The parallelization of LESSTAG

Authors:

Willem Vermin[1]

Wim Timmermans[2]

Date: 2014-11-25

Abstract

In this paper, the optimization and parallelisation of the program LESSTAG, developed at the University of Twente is described.

> The current Large Eddy Simulation (LES) model was first developed in 1995 (published in Water Resources Research in 1996) to study the effects of spatially variable surface properties on areal averaged momentum and scalar (heat and water vapour) transport at the land-atmosphere interface. At a later stage the LES model was coupled to a remote sensing based soil-vegetation-atmosphere-transfer (SVAT) scheme, or land surface model (LSM), in order to study the feedback effects between surface state and spatial variability in fluxes, through the induction of spatial variability in the lower atmosphere.
>
> Since the focus is on near-wall processes, typically where LES models do not perform very well, recently newly developed scale-dependant dynamic Sub-Grid-Scale models for viscosity and diffusivity have been implemented. These models were reported to show better performance and provide improved predictions of mean velocity and scalar profiles. This better performance is most prominent near the wall (land surface) and also under so-called stable atmospheric conditions, when motions are relatively small. Since the aim of the developed model is to study the momentum, latent and sensible heat flux exchange under a variety of atmospheric and surface conditions this implementation was deemed necessary.
>
> First test runs using synthetic data inputs yielded the expected improved mean profiles as well as improved velocity and scalar spectra at different heights form the surface. However, implementation over real world surfaces, using remotely sensed data inputs resulted in unworkable calculation duration (> 100 days). Moreover, future plans include diurnal simulations using remote sensing inputs which would be impossible using the current serial code.

It was decided to parallelize the program in order to be able to run larger simulations. The project resulted in a program that scales to 64 and more processes.

This project was subsidized by SURFsara.

Cleaning up the source code

LESSTAG was written in Fortran90, but an Fortran77 way: for example no use was made of modules. The first steps to adapt the program was converting it to a code that allows the compiler to

1 W.J. Vermin, SURFsara https://www.surfsara.nl

2 W.J. Timmermans, University of Twente https://www.utwente.nl

check for the correctness of parameters and to include automatically code that checks for correct use of array boundaries. Furthermore, the arrays had fixed extents, and all run-time parameters were hard-coded.

The program was developed using the Portland Group Fortran compiler. It appeared that errors in the program, such as use of un-initialised variables and wrong type of parameters, were not noticed with that compiler, but became apparent when using the GNU of Intel compilers. Using Valgrind[3] it was possible to find the not-initialised variables. The wrong typed parameters were detected by the compilers.

All subroutines were placed in modules, and the run-time parameters are read now using the namelist construct. Arrays are allocated dynamically in run-time. Some code was coded in single precision, and some code in double precision. This was inspired by problems with memory allocation of static declared arrays. It was decided to use double precision in the whole program. Also we now use the type complex where applicable.

Speeding up the serial version

Profiling showed that the program spends most of it's time in 2-D Fourier computations and the determination of the maximum root of 5-th order polynomials. These two components were redesigned:

The Fourier computations are now done using the FFTW3[4] package, making use of the fact that only complex-to-real and real-to-complex transformations are done. This resulted in a speed-up of about a factor 3 for the whole program.

The usage of the FFTW3 package is simplified by creating an interface, described and coded in the file fourier.f90. For example, a 2-d transform from real to complex and vice-verse is done as follows:

```
call fourtrans(a,b)
```

Where a (input) and b (output) are two-dimensional arrays, one real and one complex. Using the fortran90 interface technique, the compiler decides on the types of the parameters which Fourier routine to call.

The determination of the maximum root of the 5-th degree polynomials was also redesigned: when the Newton-Raphson method fails, the method of Brent (a combination of bracketing and secant-Newton-Raphson) is used. This made the program about a factor 2 faster. Later on, the choice to use a real root-finder was incorporated: algorithm 493 of TOMS[5]. Also a modern rootfinder was introduced: an algorithm fro the Gnu Scientific Library[6]. This algorithm is somewhat faster that the old TOMS993 code. As was expected, these methods are slower than Newton-Raphson combined with Brent: the general root-finders find all roots, and we need only the largest.

The program produced lots of output files, ASCII-encoded. To speed this up, the output files can now, depending on an input parameter, be created in binary format, suitable for using with MATLAB.

---

3 Valgrind is an instrumentation framework for building dynamic analysis tools: http://valgrind.org/
4 FFTW3: 'the fastest Fourier transform in the West' : http://www.fftw.org/
5 http://www.netlib.org/toms/493
6 http://www.gnu.org/software/gsl/

Parallelization of the code

Distribution of the data

Since the Fourier transforms are done plane by plane, it was decided to dived the planes among the processes. The planes belonging to a process are numbered 1 .. nz, taking care that plane 1 is a copy of plane nz-1 of a neighbour process, and plane nz is a copy of plane 2 of the other neighbour process. (Trivial exceptions are the first and last processes) An example:

```
global     1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
process 0  1  2  3  4  5
process 1           1  2  3  4  5  6  7
process 2                       1  2  3  4  5  6  7
process 3                                      1  2  3  4  5  6
```

In above example, 19 planes are divided among 4 processes, each number represents a layer, The Z-direction is depicted horizontally. The total number of layers is renamed from 'nz' to 'globalnz'. Using this layout, each process can compute interactions between adjacent planes. In converting the serial code to the parallel code, each loop in the z-direction was examined to see what had to be done. For example, if the serial code loops from 2 to globalnz, the processes 0 has to send layer 4 to layer 1 of process 1, and process 1 has to send layer 6 to layer 1 of process 2, and at last process 2 sends layer 6 to layer 1 of process 3.

Of course, if the serial program changes a layer with a fixed number (for example 1 or globalnz), one has to determine which processes in the parallel version should deal with it.

Many loops in the Z-direction were of the type 'do k=1,globalnz'. In the parallel version, one can use loops like 'do k=1,nz', but also 'do k=2,nz-1' followed by the communication of the layers 1 and nz with the neighbours. Measurements indicated that it depends on the size of the system and the hardware used for MPI communication which is fastest. Therefore the program measures at the start which method is fasted and uses that further on.

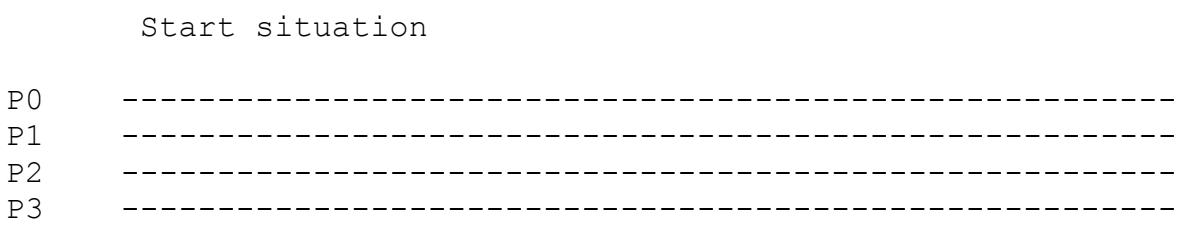
The communication of layers is done using MPI_Sendrecv, a collective operation.

Another distribution for computations in the Z-direction

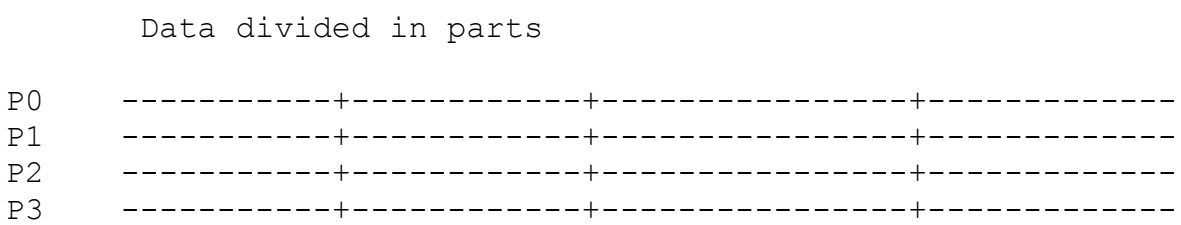
At one part in the program, above mentioned parallelisation paradigm does not work well, namely in subroutine p_stag that does global computations using some quantities in the Z-direction. More specifically: using 4 vectors, computed from vertically adjacent cells if a few arrays, of length globalnz, a tridiagonal system of linear equations is solved. To ensure that every process has all values available, the data involved is redistributed using MPI_Alltoallw, using special MPI_Types, created with MPI_Type_create_subarray. After the computations the relevant data is redistributed back.

Here follows a visualisation of the process:

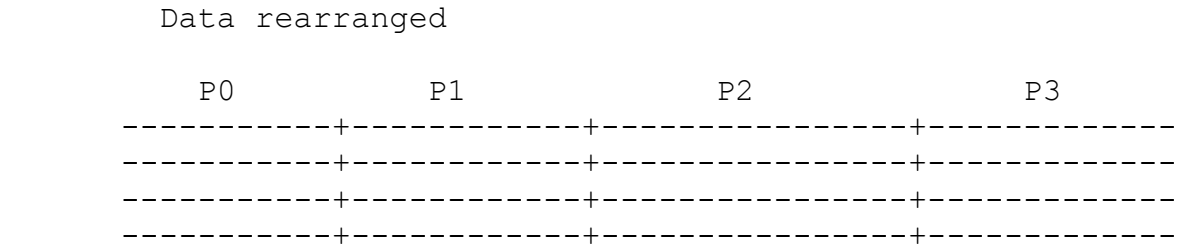
The array involved is divided over 4 processes: P0 .. P3. Each process contains part of the Z-direction and all of the other directions.



We choose a division the array in 4 parts, the number of parts must be equal to the number of processes:



In practice, this division is made as even as possible. After creating suitable MPI_Typs for each division and the call to MPI_Alltoallw, the division of the data is as follows:



So, each processor has now all data in the Z-direction available, but not in the other directions. In the program, one has to take into account, that the data in the original situation is overlapping in the Z-direction: this affects the way the MPI_Types are defined.

Input and output

All I/O is done by the master-process (the one with MPI-rank = 0). At the start of the program, the parameters are communicated to all processes using MPI_Bcast. The input-arrays are distributed using MPI_Scatterv. When arrays are to be output, the complete array is collected on the master process using MPI_Allgatherv and subsequently written to disk.

The program keeps track of a single data point with global coordinates, chosen by the user. The value of this point is, after determination which process owns the data point, sent to the master process using MPI_Send, and received using MPI_Recv.

MPI interface

In order to ease the use of MPI in the program, and not clutter the code with lengthy and error-prone MPI calls, quite a number of interface routines were coded. These are available and documented in the file lesmpi.f90. For example, to take care that layers 1 and nz of all processes are copied from the appropriate neighbour, this call suffices:

```
call lesmpi_getplane(a)
```

where 'a' is the 3-d array involved.
The output of the arrays is handled in the file outarray.f90. To output an array a call like:

```
call outarray(11,a)
```

takes care that the 3-d array a is sent to the master process, and the data is written to disk to unit 11. We implemented a few varieties of the outarray subroutine to facilitate slightly different file-layouts.

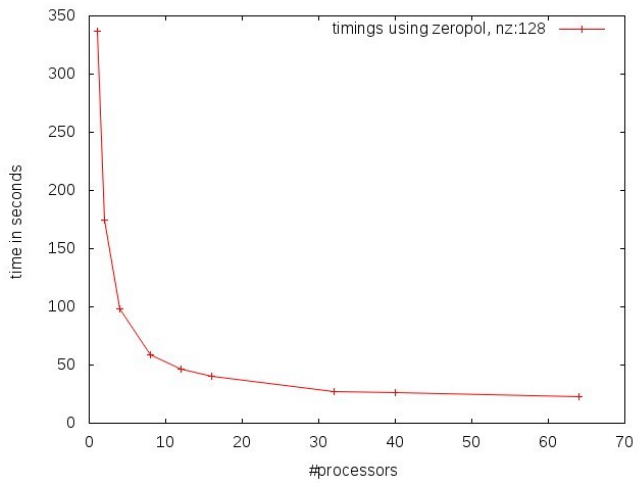
In summary: all MPI calls are hidden in specialised subroutines adapted to the special set-up of this program.

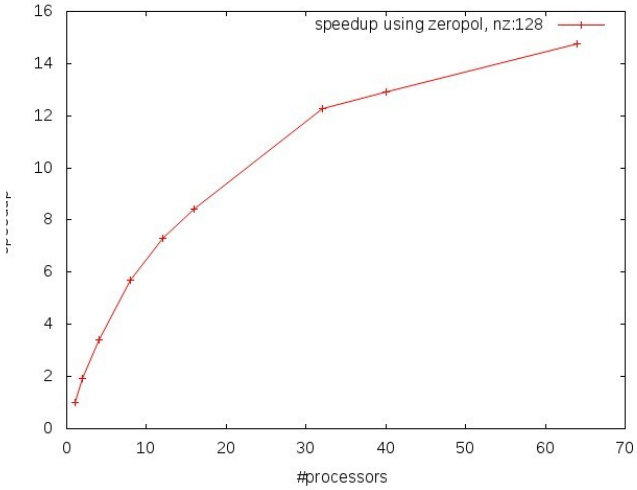
Speedup

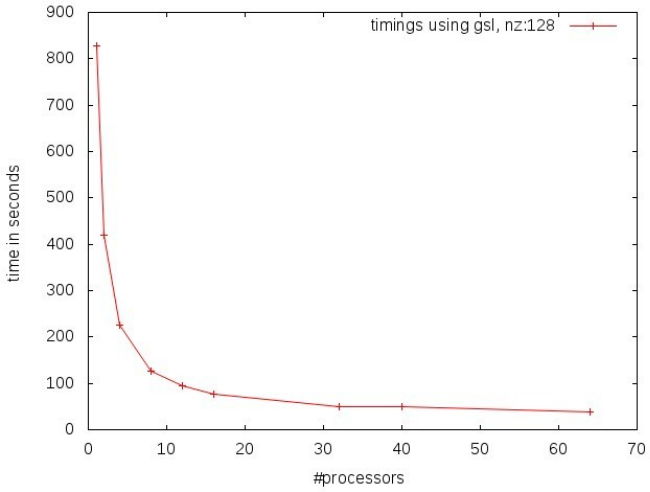
Below is a table of the runtime of an example: 128x128x128, running on 1 to 64 processes on the Cartesius system at SURFsara. The compiler was GNU fortran, the MPI library from Intel. The 'zeropol' variety is using the optimised Newton-Raphson method combined with Brent's method. The 'gsl' variety is using the 'gsl_poly_complex_solve' function from the GNU Scientific library. The 'rpoly' variety is using the RPOLY subroutine from TOMS493. The 'original' is the original code. Output to files was only done at the and of the program. Timings are in wall-clock seconds.

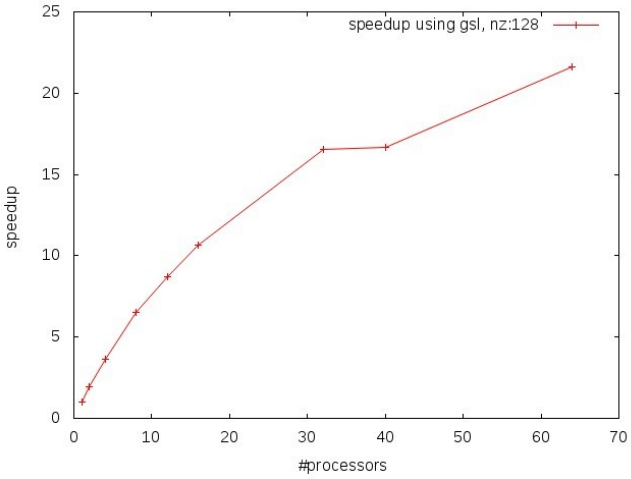
# processes	zeropol	rpoly	gsl	original
1	337	877	829	583
2	175	482	420	404
4	99	282	226	290
8	59	158	127	165
12	46	118	95	119
16	40	100	78	101
32	27	63	50	65
40	26	61	50	57
64	23	49	38	48

Graphical display of the results:









It is clear that the gsl version is about 2.5 times slower than the zeropol version, and therefore displays a better scaling behaviour. Interesting is the discontinuity in the speed-up graphs going from 32 to 40 processes: this is caused by the fact that one node on the Cartesius system contains 32 cores. Using more cores necessitates the use of the Infiniband network between the nodes, which is slower than memory-to-memory transfers within an node.

Using Scalasca, it appeared that there is some computational imbalance, mostly due to the fact that the master process is doing all the I/O. If MPI-I/O can help is not clear beforehand, but could be investigated in the future.

Concluding remarks

The parallelization was successful, satisfactorily scaling using a rather small problem to 32 – 64 processes. Larger problems will exhibit better scaling. Maybe there is something to win using MPI-I/O, but better I/O performance can be obtained by selecting the interesting data and not write nearly everything that is available.