Domain Decomposition of the Padé Scheme and Pseudo-Spectral Method, Used in Vlasov Simulations

Bengt Eliasson[1]

[1]Department of Information Technology, Scientific Computing, Box 337, SE-751 05 Uppsala, Sweden. Department of Astronomy and Space Physics, Box 537, SE-751 21 Uppsala, Sweden. email: be@tdb.uu.se and be@irfu.se

---

In order to evaluate parallel algorithms for solving the Vlasov equation numerically in multiple dimensions, the algorithm for solving the one-dimensional Vlasov equation numerically has been parallelised. The one-dimensional Vlasov equation leads to a problem in the two-dimensional phase space (x, v), plus time. The parallelisation is performed by domain decomposition to a rectangular processor grid. Derivatives in x space are calculated by a pseudo-spectral method, where FFTs are used to perform discrete Fourier transforms. In velocity v space a Fourier method is used, together with the compact Padé scheme for calculating derivatives, leading to a large number of tri-diagonal linear systems to be solved. The parallelisation of the tri-diagonal systems in the Fourier transformed velocity space can be performed efficiently by the method of domain decomposition. The domain decomposition gives rise to Schur complement systems, which are tri-diagonal, symmetric and strongly diagonally dominant, making it possible to solve these systems with a few Jacobi iterations. Therefore, the parallel efficiency of the semi-implicit Padé scheme is comparable to the parallel efficiency of explicit difference schemes. The parallelisation in x space is less effective due to the FFTs used. The code has been tested on shared memory computers, on clusters of computers, and with the help of the Globus toolkit for communication over the Internet.

KEY WORDS: Vlasov equation; Fourier method; Padé scheme; domain decomposition

1 Introduction

Simulations of the Vlasov equation in multiple dimensions require a large number of sampling points to represent the solution on a numerical grid. The full three-dimensional case has three velocity and three spatial dimensions, plus time, and therefore a numerical algorithm requires a discretised solution in six dimensions to be advanced in time. A likely requirement for a simulation is that the number of sampling points needed in each dimension is of the order 100 to resolve the numerical solution, which gives the total number of sampling points $100^6 = 10^{12}$. At least this amount of double precision numbers has to be stored in a computers memory to be able to advance the solution in time.

One way to reduce the amount of data needed is to use high-order methods, so

that approximations of derivatives et.c. can be made accurately with fewer sampling points. However, a definite limit on an equidistant grid is posed by the sampling theorem, which says that at least two sampling points per wavelength are needed to represent the solution on the grid. Adaptive grids may be used to push this limit further, but that requires the solution to be localised in some sense. If the full solution should be resolved numerically, there is clearly a need for parallel computing.

The present report discusses parallel algorithms which have been developed to solve the one-dimensional Vlasov equation. The optimisations used to speed up the single processor code before the parallelisation, are also discussed. From the investigation of the one-dimensional parallel algorithms, useful conclusions can be drawn for future generalisations of the algorithms to higher dimensions.

2 Parallelisation of the Vlasov code

2.1 Programming language and program structure

The algorithms used for solving the one-dimensional Vlasov equation [3] is implemented in Fortran 90, and the parallelisation of the code is performed by using Message Passing Interface (MPI).

The program has a hierarchy as shown in Fig. 1. The numerical objects lower down in the program hierarchy are called by those higher up. Objects that use Message Passing Interface (MPI) commands for communication are marked with "MPI."

2.2 Optimisation of the code before parallelisation

The numerical algorithms were first implemented in a straightforward manner, and the code was tested to make sure that it was correctly implemented. Thereafter, and before the parallel algorithms were implemented, the single processor code was investigated and optimised. In doing so, the computing time for a given problem was greatly reduced.

2.2.1 Time consuming subroutines

The most time-consuming subroutines were identified with the profiling tool *gprof*. Short descriptions of these subroutines are presented in the following list:

- **RungeKutta_TimeStepper:** Performs a Runge-Kutta time step. It contains a number of loops, which perform few arithmetic operations but which access memory where large two-dimensional arrays are stored.
- **pde_rhs:** Calculates numerically the function values (the right hand side of the semi-discretised Vlasov equation) to be used by the Runge-Kutta algorithm.

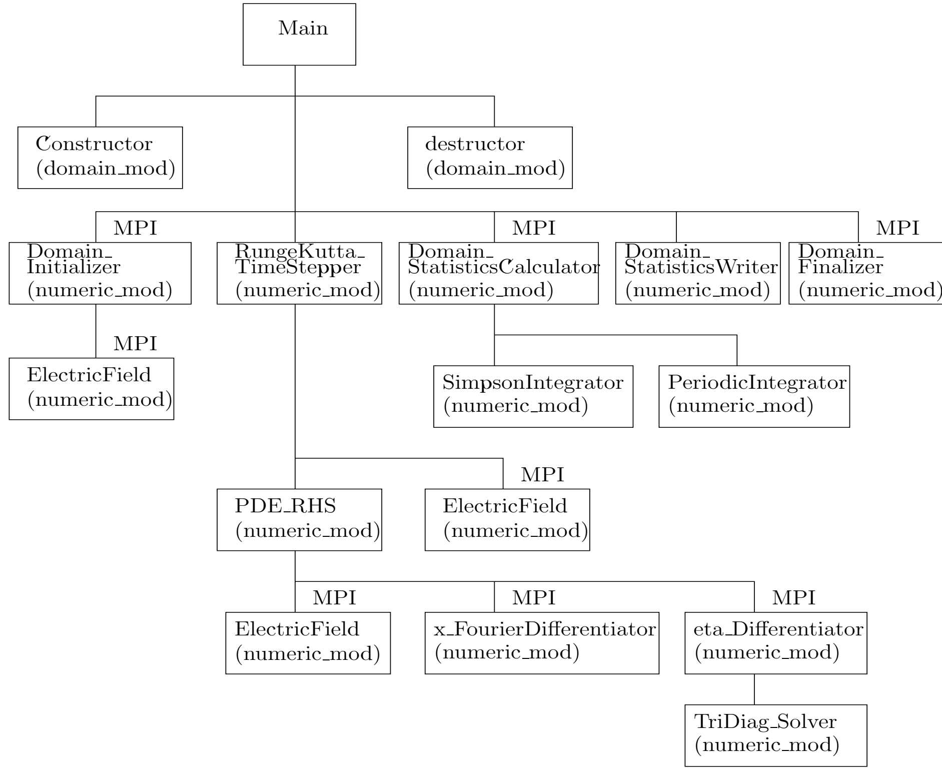

Figure 1: The program hierarchy

eta_Differentiator: Approximates first derivatives with respect to the Fourier variable η, with the Padé difference approximation, where tri-diagonal systems have to be solved; for this task the *TriDiag_Solver* routine is called.

TriDiag_Solver: Solves a small tri-diagonal linear equation system. This routine performs few arithmetic operations but it is called many times.

x_FourierDifferentiator: Approximates the x derivative with a pseudo-spectral method. This is done by using the fast Fourier transform, supplied by the *FFTPACK* package.

The code was compiled with the `+O2` optimisation flag on a single-processor HP workstation. The problem size was set to $N_x \times N_\eta = 100 \times 64$ and the number of time steps $N_t = 800$. The *gprof* utility reported the following execution times for the problem:

Subroutine	Seconds	Milliseconds/call	Calls
pde_rhs	11.34	3.54	3200
RungeKutta_TimeStepper	8.41	10.51	800
TriDiag_Solver	4.41	0.01	652800
eta_Differentiator	3.79	0.59	6400
x_FourierDifferentiator	1.89	0.59	3200

The *time* command gave the total user time 37.5 seconds for this problem.

2.2.2 Optimisation of the most time consuming subroutines

The single-processor code was investigated and optimised according to the most common hints, according to the following list:

RungeKutta_TimeStepper: This routine updates large arrays with new values. The first change was to replace the index notation supported by Fortran 90 by explicit loops over the arrays; this gave some improvement. Instead of updating two arrays in one loop, the loop was split into two loops updating one array each; this also gave some improvement. The third change was to replace the dynamic arrays, created with the `ALLOCATE` command, with static arrays, since the array sizes are not changed dynamically during run-time. This gave a very large improvement of the speed. With these changes, the time used by this routine decreased from 8.41 to 0.96 seconds.

pde_rhs: One division and one multiplication by constants in a loop, were replaced by one multiplication by a constant; this halved the execution time. Some loops were split into two, which gave some improvement. After these changes the time used by the routine decreased from 11.34 to 5.00 seconds.

eta_Differentiator: One division and one multiplication by constants in a loop, were replaced by with one multiplication by a constant. After this change, the time used by the routine decreased from 3.79 to 1.96 seconds.

TriDiag_Solver: No changes were made.

x_FourierDifferentiator: No changes were made.

The code was again compiled with the `+O2` optimisation flag with the problem size $N_x \times N_\eta = 100 \times 64$ and $N_t = 800$, with the following results:

Subroutine	Seconds	Milliseconds/call	Calls
pde_rhs	5.00	1.83	3200
RungeKutta_TimeStepper	0.96	1.70	800
TriDiag_Solver	4.66	0.01	652800
eta_Differentiator	1.96	0.31	6400
x_FourierDifferentiator	1.94	0.61	3200

The *time* command now gave the total user time 23.6 seconds, to be compared with 37.5 seconds for the original code.

2.3 Parallel algorithms

2.3.1 Processor topology

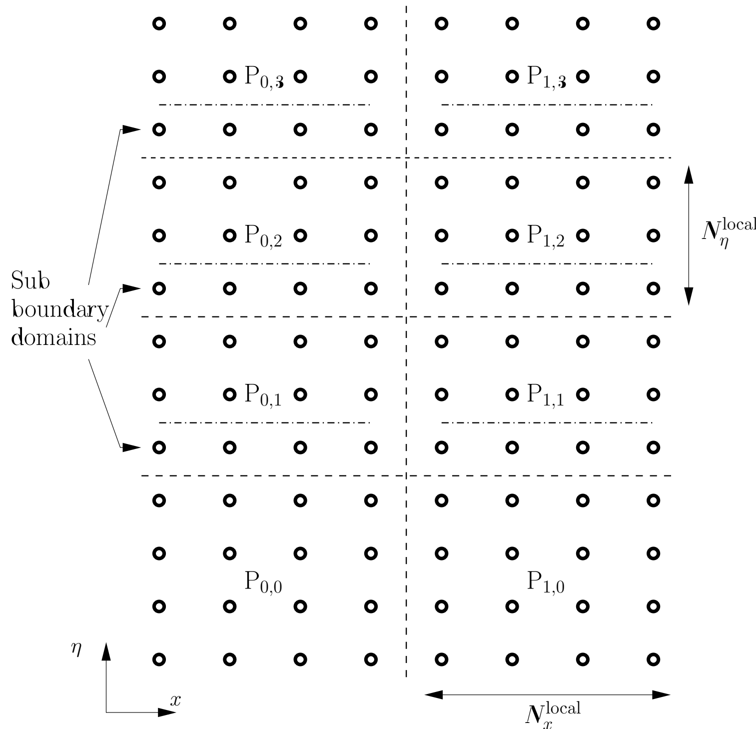

Figure 2: The processor topology; $N_x = 8$, $N_\eta = 12$, $N_{px} = 2$, $N_{p\eta} = 4$.

The computational domain is decomposed into a two-dimensional Cartesian grid of sub domains, where each sub domain is assigned to a processor. Fig. 2 shows a case with 8 processors, where the number of processors in x direction is $N_{px} = 2$ and the number of processors in η direction is $N_{p\eta} = 4$. The circles in Fig. 2 represent sample points of the solution; the total number of intervals in x and η space are $N_x = 8$ and $N_\eta = 12$, respectively. The dashed lines denote the boundaries between the sub domains. Each processor stores the function values of its sub-part of the computational grid. The dash-dotted lines denote inner boundaries, between one-element boundary domains and the rest of the sub-domain; this temporary division of the sub domain is used by the parallel algorithm of the Padé scheme, described in Section 2.3.3 below.

The local number of grid points assigned to each processor in the x and η

directions are N_x^{local} and N_η^{local}, respectively, are given by

$$N_x^{\text{local}} = \frac{N_x}{N_{\text{p}x}} \tag{1}$$

$$N_\eta^{\text{local}} = \frac{N_\eta}{N_{\text{p}\eta}} \tag{2}$$

i.e., function values for $N_x^{\text{local}} \times N_\eta^{\text{local}}$ grid points are assigned to processor $\text{P}_{I,J}$, where $I = 0, \ldots, N_{\text{p}x} - 1$ and $J = 1, \ldots, N_{\text{p}\eta} - 1$. A special case is $J = 0$, for which the number of grid points is $N_x^{\text{local}} \times (N_\eta^{\text{local}} + 1)$; see Fig. 2. In the example given in Fig. 2, the local numbers of grid points in the x and η directions are $N_x^{\text{local}} = 4$ and $N_\eta^{\text{local}} = 3$, respectively.

2.3.2 Domain decomposition in x space

By the partitioning of data in x space, the following subroutines need to perform communication:

pde_rhs: Performs the treatment of the artificial outflow boundary condition in η space, which requires a forward and backward FFT on one vector in x space. This operation is performed serially by processor $\text{P}_{0,N_{\text{p}\eta}-1}$; the communication is performed with the MPI commands `MPI_GATHER` and `MPI_SCATTER`.

ElectricField: Calculates the Electric field by a pseudo-spectral method which requires a forward and backward FFT on one vector in x space, plus a multiplication by a constant. This is performed serially by processor $\text{P}_{0,0}$ and the communication is performed with the MPI commands `MPI_GATHER` and `MPI_SCATTER`.

Domain_StatisticsCalculator Performs statistics of the data. Needs to communicate partial sums to a total sum. The partial sums are sent to processor $\text{P}_{0,0}$ with the `MPI_REDUCE` command.

x_FourierDifferentiator Calculates the x derivative on the whole domain. It does so by using a pseudo-spectral method. It requires a forward and backward FFT on N_η vectors, each of length N_x, and a multiplication with a constant. Instead of parallelising the FFT algorithm or using a parallelised FFT package, the work is partitioned by first distributing one vector to each processor. The processors then perform the forward and inverse FFTs and multiplications by a constant, in parallel, where-after they distribute back the results. This procedure is repeated until all the N_η^{local} vectors has been processed. The communication is performed with the MPI commands `MPI_GATHER` and `MPI_SCATTER`. This part of the code puts the heaviest load on the communication.

2.3.3 Domain decomposition in η space and parallelisation of the compact Padé scheme

For the domain decomposition in η space, the following subroutine is parallelised:

pde_rhs: Calculates the an approximation of the η derivative by the compact Padé scheme.

The compact Padé scheme gives rise to linear tri-diagonal systems which have to be solved. There are N_x such systems, one system for each sampling point in x space.

By using domain decomposition in η space, the tri-diagonal systems are separated into smaller tridiagonal systems which are solved independently on each processor, and tridiagonal Schur complement systems which are solved iteratively in parallel by means of a few Jacobi iterations.

In order to explore the idea, the example with $N_{\mathrm{p}\eta} = 4$, illustrated in Fig. 2, is here described in more detail. The processors are enumerated as $\mathrm{P}_{I,J}$, where $J = 0, 1, 2, 3$ denotes the enumeration of processors in η space, and I denotes the enumeration of processors in x space. The Padé scheme, which is used to calculate the η-derivative of the distribution function, gives rise to N_x identical tridiagonal systems for respectively the real and imaginary parts of the solution.

We study the algorithm for differentiating the real part of the distribution function with respect to η; see Eqs. (80, 82, 84) in [3]. Each of the resulting equation systems can be written in the form

$$\begin{bmatrix} 1 & 0 & & & \cdots & 0 \\ 0 & 4 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 4 & 1 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ & & & 1 & 4 & 1 \\ 0 & & \cdots & 0 & 1 & 0.5 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \\ \vdots \\ \\ v_{N_\eta+1} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \\ \vdots \\ \\ b_{N_\eta+1} \end{bmatrix} \tag{3}$$

where the right-hand side depends on the function to be differentiated and v_j are the approximations of the η derivative. After the domain decomposition into four

sub-domains in η space, the equation system (3) attains the block tri-diagonal form

$$
\left[\begin{array}{ccccccc}
\begin{bmatrix} 1 & 0 & & \cdot\cdot & 0 \\ 0 & 4 & 1 & 0 \cdot\cdot & 0 \\ 0 & 1 & 4 & 1 & \vdots \\ \vdots & & \ddots & \ddots & \ddots\ 0 \\ & & 1 & 4 & 1 \\ 0 & & \cdot\cdot\ 0 & 1 & 4 \end{bmatrix} & \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} & 0 & & \cdots & & 0 \\
\begin{bmatrix} 0 & \cdot\cdot & 0 & 1 \end{bmatrix} & \begin{bmatrix} 4 \end{bmatrix} & \begin{bmatrix} 1 & 0 & \cdot\cdot & 0 \end{bmatrix} & & & & \\
0 & \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} & \begin{bmatrix} 4 & 1 & 0 \cdot\cdot & 0 \\ 1 & 4 & 1 & \vdots \\ \vdots & \ddots & \ddots & \ddots\ 0 \\ & 1 & 4 & 1 \\ 0 \cdot\cdot & 0 & 1 & 4 \end{bmatrix} & \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} & \ddots & & \vdots \\
 & & \begin{bmatrix} 0 & \cdot\cdot & 0 & 1 \end{bmatrix} & \begin{bmatrix} 4 \end{bmatrix} & \begin{bmatrix} 1 & 0 & \cdot\cdot & 0 \end{bmatrix} & & \\
\vdots & & \ddots & \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} & \begin{bmatrix} 4 & 1 & 0 \cdot\cdot & 0 \\ 1 & 4 & 1 & \vdots \\ \vdots & \ddots & \ddots & \ddots\ 0 \\ & 1 & 4 & 1 \\ 0 \cdot\cdot & 0 & 1 & 4 \end{bmatrix} & \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} & 0 \\
 & & & & \begin{bmatrix} 0 & \cdot\cdot & 0 & 1 \end{bmatrix} & \begin{bmatrix} 4 \end{bmatrix} & \begin{bmatrix} 1 & 0 & \cdot\cdot & 0 \end{bmatrix} \\
0 & & \cdots & & 0 & \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} & \begin{bmatrix} 4 & 1 & 0 \cdot\cdot & 0 \\ 1 & 4 & 1 & \vdots \\ \vdots & \ddots & \ddots & \ddots\ 0 \\ & 1 & 4 & 1 \\ 0 \cdot\cdot & 0 & 1 & 0.5 \end{bmatrix}
\end{array}\right]
\begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ V_6 \\ V_7 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \\ B_5 \\ B_6 \\ B_7 \end{bmatrix} \tag{4}
$$

which can be written as

$$
\begin{bmatrix}
A_{11} & A_{12} & 0 & & \cdots & & 0 \\
A_{12}^T & A_{22} & A_{23} & & & & \\
0 & A_{23}^T & A_{33} & A_{34} & \ddots & & \vdots \\
 & & A_{34}^T & A_{44} & A_{45} & & \\
\vdots & & \ddots & A_{45}^T & A_{55} & A_{56} & 0 \\
 & & & & A_{56}^T & A_{66} & A_{67} \\
0 & & \cdots & & 0 & A_{67}^T & A_{77}
\end{bmatrix}
\begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ V_6 \\ V_7 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \\ B_5 \\ B_6 \\ B_7 \end{bmatrix} \tag{5}
$$

where the superscript T denotes the transpose of matrices, and where matrices of sizes $(N_\eta^{\mathrm{local}}+1)\times(N_\eta^{\mathrm{local}}+1)$ and $(N_\eta^{\mathrm{local}}+1)\times 1$ are

$$
A_{11} = \begin{bmatrix}
1 & 0 & & & \cdots & 0 \\
0 & 4 & 1 & 0 & \cdots & 0 \\
0 & 1 & 4 & 1 & & \vdots \\
\vdots & & \ddots & \ddots & \ddots & 0 \\
 & & & 1 & 4 & 1 \\
0 & & \cdots & 0 & 1 & 4
\end{bmatrix} \quad
A_{12} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \tag{6}
$$

$$
V_1 = \begin{bmatrix} v_{1,1} \\ v_{1,2} \\ \vdots \\ v_{1,N_\eta^{\mathrm{local}}+1} \end{bmatrix} \quad
B_1 = \begin{bmatrix} b_{1,1} \\ b_{1,2} \\ \vdots \\ b_{1,N_\eta^{\mathrm{local}}+1} \end{bmatrix} \tag{7}
$$

respectively, matrices of sizes 1×1 and $1\times(N_\eta^{\mathrm{local}}-1)$ are

$$
A_{22} = A_{44} = A_{66} = \begin{bmatrix} 4 \end{bmatrix} \qquad A_{23} = A_{45} = A_{67} = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix} \tag{8}
$$

$$
V_j = \begin{bmatrix} v_{j,1} \end{bmatrix} \qquad B_j = \begin{bmatrix} b_{j,1} \end{bmatrix} \qquad j = 2, 4, 6 \tag{9}
$$

respectively, and matrices of sizes $(N_\eta^{\text{local}}-1)\times(N_\eta^{\text{local}}-1)$ and $(N_\eta^{\text{local}}-1)\times 1$ are

$$A_{33}=A_{55}=\begin{bmatrix}4 & 1 & 0 & \cdots & 0\\ 1 & 4 & 1 & & \vdots\\ \vdots & \ddots & \ddots & \ddots & 0\\ & & 1 & 4 & 1\\ 0 & \cdots & 0 & 1 & 4\end{bmatrix}\quad A_{34}=A_{56}=\begin{bmatrix}0\\ \\ \vdots\\ \\ 0\\ 1\end{bmatrix} \tag{10}$$

$$A_{77}=\begin{bmatrix}4 & 1 & 0 & \cdots & 0\\ 1 & 4 & 1 & & \vdots\\ \vdots & \ddots & \ddots & \ddots & 0\\ & & 1 & 4 & 1\\ 0 & \cdots & 0 & 1 & 0.5\end{bmatrix} \tag{11}$$

$$V_j=\begin{bmatrix}v_{j,1}\\ v_{j,2}\\ \\ \vdots\\ \\ v_{j,N_\eta^{\text{local}}-1}\end{bmatrix}\quad B_j=\begin{bmatrix}b_{j,1}\\ b_{j,2}\\ \\ \vdots\\ \\ b_{j,N_\eta^{\text{local}}-1}\end{bmatrix}\quad j=3,5,7 \tag{12}$$

respectively.

The block-tridiagonal system (5) is separated into the systems

$$A_{11}V_1+A_{12}V_2=B_1 \tag{13}$$
$$A_{23}^TV_2+A_{33}V_3+A_{34}V_4=B_3 \tag{14}$$
$$A_{45}^TV_4+A_{55}V_5+A_{56}V_6=B_5 \tag{15}$$
$$A_{67}^TV_6+A_{77}V_7=B_7 \tag{16}$$

and

$$A_{12}^TV_1+A_{22}V_2+A_{23}V_3=B_2 \tag{17}$$
$$A_{34}^TV_3+A_{44}V_4+A_{45}V_5=B_4 \tag{18}$$
$$A_{56}^TV_5+A_{66}V_6+A_{67}V_7=B_6 \tag{19}$$

respectively.

By solving Eqs. (13–16) for V_1, V_3, V_5 and V_7, respectively,

$$V_1=A_{11}^{-1}(B_1-A_{12}V_2) \tag{20}$$
$$V_3=A_{33}^{-1}(B_3-A_{23}^TV_2-A_{34}V_4) \tag{21}$$
$$V_5=A_{55}^{-1}(B_5-A_{45}^TV_4-A_{56}V_6) \tag{22}$$
$$V_7=A_{77}^{-1}(B_7-A_{67}^TV_6) \tag{23}$$

and inserting the result into Eqs. (17–19), one obtains the resulting tri-diagonal Schur complement system

$$\begin{bmatrix} c_{11} & c_{12} & 0 \\ c_{21} & c_{22} & c_{23} \\ 0 & c_{32} & c_{33} \end{bmatrix} \begin{bmatrix} V_2 \\ V_4 \\ V_6 \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \tag{24}$$

where the matrix elements of the Schur matrix C are

$$c_{11} = -A_{12}^T A_{11}^{-1} A_{12} + A_{22} - A_{23} A_{33}^{-1} A_{23}^T \tag{25}$$
$$c_{22} = -A_{34}^T A_{33}^{-1} A_{34} + A_{44} - A_{45} A_{55}^{-1} A_{45}^T \tag{26}$$
$$c_{33} = -A_{56}^T A_{55}^{-1} A_{56} + A_{66} - A_{67} A_{77}^{-1} A_{67}^T \tag{27}$$
$$c_{12} = -A_{23} A_{33}^{-1} A_{34} \tag{28}$$
$$c_{21} = -A_{34}^T A_{33}^{-1} A_{23}^T = c_{12}^T = c_{12} \tag{29}$$
$$c_{23} = -A_{45} A_{55}^{-1} A_{56} \tag{30}$$
$$c_{32} = -A_{56}^T A_{55}^{-1} A_{45}^T = c_{23}^T = c_{23} \tag{31}$$

and the components of the right hand side

$$u_1 = B_2 - A_{12}^T A_{11}^{-1} B_1 - A_{23} A_{33}^{-1} B_3 \tag{32}$$
$$u_2 = B_4 - A_{34}^T A_{33}^{-1} B_3 - A_{45} A_{55}^{-1} B_5 \tag{33}$$
$$u_3 = B_6 - A_{56}^T A_{55}^{-1} B_5 - A_{67} A_{77}^{-1} B_7 \tag{34}$$

respectively.

The data is partitioned between the processors $\mathrm{P}_{I,0}$–$\mathrm{P}_{I,3}$ as

$$B_1 \rightarrow \mathrm{P}_{I,0} \tag{35}$$
$$B_2, B_3 \rightarrow \mathrm{P}_{I,1} \tag{36}$$
$$B_4, B_4 \rightarrow \mathrm{P}_{I,2} \tag{37}$$
$$B_6, B_7 \rightarrow \mathrm{P}_{I,3} \tag{38}$$

and similarly, the unknowns are partitioned as

$$V_1 \rightarrow \mathrm{P}_{I,0} \tag{39}$$
$$V_2, V_3 \rightarrow \mathrm{P}_{I,1} \tag{40}$$
$$V_4, V_4 \rightarrow \mathrm{P}_{I,2} \tag{41}$$
$$V_6, V_7 \rightarrow \mathrm{P}_{I,3} \tag{42}$$

and the matrices are partitioned between the processors as

$$A_{11}, A_{12}, A_{22} \rightarrow \mathrm{P}_{I,0} \tag{43}$$
$$A_{33}, A_{23}, A_{22}, A_{44}, A_{34} \rightarrow \mathrm{P}_{I,1} \tag{44}$$
$$A_{55}, A_{45}, A_{44}, A_{56}, A_{66} \rightarrow \mathrm{P}_{I,2} \tag{45}$$
$$A_{77}, A_{67}, A_{66} \rightarrow \mathrm{P}_{I,3} \tag{46}$$

The matrices A_{11}, A_{33}, and A_{77} are stored in an LU factorised, banded format, i.e., only non-zero values are stored.

The processors calculates, in parallel, the following quantities needed to form the matrix of the Schur complement system:

- $P_{I,0}$ calculates the quantity $x_{12} = A_{11}^{-1} A_{12}$ and obtains the term $d_{12} = A_{12}^T A_{11}^{-1} A_{12} = (x_{12})_{N_\eta^{\text{local}}}$ by evaluation.
- $P_{I,1}$ calculates the quantity $x_{34} = A_{33}^{-1} A_{34}$ and obtains the term $d_{34} = A_{34}^T A_{33}^{-1} A_{34} = (x_{34})_{N_\eta^{\text{local}}-1}$ by evaluation. $P_{I,1}$ also calculates the quantity $x_{23} = A_{33}^{-1} A_{23}^T$ and obtains the terms $d_{23} = A_{23} A_{33}^{-1} A_{23}^T = (x_{23})_1$ and $d_{24} = A_{34}^T A_{33}^{-1} A_{23}^T = A_{23} A_{33}^{-1} A_{34} = (x_{23})_{N_\eta^{\text{local}}-1}$ by evaluation.
- $P_{I,2}$ calculates the quantity $x_{56} = A_{55}^{-1} A_{56}$ and obtains the term $d_{56} = A_{56}^T A_{55}^{-1} A_{56} = (x_{56})_{N_\eta^{\text{local}}-1}$ by evaluation. $P_{I,2}$ also calculates the quantity $x_{45} = A_{55}^{-1} A_{45}^T$ and obtains the terms $d_{45} = A_{45} A_{55}^{-1} A_{45}^T = (x_{45})_1$ and $d_{46} = A_{56}^T A_{55}^{-1} A_{45}^T = A_{45} A_{55}^{-1} A_{56} = (x_{45})_{N_\eta^{\text{local}}-1}$ by evaluation.
- $P_{I,3}$ calculates the quantity $x_{76} = A_{77}^{-1} A_{67}^T$ and obtains the term $d_{67} = A_{67}^T A_{77}^{-1} A_{67}^T = (x_{67})_1$ by evaluation.

The terms in the right hand side (32–34) of the Schur complement system can now be calculated, in parallel, as scalar products with the above calculated quantities:

- $P_{I,0}$ calculates the term $y_{12} = A_{12}^T A_{11}^{-1} B_1 = (A_{11}^{-1} A_{12})^T B_1 = x_{12}^T B_1$.
- $P_{I,1}$ calculates the terms $y_{34} = A_{34}^T A_{33}^{-1} B_3 = (A_{33}^{-1} A_{34})^T B_3 = x_{34}^T B_3$ and $y_{32} = A_{23} A_{33}^{-1} B_3 = (A_{33}^{-1} A_{23}^T)^T B_3 = x_{23}^T B_3$.
- $P_{I,2}$ calculates the terms $y_{56} = A_{56}^T A_{55}^{-1} B_5 = (A_{55}^{-1} A_{56})^T B_5 = x_{56}^T B_5$ and $y_{54} = A_{45} A_{55}^{-1} B_5 = (A_{55}^{-1} A_{45}^T)^T B_5 = x_{45}^T B_5$.
- $P_{I,3}$ calculates the term $y_{76} = A_{67} A_{77}^{-1} B_7 = (A_{77}^{-1} A_{67}^T)^T B_7 = x_{67}^T B_7$.

In terms of the above quantities, the components of the Schur complement matrix are

$$c_{11} = -d_{12} + A_{22} - d_{23} \tag{47}$$

$$c_{22} = -d_{34} + A_{44} - d_{45} \tag{48}$$

$$c_{33} = -d_{56} + A_{66} - d_{67} \tag{49}$$

$$c_{12} = c_{21} = -d_{24} \tag{50}$$

$$c_{23} = c_{32} = -d_{46} \tag{51}$$

and the components of the right hand side are

$$u_1 = B_2 - y_{12} - y_{32} \tag{52}$$

$$u_2 = B_4 - y_{34} - y_{54} \tag{53}$$

$$u_3 = B_6 - y_{56} - y_{76} \tag{54}$$

2.3.4 Solution of the Schur complement system

N_η^{local}	c_{11}	c_{22}	c_{33}	c_{12}, c_{21}, c_{23}, c_{32}	$\lvert c_{23}/c_{33}\rvert$	N_{Jacobi}
2	3.48	3.50	1.75	-2.50×10^{-1}	1.43×10^{-1}	12
3	3.47	3.47	3.23	6.67×10^{-2}	2.06×10^{-2}	6
4	3.46	3.46	3.45	-1.79×10^{-2}	5.18×10^{-3}	4
5	3.46	3.46	3.46	4.78×10^{-3}	1.38×10^{-3}	3
6	3.46	3.46	3.46	-1.28×10^{-3}	3.70×10^{-4}	2
7	3.46	3.46	3.46	3.44×10^{-4}	9.92×10^{-5}	2
8	3.46	3.46	3.46	-9.20×10^{-5}	2.66×10^{-5}	2
9	3.46	3.46	3.46	2.47×10^{-5}	7.12×10^{-6}	1
10	3.46	3.46	3.46	-6.61×10^{-6}	1.91×10^{-6}	1
11	3.46	3.46	3.46	1.77×10^{-6}	5.11×10^{-7}	1
12	3.46	3.46	3.46	-4.74×10^{-7}	1.37×10^{-7}	1
13	3.46	3.46	3.46	1.27×10^{-7}	3.67×10^{-8}	1
14	3.46	3.46	3.46	-3.41×10^{-8}	9.83×10^{-9}	1
15	3.46	3.46	3.46	9.13×10^{-9}	2.64×10^{-9}	1
16	3.46	3.46	3.46	-2.45×10^{-9}	7.06×10^{-10}	1
17	3.46	3.46	3.46	6.55×10^{-10}	1.89×10^{-10}	1
18	3.46	3.46	3.46	-1.76×10^{-10}	5.07×10^{-11}	0

Table 1: Values of non-zero elements of the 3×3 Schur complement matrix, and the number of Jacobi iterations used to solve the Schur complement system.

The Schur matrix is tri-diagonal, symmetric and strongly diagonally dominant, where the absolute values of the off-diagonal elements decrease exponentially with the number of grid points N_η^{local}, separating the sub-boundaries. Therefore, the system is solved efficiently and in parallel by a few Jacobi iterations. Table 1 shows the values of the elements of C as a function of the number of grid points N_η^{local}. It also shows the number of Jacobi iterations N_{Jacobi} used in the program, giving an accuracy of about 10 digits. For $N_\eta^{\text{local}} \geq 18$, no Jacobi iterations are performed (i.e., the steps 1 and 2 of the Jacobi algorithm described below, are skipped), but the solution to the Schur complement system is found by dividing the right hand side with the diagonal elements of the Schur matrix; see the steps (55–57) below.

The cases with different numbers of sub-domains converge at a similar rate as the special case above. Therefore the same number of Jacobi iterations N_{Jacobi} are used for all cases.

The Jacobi iterations needed to solve the Schur complement system (24) are performed by processor $\text{P}_{I,1}$, $\text{P}_{I,2}$ and $\text{P}_{I,3}$, solving for V_2, V_4 and V_6, respectively. Before the Jacobi iterations begin, processor $\text{P}_{I,1}$–$\text{P}_{I,3}$ must have the necessary components of the Schur system. Therefore the following communication

is needed:

$$\begin{aligned}
&\mathrm{P}_{I,0} \text{ sends } d_{12} \text{ and } y_{12} \text{ to } \mathrm{P}_{I,1}.\\
&\mathrm{P}_{I,1} \text{ sends } d_{34},\, d_{24} \text{ and } y_{34} \text{ to } \mathrm{P}_{I,2}.\\
&\mathrm{P}_{I,2} \text{ sends } d_{56},\, d_{46} \text{ and } y_{56} \text{ to } \mathrm{P}_{I,3}.
\end{aligned}$$

The variables $d_{12}, \ldots, d_{46}$ do not depend on the solution, and they therefore only have to be communicated once, and the same numbers can be used for all N_{px} systems. The values on y_{12}, y_{34} and y_{56} do, on the other hand, depend on the solution, and therefore they have to be communicated in each time sub-step of the Runge-Kutta algorithm, and there are one N_{px} values each to communicate, one per equation system.

Before the Jacobi iterations begin, the solution is initially set to

$$V_2^{(\text{old})} \leftarrow u_1/c_{11} \tag{55}$$

$$V_4^{(\text{old})} \leftarrow u_2/c_{22} \tag{56}$$

$$V_6^{(\text{old})} \leftarrow u_3/c_{33} \tag{57}$$

Thereafter, one Jacobi iteration is performed as

1. A communication step:

$$\begin{aligned}
&\mathrm{P}_{I,1} \text{ sends } V_2^{(\text{old})} \text{ to } \mathrm{P}_{I,2}.\\
&\mathrm{P}_{I,2} \text{ sends } V_4^{(\text{old})} \text{ to } \mathrm{P}_{I,1} \text{ and } \mathrm{P}_{I,3}.\\
&\mathrm{P}_{I,3} \text{ sends } V_6^{(\text{old})} \text{ to } \mathrm{P}_{I,2}.
\end{aligned}$$

2. A calculation step, performed in parallel:

$$\begin{aligned}
\mathrm{P}_{I,1}: &\quad V_2^{(\text{new})} \leftarrow (u_1 - c_{12}V_4^{(\text{old})})/c_{11}\\
\mathrm{P}_{I,2}: &\quad V_4^{(\text{new})} \leftarrow (u_2 - c_{21}V_2^{(\text{old})} - c_{23}V_6^{(\text{old})})/c_{22}\\
\mathrm{P}_{I,3}: &\quad V_6^{(\text{new})} \leftarrow (u_3 - c_{32}V_4^{(\text{old})})/c_{33}\\
\mathrm{P}_{I,1}: &\quad V_2^{(\text{old})} \leftarrow V_2^{(\text{new})}\\
\mathrm{P}_{I,2}: &\quad V_4^{(\text{old})} \leftarrow V_4^{(\text{new})}\\
\mathrm{P}_{I,3}: &\quad V_6^{(\text{old})} \leftarrow V_6^{(\text{new})}
\end{aligned}$$

Step 1 and 2 are repeated N_{Jacobi} times.

After the Jacobi algorithm is finished, V_2, V_4 and V_6 can be used to obtain the rest of the solution by Eqs. (20–23), which is performed in two steps:

1. A communication step:

$$\begin{aligned}
&\mathrm{P}_{I,3} \text{ sends } V_6 \text{ to } \mathrm{P}_{I,2}.\\
&\mathrm{P}_{I,2} \text{ sends } V_4 \text{ to } \mathrm{P}_{I,1}.\\
&\mathrm{P}_{I,1} \text{ sends } V_2 \text{ to } \mathrm{P}_{I,0}.
\end{aligned}$$

2. A calculation step where linear, tridiagonal equation systems are solved, performed in parallel:

$$\begin{aligned}
&\mathrm{P}_{I,0} \text{ solves } A_{11}V_1 = B_1 - A_{12}V_2 \text{ to obtain } V_1.\\
&\mathrm{P}_{I,1} \text{ solves } A_{33}V_3 = B_3 - A_{23}^T V_2 - A_{34}V_4 \text{ to obtain } V_3.\\
&\mathrm{P}_{I,2} \text{ solves } A_{55}V_5 = B_5 - A_{45}^T V_4 - A_{56}V_6 \text{ to obtain } V_5.\\
&\mathrm{P}_{I,3} \text{ solves } A_{77}V_7 = B_7 - A_{67}^T V_6 \text{ to obtain } V_7.
\end{aligned} \tag{58}$$

After these steps, the approximations of the η derivative of the real part of the distribution function are obtained in the variables V_1, ..., V_7. These steps have to be repeated for the imaginary part of the function, which give rise to a slightly different equation system due to different boundary condition at the symmetry boundary $\eta = 0$; see [3].

2.4 Performance model

A precise performance model can only be made if one has the knowledge on how the computer system is constructed and how the communication between processors is performed in detail. The performance model presented here is very simple and does not take into account the differences in computer systems; it only contains the very basic elements of a performance model. Therefore one can at best expect the real computation times and speedups to follow the model qualitatively, not quantitatively. For a more detailed discussion about performance models for different computer designs, see for example the book by Kumar et al. [5].

A basic model is that the total execution time T can be divided into three parts

$$T = T_{\text{comp}} + T_{\text{msg}} + T_{\text{idle}} \tag{59}$$

where T_{comp}, T_{msg} and T_{idle} is the *computation time*, *messaging time* and *idle time*, respectively.

One of the most interesting aspect of parallel computing is, except for the total executing time, how large the *relative speedup*

$$S = \frac{T(1,1)}{T(N_{\text{p}x}, N_{\text{p}\eta})} \tag{60}$$

is when using more than one processor to do calculations. The time $T(1,1)$ is the time needed for one processor to solve a given problem, and $T(N_{\text{p}x}, N_{\text{p}\eta})$ is the time needed for $N_{\text{p}x} \times N_{\text{p}\eta}$ processors to solve the same problem. One can assume that the messaging time T_{msg} and idle time T_{idle} are equal to zero for the single processor case, since no communication occurs; this gives rise to the expression

$$S = \frac{T_{\text{comp}}(1,1)}{T_{\text{comp}}(N_{\text{p}x}, N_{\text{p}\eta}) + T_{\text{msg}}(N_{\text{p}x}, N_{\text{p}\eta}) + T_{\text{idle}}(N_{\text{p}x}, N_{\text{p}\eta})} \tag{61}$$

for the relative speedup.

The idle time $T_{\rm idle}$ is the time the processor has to wait when another processor is doing work. This can happen due to bad load balance or when a purely serial part of an algorithm is processed by a processor.

For a symmetric multi processor machine, the computation time can simply be assumed to be inversely proportional to the number of processors

$$T_{\rm comp}(N_{{\rm p}x}, N_{{\rm p}\eta}) = \frac{T_{\rm comp}(1,1)}{N_{{\rm p}x}N_{{\rm p}\eta}} \tag{62}$$

It is neglected that the parallel algorithm may add extra calculations not present in the single processor algorithm; for example, in the present problem, the domain decomposition in η space introduces extra calculations such as the Jacobi iterations used to solve Schur complement systems, but for most problems this amount of extra calculations is negligible compared to the rest of the calculations. There are also a few serial parts of the algorithm which are neglected: The treatment of the boundary conditions in η space and the calculation of the electric field E requires two FFTs each on a vector of length N_x, and are performed serially by one processor.

Some conclusions can be drawn at this early stage: The resulting expression for the speedup, after using Eq. (62) in Eq. (61), is

$$S = \frac{N_{{\rm p}x}N_{{\rm p}\eta}}{1 + [T_{\rm msg}(N_{{\rm p}x}, N_{{\rm p}\eta}) + T_{\rm idle}(N_{{\rm p}x}, N_{{\rm p}\eta})]/T_{\rm comp}(N_{{\rm p}x}, N_{{\rm p}\eta})} \tag{63}$$

If $T_{\rm msg}$ and $T_{\rm idle}$ are negligible compared with $T_{\rm comp}$, then the speedup is linear, $S = N_{{\rm p}x}N_{{\rm p}\eta}$. However, since $T_{\rm comp}(N_{{\rm p}x}, N_{{\rm p}\eta})$ is decreasing with a growing number of processors according to Eq. (62), the denominator of Eq. (63) may grow much larger than unity and ruin the linear speedup.

In the analysis of the computation time it is convenient to used the *problem size* $M_{\rm tot}$, which is the total number of double precision numbers produced for a problem, including all time steps. It is given approximately by

$$M_{\rm tot} = 2N_t N_\eta N_x \tag{64}$$

where N_t, N_η and N_x are the number of grid points in the time, η and x space, respectively. The factor 2 comes from that the solution is complex-valued, where the real and imaginary parts of the solution give rise to one double precision number each per sampling point.

For the one-processor algorithm, the computation time is

$$T_{\rm comp}(1,1) = \tau_{\rm f} N_{\rm f} \tag{65}$$

and, by Eq. (62), for the multi processor algorithm,

$$T_{\rm comp}(N_{{\rm p}x}, N_{{\rm p}\eta}) = \frac{\tau_{\rm f} N_{\rm f}}{N_{{\rm p}x}N_{{\rm p}\eta}} \tag{66}$$

where τ_f is the time to perform one floating point operation and N_f is the total number of floating point operations needed. It is here neglected that different types of floating point operations take different times to perform; for example, *divisions* by numbers often take longer times than *additions*, *subtractions* and *multiplications*. The number of floating point operations is estimated as follows:

- The Runge-Kutta algorithm needs $7M_\text{tot}$ additions and multiplications, which gives a total of $14M_\text{tot}$ floating point operations.
- In the Runge-Kutta algorithm, four steps of calculation of function values are performed, each step including:
  - The approximation of the x derivative is performed with the pseudo-spectral method, which requires one forward and one backward FFT, and one multiplication by a constant. The FFTs are performed on vectors of length N_x, which requires approximately $5N_x \log_2 N_x$ arithmetic operations per FFT. There are N_η such vectors in the domain and N_t time steps, which gives the number of operations $2N_\eta N_t(5N_x \log_2 N_x) = 5M_\text{tot} \log_2 N_x$ operations for the FFTs. The multiplication by a constant requires M_tot operation. The total is thus $5M_\text{tot} \log_2 N_x + M_\text{tot}$ arithmetic operations.
  - The approximation of the η derivative is performed with the Padé scheme. It requires the solution of linear tri-diagonal systems. The calculation of the right hand sides of the systems includes one subtraction and one multiplication for each complex valued function values, giving $2M_\text{tot}$ arithmetic operations. The solution of an equation system of size N requires in general $8N$ operations, where $3N$ is needed for the LU factorisation and $5N$ is needed for the back substitution. In the present algorithm, the LU factorisation is performed once and the result is reused; therefore only $5N$ operation is needed per system, giving $5M_\text{tot}$ operations for the present problem. The total is $7M_\text{tot}$ operations.
  - The summation of the terms in the Vlasov equation requires one addition and one multiplication per function value, which gives $2M_\text{tot}$ operations.
  - The numerical dissipation is performed by a centred scheme which approximates the sixth derivative of the function, which requires $10M_\text{tot}$ operations.

  The number of arithmetic operations needed for the calculation of function values is thus $4(5M_\text{tot} \log_2 N_x + 20M_\text{tot})$.

From the above estimates, the total number of arithmetic operations is

$$N_\text{f} = 20M_\text{tot} \log_2 N_x + 94M_\text{tot} = (20 \log_2 N_x + 94)M_\text{tot} \qquad (67)$$

and the time needed to perform these operation, by Eq.(66), is

$$T_{\text{comp}}(N_{\text{p}x}, N_{\text{p}\eta}) = \frac{\tau_{\text{f}}}{N_{\text{p}x}N_{\text{p}\eta}}(20\log_2 N_x + 94)M_{\text{tot}} \tag{68}$$

The messaging time can be divided into two parts,

$$T_{\text{msg}} = \tau_{\text{s}}N_{\text{s}} + \tau_{\text{d}}N_{\text{d}} \tag{69}$$

where τ_{s} is the *startup time* (or *latency*) for a message, N_{s} is the *number of messages to send per processor*, and τ_{d} is the *time to send one double precision number* and N_{d} is the *total number of double precision numbers to send per processor*. The importance of these terms may vary with different computer systems: On a shared memory machine, the total startup time $\tau_{\text{s}}N_{\text{s}}$ is often negligible compared to the sending time $\tau_{\text{d}}N_{\text{d}}$. On a cluster of computers, however, the startup time τ_{s} may be very large, so that it is very important to minimise the number of messages N_{s}.

The number of messages N_{s} and data N_{d} sent by each processor can be estimated as follows:

- In each of the four Runge-Kutta sub-steps, the calculation of the function values requires the following messages to be sent
  - In the calculation of the x derivative, vectors for each x value are first distributed to each of the processors in the processor row. Then FFTs and multiplications are performed in parallel, where-after the resulting vectors are re-distributed. The number of messages to send for each processor is approximately $2(N_{\text{p}x} - 1)N_\eta^{\text{local}}N_t$, each message containing N_x^{local} complex numbers, which gives that $4(N_{\text{p}x} - 1)N_\eta^{\text{local}}N_tN_x^{\text{local}} = 2M_{\text{tot}}(N_{\text{p}x} - 1)/(N_{\text{p}x}N_{\text{p}\eta})$ double precision (real) numbers have to be sent.
  - In the calculation of the η derivative, communication of vectors have to be performed between nearest neighbours in the η direction. First each processor has to send vectors of length $N_{\text{p}x}$ to its two neighbours, in order to form the right hand sides B_j which contain centred difference approximations of the η derivatives. Then the vectors y_{ij}, of length N_x^{local}, have to be communicated to the nearest upper neighbour. In the Jacobi iterations, two vectors of length N_x^{local} have to be communicated in each Jacobi step. Finally, a vector of length N_x^{local} has to be communicated to the nearest lower neighbour. This gives $(4 + 2N_{\text{Jacobi}})N_t$ messages, and the number of double precision numbers $2(4 + 2N_{\text{Jacobi}})N_tN_x^{\text{local}}$.

The above estimates give the total number of messages, for the four Runge-Kutta sub-steps, as

$$\begin{aligned} N_{\text{s}} &= 4[2(N_{\text{p}x} - 1)N_\eta^{\text{local}}N_t + (4 + 2N_{\text{Jacobi}})N_t] \\ &= 8(N_{\text{p}x} - 1)N_\eta^{\text{local}}N_t + (16 + 8N_{\text{Jacobi}})N_t \end{aligned} \tag{70}$$

and the amount of double precision numbers

$$
\begin{aligned}
N_{\mathrm{d}} &= 4\left[2\frac{M_{\mathrm{tot}}(N_{\mathrm{p}x}-1)}{N_{\mathrm{p}x}N_{\mathrm{p}\eta}} + 2(4+2N_{\mathrm{Jacobi}})N_t N_x^{\mathrm{local}}\right] \\
&= 8\frac{M_{\mathrm{tot}}}{N_{\mathrm{p}\eta}}\left(1-\frac{1}{N_{\mathrm{p}x}}\right) + (32+16N_{\mathrm{Jacobi}})N_t N_x^{\mathrm{local}}
\end{aligned}
\tag{71}
$$

The resulting expression for the messaging time is

$$
\begin{aligned}
T_{\mathrm{msg}} &= \tau_{\mathrm{s}}[8(N_{\mathrm{p}x}-1)N_\eta^{\mathrm{local}}N_t + 20N_t] \\
&\quad + \tau_{\mathrm{d}}\left[8\frac{M_{\mathrm{tot}}}{N_{\mathrm{p}\eta}}\left(1-\frac{1}{N_{\mathrm{p}x}}\right) + (32+16N_{\mathrm{Jacobi}})N_t N_x^{\mathrm{local}}\right]
\end{aligned}
\tag{72}
$$

One can note that an increase of the number of processors $N_{\mathrm{p}x}$ in x space gives a substantial increase in both the number of messages and in the data to communicate, with a resulting increase of the messaging time T_{msg}. Thus the domain decomposition in x space is not very efficient, and the analysis gives an explanation of the poor results (slowdown instead of speedup) on computer grids [1]. The messaging time is, on the other hand, independent of the number of processors in η space, where only nearest neighbour communications are performed between processors. Therefore, the parallelisation in η space can be performed with a good efficiency, which is comparable with the efficiency of parallel algorithms for explicit difference schemes, despite the fact that tri-diagonal schemes have to be parallelised in the present algorithm.

2.5 Shared memory computers and computational grids

An effort has been put on making the Vlasov code portable on different computer systems. Therefore machine-dependent code is avoided and the only mathematical library used, for performing FFTs, is FFTPACK, which is freely downloadable from the Web site *www.netlib.org*. The parallelisation of the code is being performed by Message Passing Interface (MPI), which makes the parallel code portable; it can be run both on shared memory and distributed memory machines.

The Vlasov code has been compiled and tested on different brands of parallel computers, such as the Hewlett-Packard parallel computer at Dept. of Astronomy and Space Physics and the Sun parallel computers at Dept. of Scientific Computing [2], and on the IBM SP2 system at the Royal Institute of Technology (KTH).

As a part of a students project [1] at the Department of Scientific Computing, the code has also been tested with MPI between separate HP workstations at the Institute of Space physics, and on a Linux PC cluster at Dept. of Scientific computing. Initial tests have also been performed to run the code with the help of the Globus Toolkit, in which the PC cluster was connected with a HP cluster over the Internet.

2.6 Computational grids and the Globus project

A project group [1] has made extensive performance tests on different computer systems, and has put a large effort on investigating the procedure to set up a working Globus environment for running the parallelised code between different clusters.

The main experiences and conclusions reported by the group are:

- The communication times are large on computational grids. It is still possible to gain speedup by:
  - Minimise the number of communication events, and thereby avoiding the large startup times needed. Especially when using Globus over the Internet, the startup time is extremely large.
  - Hide the communication with calculations by using non-blocking communication, i.e., while data is transfered, calculations should be performed.
- When using asymmetric clusters to perform computations easily gives rise to unbalanced work load and large idle times. It is important to balance the amount of work so that the faster computers do not have to wait for the slower computers to complete work.
- The spirit of the Globus project is that much of the difficulties with security et.c., should be hidden for the user. In practice, it is still necessary to know many details, for example:
  - The computer code must be compiled by compilers for each computer architecture, i.e., a program compiled on the HP cluster cannot be run on the Linux PC cluster. The computational grid in the present project consisted of two Single Program Multiple Data (SIMD) systems, which together constituted a Multiple Program Multiple Data (MPMD) system.
  - The correct types and versions of Grid applications and compilers have to be used on all computers; different versions of the same application are not likely to be compatible. In the present tests, the Grid application *GLOBUS 2.0 beta* and the compiler *MPICH-G2 v.1.2.3* were used on both the Linux cluster and HP cluster.

The method used to parallelise the code (the pseudo-spectral method) in x space, described in Section 2.3.2, gave rise to severe slowdown in all numerical test on clusters of computers. The group replaced the pseudo-spectral method by a sixth-order difference scheme and optimised the code by hiding the communication, where-after code was run with very good (often strongly super-linear) speedup on the Linux cluster, and with a reasonably good speedup when run with Globus between the two main nodes of the clusters. The method of parallelising the η space,

described in Section 2.3.3 gave speedup on the clusters but not in the tests with Globus; by hiding communication and minimising the number of communication events, this algorithm could possibly be made more efficient with respect to communication.

2.7 Numerical tests on a Shared memory computer

$N_{px} \times N_{p\eta}$	$N_x \times N_\eta =$ 100×100	$N_x \times N_\eta =$ 200×200	$N_x \times N_\eta =$ 400×400	$N_x \times N_\eta =$ 800×800
1×1	46 ; 1.0	174 ; 1.0	882 ; 1.0	5808 ; 1.0
2×1	31 ; 1.5	113 ; 1.5	448 ; 2.0	2709 ; 2.1
4×1	23 ; 2.0	72 ; 2.4	253 ; 3.5	1335 ; 4.4
5×1	22 ; 2.1	65 ; 2.7	218 ; 4.0	968 ; 6.0
8×1	–	54 ; 3.2	169 ; 5.2	613 ; 9.5
10×1	21 ; 2.2	51 ; 3.4	146 ; 6.0	515 ; 11.3
16×1	–	–	150 ; 5.9	472 ; 12.3
1×2	28 ; 1.6	105 ; 1.7	407 ; 2.2	2501 ; 2.3
1×4	15 ; 3.1	54 ; 3.2	203 ; 4.3	1037 ; 5.6
1×5	13 ; 3.5	46 ; 3.8	165 ; 5.3	801 ; 7.3
1×8	–	32 ; 5.4	107 ; 8.2	452 ; 12.8
1×10	9 ; 5.1	25 ; 7.0	88 ; 10.0	354 ; 16.4
1×16	–	–	63 ; 14.0	249 ; 23.3
4×4	10 ; 4.6	25 ; 7.0	83 ; 10.6	304 ; 19.1

Table 2: The total computing time T in seconds, and the relative speedup S (written as "T ; S"), as functions of different problem sizes $N_x \times N_\eta$ and number of processors $N_{px} \times N_{p\eta}$.

In order to test the performance of the code on a shared memory computer, the code was run on the computer system "Tee", a Sun Fire 6800 system with 16 Ultra Sparc III processors, 750 MHz each, with a 64 k L1 4-way cache, and with 16 GB memory (8ML2), at Department of Scientific Computing, Uppsala university. The code was compiled with the optimisation flag `-fast`.

In order to compare the efficiency of the parallelisation in the x and η directions, the problem size was varied the same amount in the x and η directions while keeping the number of time steps $N_t = 1000$ constant.

Table 2 shows the total computing time T in seconds, and the relative speedup S, as functions of the number of grid-points $N_x \times N_\eta$ and the processor grid $N_{px} \times N_{p\eta}$. The time is the "real" time measured using the Unix `time` command.

It is apparent that the parallelisation in η space is more efficient than the parallelisation in x space. Both methods give speedup in the numerical test, especially for the larger problems, but the parallelisation in η space give for some problems a large super-linear speedup, not seen for the parallelisation in x space.

3 Discussion

The conclusion from the numerical experiments both on shared memory machines and on clusters of computers is that the code should be parallelised in η space. The Padé scheme gives rise to a large number of tri-diagonal linear systems, which can be efficiently parallelised by the method of domain decomposition. The domain decomposition gives rise to Schur complement systems which are tri-diagonal, symmetric and strongly diagonally dominant, which makes it possible to solve these systems in parallel by a few Jacobi iterations. Only nearest-neighbour communications of a few vectors are needed between processors, and therefore the the parallel efficiency of the algorithm used for the semi-implicit Padé scheme is comparable to the parallel efficiency of algorithms for explicit difference schemes.

Some conclusions can be drawn regarding the parallelisation of the Vlasov code in higher dimensions. A closer look on the algorithms used to solve the two-dimensional Vlasov equation [4] gives that it is different from the one-dimensional algorithm [3] in one respect. In the one-dimensional code, the derivatives in x and η space are calculated independently from each other, where-after the boundary condition is applied. This makes it simple to replace the pseudo-spectral method in x space with difference schemes. In the two-dimensional code, however, the domain is Fourier transformed in $\mathbf{x}$ space before derivatives in $\boldsymbol{\eta}$ space are calculated, in order to maintain numerical stability. (This was not needed for the one-dimensional case [3].) Thus the algorithm is more closely based on the $\mathbf{x}$ space Fourier transform in the two-dimensional case than in the one-dimensional case, and therefore the algorithm should in the first place be parallelised in $\boldsymbol{\eta}$ space. More research is needed to investigate if high-order difference schemes can be used also in $\mathbf{x}$ space in more than one dimensions, without the need to use Fourier transforms.

The use of clusters of computers is an interesting possibility in which relatively cheap computers can be used to perform large scale computations. Local clusters show good speedup in the numerical tests performed on the one-dimensional Vlasov code. Tests between clusters of computers by the use of the Globus toolkit shows that it is possible to achieve numerical speedup after a careful optimisation of the parallel algorithms.

References

[1] Jonas Agmund, Johan Granat, and Magnus Ingelsson. *Parallel computing on the grid*. Internrapport nr. 3/2002, Dept. of Scientific Computing, Uppsala University, 2002.

[2] B. Eliasson. *Numerical Simulation of Kinetic Effects in Ionospheric plasma*. IT Licentiate thesis 2001-004, Uppsala University, Department of Information Technology, 2001.

[3] B. Eliasson. Outflow boundary conditions for the Fourier transformed one-dimensional Vlasov-Poisson system. *J. Sci. Comput.*, 16(1):1–28, 2001.

[4] B. Eliasson. Outflow boundary conditions for the Fourier transformed two-dimensional Vlasov equation. *J. Comput. Phys.*, 181:98–125, 2002.

[5] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*, chapter 4. The Benjamin/Cummings Publishing Company, Inc, 1994.