2 has eight and juropa3 16. This means that the use of eight, 16 and 32 threads respectively utilises hyperthreading, i.e. overbooking each computational core. On all three platforms the speedup when using hyperthreading falls below that of using only hardware threads.

The third observation is that the hardware can improve the speedup. It is important to remember that the speedup is always calculated against the runtime of the serial version on the same hardware. This means that while the code might already run faster in general, better hardware also improves our scaling. This can be seen for the huge case where the speedup on juropa3 is consistently better than the speedup for the same thread count on the workstation and juropa2. This is most likely due to the differences in memory bandwidth and cache size. Juropa3 has 51.2 GB/s whereas juropa2 has 32 GB/s and my workstation only 21 GB/s. Hence investing in a newer processor generation with higher bandwidth might be worth the higher cost even when the OpenMP version of FDS obviously cannot make efficient use of the higher CPU

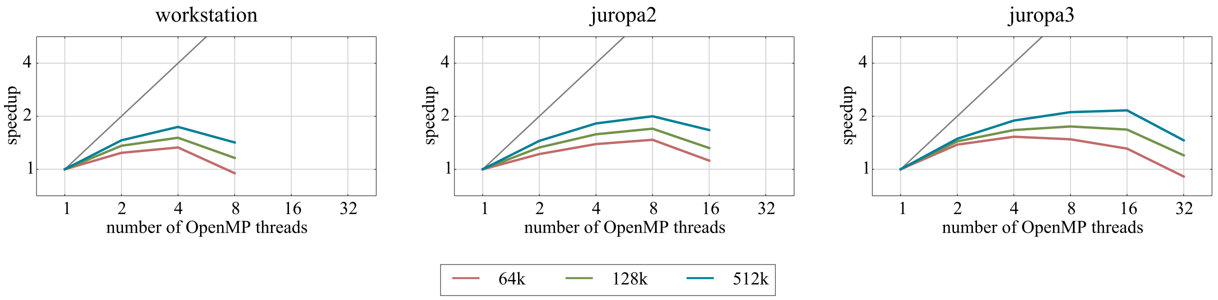

**Figure 4:** *FDS scaling of pure OpenMP. The benchmarks are run on three different computer configurations (Table 2) with three different mesh sizes: 64k, 128k and 512k cells in total. In all cases the speedup rises – although not ideally, as indicated by the grey line – up to a factor of about two. The last increase in the number of threads represents an oversubscription of two for each computational core, which leads to a decrease in speedup.*

count.

## Parallel Performance

At first a serial profiling is required to identify the regions to parallelise. After that a measurement of the success of the subsequent parallelisation is needed. This subsection covers the methods used to measure the degree of parallelisation and the effect of scheduling choices.

Measuring the degree of parallelism in a code is not straightforward. Counting the lines of code that are executed in parallel is of no use, as we are interested in execution time. One might consider counting machine instructions but it is questionable whether that is of any greater value, as modern computer architectures overlap instructions to hide memory access latency.

As a consequence a thread-individual runtime is a good choice, as provided by tools like scalasca [scalasca]. With this a parallel percentage $P$ can be calculated,

$$P = \frac{n \cdot t_{child}}{t_{master} + (n-1) \cdot t_{child}} \tag{1}$$

with $P$ being the parallel fraction, $t_{master}$ and $t_{child}$ the time spent respectively in the master and a single child thread and $n$ the number of threads used.

An alternative approach is based on the total runtime. The degree of parallelisation should give an indication as to the attainable speedup for a given number of cores or in this case threads. By taking the number of parallel threads into account, we can work backwards to determine a degree of parallelisation from the achieved reduction of runtime. By doing this we can also obtain a parallel percentage from the total runtimes reported by FDS.

$$P = \frac{t_{serial} - t_{OMP}}{n-1} \cdot \frac{n}{t_{serial}} \tag{2}$$

With P being the parallel fraction, $t_{serial}$ and $t_{OMP}$ the time used respectively by the serial and OpenMP version, and $n$ the number of threads used.

Bad scheduling means that the work is not spread equally between the OpenMP threads resulting in threads running idle while they wait for the others to complete. Such imbalance can usually be alleviated by changing the scheduling of an OpenMP loop. In most cases a static scheduling is sufficient.

Static scheduling means that the loop iterations are divided by the number of threads and each thread is given an equally sized block. In cases where the workload of the loop iterations is not equal, static scheduling causes an imbalance. A lot of loops in FDS have cycle statements that result in such differences in the workloads.

Such imbalances can be fixed using guided scheduling which assigns each thread a chunk of the iterations and when it completes assigns it a new chunk. With guided scheduling, the chunk size starts large and then decreases. Such scheduling incurs a greater overhead than static scheduling which is why it should only be applied at large imbalances.

## OpenMP Performance

Using the measure of cell updates per second on the data gathered on juropa2, the performance of FDS on juropa2 can be measured (Figure 5). As the mesh size increases so does the number of cell updates per second. However, as the thread count increases, performance does not increase in the same way; the resulting scaling is sub-optimal.

The detrimental effect of hyperthreading can be seen more clearly here. The cells per operation when using hyperthreading to accommodate 16 threads are consistently lower than when using eight or even only four threads.

One new effect we can see in Figure 5 is that the updates per second stagnate and decrease for the cases using more than half a million cells. This stagnation of the performance for the larger cases is very interesting. The fact that the performance remains constant for four threads and that the eight and 16 thread numbers converge to the same value points towards some bottleneck being saturated. That the hyperthreaded performance reaches the performance of the eight threaded case at four million cells is also intruiging. Which bottleneck(s) might be causing this is discussed later.

The following Table 5 lists all FDS routines that have so far been parallelised using OpenMP. Their parallel percentage $P$ was computed individually for each routine following Equation 2. For all functions except the `pressure_solver`, a sufficient parallelisation was achieved.

| **function** | $P$ |
|---|---|
| `divergence_part_1` | 82.8 |
| `species_advection` | 78.4 |
| `radiation_fvm` | 87.5 |
| `compute_viscosity` | 79.4 |
| `enthalpy_advection` | 69.5 |
| `mass_finite_differences` | 75.7 |
| `velocity_flux` | 97.2 |
| `density_advection` | 65.4 |
| `pressure_solver` | 17.2 |
| `test_filter` | 99.4 |
| `baroclinic_correction` | 99.8 |

**Table 5:** *The parallel percentage $P$ of all routines parallelised with OpenMP.*

## MPI and Hybrid Speedup

Having looked at OpenMP performance, the MPI and hybrid (MPI and OpenMP) parallelisation speedup is demonstrated. The benchmarks were done with only a single mesh size of 512k cells.

For the MPI cases the meshes were divided into equal parts, so that each MPI process computes one mesh. In the hybrid case, each process was assigned OpenMP threads. Thus for example with two MPI processes and four OpenMP threads, each MPI process spawned three child threads, in total utilising eight cores; no hyperthreading was used here.

The achieved pure MPI speedups are presented in Figure 6 (left). As was to be expected, the pure MPI parallelisation outperforms the OpenMP parallelisation (compare with Figure 4). This is due to the fact that a domain decomposition introduces parallelism at the outer level. Compared to the incremental approach with OpenMP, the parallelisation should consequently be greater, leading to greater speedup.

Accordingly, the speedup achieved using hybrid parallelisation lies below that of MPI parallelisation when using the same amount of CPUs, see Table 6.

| **MPI ranks** | **OMP tasks** | **speedup** |
|---|---|---|
| 4 | 8 | 1.00 |
| 8 | 4 | 1.89 |
| 16 | 2 | 2.71 |
| 32 | 1 | 3.42 |

**Table 6:** *Timing comparison of various combinations of MPI ranks and OpenMP tasks all using 32 cores in total.*

The important thing to note here is that MPI parallelisation is not always an option in FDS. This, of course, was the original motivation of this work. So as can be seen, when further MPI parallelisation is not possible but further computational resources are available on the same node, a hybrid parallelisation can utilise these additional resources, see right part of Figure 6.

# DISCUSSION

The goal of this work is to provide an OpenMP version of FDS that can also be used for hybrid parallelisation with MPI. This has been successfully implemented, verified and benchmarked. The current degree of parallelisation for OpenMP parallelised routines lies somewhere between 40 and 70 percent. On modern architectures (e.g. Intel Sandy Bridge) this leads to a speedup of roughly two when using four threads and

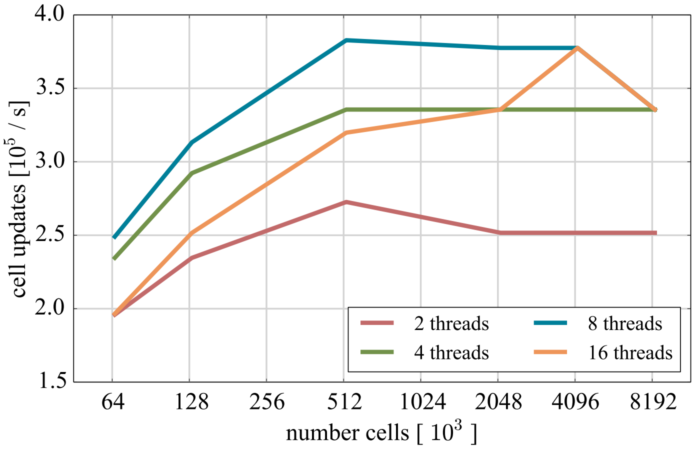

**Figure 5:** *Number of cell updates per second. Six mesh sizes (64k, 128k, 512k, 2M, 4M and 8M) and up to 8 (16 using hyper-threading) OpenMP threads on the juropa2 system are utilised. The maximal performance for all thread configurations is achieved at a mesh size of at least 512k cells.*

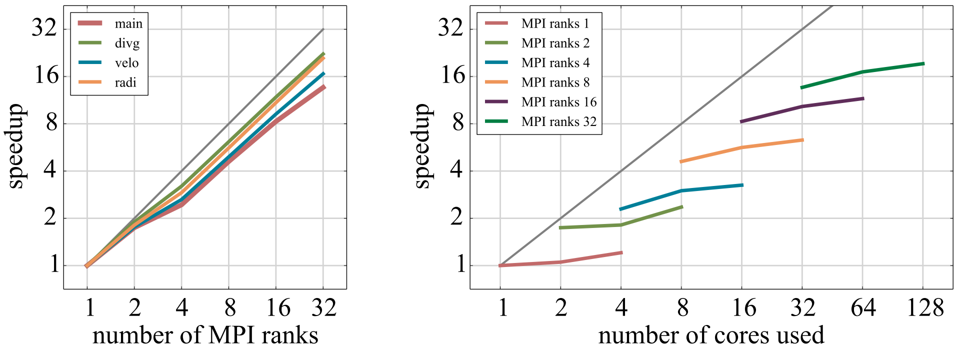

**Figure 6:** *Scaling of selected FDS modules in pure MPI execution (left) and hybrid (right), i.e. MPI in combination with OpenMP. The number of cells was 512k. The grey line indicates in both figures an ideal linear scaling, normalised to the serial execution. All runs were performed on juropa2. The pure MPI scaling behaviour of the main function and three selected modules (divg, velo and radi) of FDS are presented. In all cases the scaling is good. Each line in the hybrid scaling figure indicates a given number of MPI ranks, while the number of OpenMP tasks is increased (1, 2 and 4) and therefore the total number of cores utilised.*

thereby skirts the lower limit of what is considered acceptable computational efficiency (50 percent).

The parallelisation and speedup achieved is very low; it is not uncommon for scientific code to scale to thousands if not hundreds of thousands of cores. However these typically are more focused in their application and restricted in their setting. Thus a speedup of two, for a code that supports a wide range of applications and geometries is very respectable and great boost to the many scientists and especially engineers relying on FDS. That this speedup does not require additional configuration work by the user also deserves special mention.

Although the MPI based performance outruns the OpenMP one, the effort is not worthless, as in many applications of FDS the mesh can not be split arbitrary and therefore the MPI applicability is limited. The performance discrepancy is mainly due to the fact, that MPI decomposes the work on the very big scale and the overhead or serial part (communication, additional structures and work) is small. On the other side, OpenMP tackles small loops and hence faces relatively larger overheads.

The previously described stagnation of OpenMP performance (Figure 5) for cases involving meshes with more that 512k cells deserves futher study. Such stagnation usually points towards some bottleneck being saturated. Given that none of the code in FDS stands out as involving large amounts of floating point operation on little amounts of data, FDS is most likely memory bound. This means that some form of memory bandwidth is being saturated.

The memory bandwidth varies greatly depending on which level of cache is being accessed or whether we have to retrieve data from memory. As problem sizes increase the data required for computation stops fitting into the various levels of cache, so that memory access becomes slower.

Many hardware platforms now involve multiple sockets. To reduce costs each socket has access to all of the available memory, but certain regions can be accessed faster. This is referred to as non-uniform memory access (NUMA). Depending on how the memory is initially allocated it might happen that half of the threads have to constantly access the region that is slower for them.

While other factors might also contribute to the stagnation, these two (caching, NUMA) are the most common culprits for the saturation of memory bandwidth.

Approaches to solve these issues exist. Most NUMA issues are usually relatively straightforward to fix since one can allocate the memory using the parallel threads, forcing them to create their working sets in their own (faster) domain. This is of course only effective when the same threads access the same working sets later on. Dynamic scheduling or code like the new, parallel radiation solver don't benefit from this approach.

Caching issues can be fixed by tiling the problem set or generally adapting the way data is structured and loaded. Such changes are often non-trivial to implement and also due to the different cache sizes and architectures not necessarily portable.

So while a deeper analysis is definitely interesting, it might turn out that the changes required to overcome the bottleneck are not feasible given the limited ressources available for FDS development. The parallelisation of the pressure and radiation solver can still be improved with little effort and will be performed following this work. Further parallelisation beyond that is increasingly less likely to be of significant use.

## CONCLUSIONS

The profiling of FDS using ScoreP and VTune showed that FDS does not have any single loop that constitues the majority of the runtime. Instead the runtime is spread over many regions, with only a handfull contributing more than five percent. No loop in FDS exhibits an exclusive runtime of more than five percent. Many of the loops and functions in FDS involve many cases and function calls due to the flexibility of the codes usage. This makes it difficult to ensure thread-safe execution and avoid dataraces that cause invalid results.

A parallel version of the tophat filter for the LES equations was implemented, resulting in the filter exhibiting a parallelisation of over 99 percent. The wavefront loop-carried dependency in the radiation solver was also removed, allowing this important solver to also be parallelised.

Using a performance measure normalised over the number of cells, we can see that the performance of the OpenMP version increases with the number of cells until we hit a bottleneck between 500.000 and one million cells. The achieved speedup also increases with the number of cells to be computed.

The used hardware also makes a difference, with newer architectures with higher memory bandwidth exhibiting greater speedups.

The speedup of pure OpenMP is considerably smaller than that using MPI, but the OpenMP version does not require manual decomposition of the meshes.

Overall several conclusions can be made regarding the performance of the implemented version:

- a speedup of two is attainable with four cores
- larger cellcounts increase the speedup
- hyperthreading reduces performane
- MPI offers much greater speedup
- hybrid (MPI and OpenMP) use is possible

To achieve larger performance and scaling gains MPI is most likely the only path. This would require a new pressure and radiation solver, but could ensure valid results at mesh boundaries as well as an automated domain decomposition.

## ACKNOWLEDMENTS

The authors gratefully acknowledge the computing time granted on the supercomputer juropa at Jülich Supercomputing Centre (JSC).

## REFERENCES

[mpi] MPI: A Message-Passing Interface Standard, Version 3.0, www.mpi-forum.org

[openmp] OpenMP Application Program Interface, Version 4.0, www.openmp.org

[Geer 2005] Geer, D. (2005), ”Chip makers turn to multicore processors”, Computer, **38(5)**, 11-13

[Rabenseifner et al., 2009] Rabenseifner, R., Hager, G. and Jost, G. (2009), Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes., proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, in Weimar, Germany, Feb. 16-18, 2009, Computer Society Press, 427-436

[Hager & Wellein, 2010] Hager, G. and Wellein,G. (2010), ”Introduction to High Performance Computing for Scientists and Engineers”, CRC Press, ISBN 9781439811924

[Anvik et al., 2001] Anvik, J., MacDonald, S., Szafron, D., Schaeffer, J., Bromling, S. ; Kai Tan (2001), ”Generating parallel programs from the wavefront design pattern”, Parallel and Distributed Processing Symposium, 15-19 April 2001

[scalasca] Geimer, M., Wolf, F., Wylie, B., Ábrahám, E., Becker, B. and Mohr, B. (2010), ”The Scalasca performance toolset architecture”, Concurrency And Computation: Practice And Experience, 702-719

[vtune] Intel VTune Amplifier XE 2013, https://software.intel.com/en-us/intel-vtune-amplifier-xe