# ACCELERATING SOLAR WIND CALCULATIONS

**J. Schüle**

Gauß-IT-Zentrum, TU Braunschweig, Germany

*J.Schuele@TU-BS.de* (J. Schüle)

**Abstract**

A hybrid simulation model (PIC code) for the solar wind interaction with the ionosphere of stellar objects is optimized sequentially and its runtime almost halved. Parallelization with decomposition of the computational grid in static subdomains supported by a single ghost cell performs well because particles are almost uniformly distributed and thus cause no load balancing problems. We observe that we can spawn two independent threads on a dual-core Xeon processor without any performance loss due to the effective Hyper-Threading. We have compared partitionings in two different directions namely $x$ and $y$. A partitioning in $x$ direction shows a performance degradation due to increased L2 cache traffic and more communication traffic than a partitioning in the direction of $y$. If we combine both partitionings the application will show an increasing speedup with up to 32 processors. For production runs the performance is reasonable with an effecitvity of 78% with up to 16 processors. This effectivity could be improved on a cluster with faster network interfaces or a more effective open source MPI implementation. Currently we reach on a Gb Ethernet network for a unidirectional send only a bandwidth of 25 MB/s compared to 128 MB/s hardware bandwidth. In total, sequential optimization and parallelization both have reduced the waiting time for a simulation by a factor of 23 on 16 processors.

**Keywords: Solar Wind, PIC, MPI, Performance Optimization, Profiling**

## Presenting Author's Biography

J. Schüle, born 1960, studied Chemistry at universities in Tübingen and Münster (Germany). After his PhD he worked three years as Post-Doc in Stockholm (Sweden) and Berlin. Since 1990 he has been employed at the computing centre at the Technical University Braunschweig. In his spare time he got a diploma degree in Mathematics. Since 1993 he has given lectures in various topics of Parallel and High Performance Computing.

ISBN 978-3-901608-32-2  Copyright © 2007 EUROSIM / SLOSIM

# 1 Introduction

## 1.1 Physics

A hybrid simulation model for the solar wind interaction with the ionosphere of planets like Mars and Titan has been developed at the Institute for Theoretical Physics at TU Braunschweig [1]. The present version of this program code has already been successfully applied to the solar wind interaction with comets [2, 3] as well as to the plasma environment of Mars [4] and Titan [5].

In the hybrid approximation the electrons are modeled as a massless charge-neutralizing fluid, whereas the ions are treated as macro-particles. The model dynamically solves the following equations on a curvilinear grid in three spatial dimensions:

$$\frac{dv_s}{d} = \frac{q_s}{m_s}(E + v_s \times B) - k_D n_n v_s \,, \quad (1)$$

$$E = -u_i \times B + \frac{(\nabla \times B) \times B}{\mu_0 e n_e} - \frac{\nabla P_{e,SW} + \nabla P_{e,hi}}{e n_e} \,, \quad (2)$$

$$\frac{\partial B}{\partial t} = \nabla(u_i \times B) - \nabla \times \Big[\frac{(\nabla \times B) \times B}{\mu_0 e n_e}\Big] \,. \quad (3)$$

Here $q_s$, $m_s$ and $v_s$ denote the charge, mass and velocity of a macro-particle of species $s$, respectively. $k_D$ is a constant, $n_n$ is the number density of the neutrals, $u_i$ is the mean ion velocity, $n_e$ the electron density, $P_{e,SW}$, and $P_{e,hi}$ correspond to two different electron pressure terms, and $\nabla$ and $\times$ represent the Nabla operator and the cross product, respectively. The code operates on a curvilinear grid in three spatial dimensions which reflect the spherical resolution in the vicinity of the planetary atmosphere. Details of the calculations may be found in [1] and [4].

## 1.2 Simulation Code

The basic scheme to solve the specified set of equations is the 'Particle-in-Cell' (PIC) method: a fixed grid is defined in coordinate space and the electromagnetic field quantities as well as charge densities and currents are defined only on the nodes of this grid whereas the macro-particles are located anywhere in the computational domain. A PIC code consists of four basic steps, namely:

(i) Gathering moments; densities and currents are computed (gathered) on each grid point from given particle positions and velocities.

(ii) Solution of field equations; using densities and currents the electric and magnetic fields are updated in a leap-frogged algorithm.

(iii) Force interpolation; the electromagnetic field quantities are interpolated from the grid nodes to each particle position.

(iv) Particle movement; given the forces acting on each particle, both their position and velocity are updated, accordingly.

In addition to this basic PIC steps a simple smoothing procedure is applied to the magnetic and the electric field to reduce noise from statistical particles in each time step. More details are given in [2].

## 1.3 Implementation

The code is implemented in C++ where all fields are defined in a common class `vfield` containing a three-dimensional data array and several operators with three to five arguments among other things. These operators return the pointer to a memory location in this data array depending in all cases on the coordinates, in some cases on the number of vector components of the field and sometimes even on a conditional parameter used to interchange coordinate directions to code cross products smarter. A second class contains and handles particles which are organized in a chained list to add/remove particles easily as required.

# 2 Performance Optimization

The results of a simple profiling (gprof) of the original sequential program are given in column 2 of table 1. Listed are the 10 most time consuming functions amounting to 84.2% of the total time of 3m 40s required for 3 time steps (of some thousands required in a production calculation) on an Intel Xeon Dual-Core processor running at 3.2 GHz. This processor is known to have deficiencies in its memory access times. This is supported by a STREAM benchmark [6] resulting in 3.67 Gflops compared to 6.4 Gflops peak performance indicating a performance loss of 1.73 Gflops or 27% due to its weak memory system.

| function | %[1] | %[2] | %[3] |
|---|---|---|---|
| `Solve_B` | 25.0 | 11.0 | 13.1 |
| `vfield::ijkl` | 15.3 | 46.3 | 2.9 |
| `calc_m_rJLG` | 9.4 | 5.4 | 2.8 |
| `field_interpol` | 7.6 | 2.0 | 4.7 |
| `get_v` | 6.4 | 2.8 | 1.9 |
| `main` | 5.1 | 0.0 | 0.1 |
| `smooth` | 5.0 | 4.3 | 0.8 |
| `cart_coord` | 3.9 | 4.8 | 3.4 |
| `get_wijk` | 3.5 | 7.9 | 3.5 |
| `calc_m_rJ` | 3.1 | 1.6 | 3.0 |

Tab. 1 Functions and their profiling amounts are listed in percentage of total time used: [1] Original C++ code on final target computer. [2] Original C++ code on ES45. [3] Code after improvements. The values are divided by 1.9 to account for the reduced overall runtime compared to column 2.

To illustrate differences in profiling data related

ISBN 978-3-901608-32-2

Copyright © 2007 EUROSIM / SLOSIM

to different processors we have listed the results for the same profiling experiment on an 1000 MHz Alpha-processor in a COMPAQ ES45 in column 3 of table 1. A single processor of this machine yields a STREAM benchmark result of 1.9 Gflops compared to 2.0 Gflops peak performance. Clearly this computer is better balanced than the Xeon processor which becomes even more pronounced when more than one processor is utilized.

While on the Xeon processor most time is spent in function `Solve_B`. In this function various cross products in all three dimensions of the magnetic field are formed. It consumes only 11% on the Alpha-processor due to its faster memory system. In contrast to this, we have recognize a change in the amount required for the operator `vfield::ijkl` in the C++ field class `vfield`, which is a pure integer manipulating function. Here the Xeon system with fast integer arithmetic units clearly outperforms the Alpha-processor.

Nonetheless we have noticed in this first profiling output a significant drawback of the smart C++ field class operator `vfield::ijkl` - it has slowed down the program. By replacing this with inline address arithmetic directly working on the data array the computational time has indeed been reduced by 15%.

In the following we mainly concentrate on the profiling results for the Xeon processor as the final target computer system consists of a cluster of these systems.

The distribution of most time consuming functions among the PIC code steps is of great importance. `Solve_B` is the major function in step (ii), `smooth` and to a great extend `vfield::ijkl` belong to it as well. The function `field_interpol` is part of step (iii). These functions manipulate field quantities on the grid and are independent of the macro-particles. Depending on the macro-particles are the functions `calc_m_rJLG` and `calc_m_rJ` in step (i) and `get_v` (Maxwellian velocity for new particles), `cart_coord` (Cartesian coordinates for a particle) and `get_wijk` (weighted coordinates for a particle). At first it is surprising that functions related to step (iv), the particle movement, are missing.

It would be boring and overburdening this paper to give full details of all performance optimizing steps. But one simple improvement is the replacement of a repeated call of the power function in `smooth` by a precalculated array in the following piece of code.

```
// Replacement of
//  pow(2.,(double)(-(i*i+j*j+k*k+3)))
//  in i,j,k loops below running from
//  -1 to 1 with increment 1  for each
//  grid node.
   potenz[0]=1.;
   for(i=1;i<7;i++) potenz[i]=0.5*potenz[i-1];
```

In addition to this modification the changes mainly consist of removing operator calls in class `vfield` and of rearranging the particle class into boxes with tunable size. Now, instead of allocating/freeing each particle, we have arranged the particle data in these boxes and have added some cleaning functions to get rid of almost empty boxes.

As may be seen in column 4 in table 1 these modifications have indeed changed the profiling amounts drastically, and even more important they have reduced the overall runtime for a sequential run from 3m 40s down to 1m 54s or almost have halved the runtime for production runs. This is an important improvement because production runs require between 3000 to 10000 more time steps than our very limited test case.

## 3 Parallelization

Parallelization of a PIC code may introduce severe load balancing problems if particles are not equally distributed among all grid nodes. This is especially important if functions in steps (i) and (iv) - here particles are involved - are very time consuming [7]. Fortunately this is not the case in our application because solar wind particles are almost uniformly distributed on the computational domain with the exception of the stellar object considered. And furthermore, as pointed out in the previous section, step (iv) does not require significant amounts of time. Therefore it is a promising approach to begin with a static partitioning of the grid and to distribute particles accordingly.

Subdividing a cubical computational domain is possible in all three directions but should take the following into account

- load balancing, that is it should generate domains that are almost equal in size;
- preserve regular data layout and generate regular (straight) interfaces between domains;
- preserve inner loops, that is it should preserve data access with stride 1;
- reduce the number of communication toward an increase in communicated data between processes with the help of a ghost or halo region around domains;
- consider a preferred movement direction of particles in order to reduce the number of particles passing from one domain to another.

The above specified viewpoints clearly favour regular cuts in the physical $x$ and $y$ directions of the cube becausebecause $z$ direction is organized in inmost loops with data access pattern with unit stride. The halo region consists of one ghost cell layer, because only direct neighbouring cell information is required in the calculation (cf. Fig. 1).

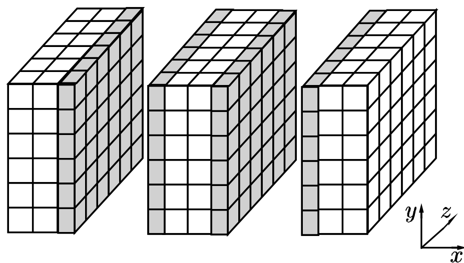


Fig. 1 Schematic decomposition of the computational domain into three subdomains along $x$ direction. Indicated is the halo region consisting of one ghost cell layer at each interface.

Since we do not expect significant problems in the load balancing with a static decomposition as many subdomains are formed as processors are used. In future developments we will extend this to more subdomains than processors to account for unbalance in the work load which arises due to missing particles in grid cells representing the stellar object. This will become more important in the future. We are planning to move from the curvilinear grid to an adaptive Cartesian grid. Representation of the stellar object in this grid as well as the adaptivity of the grid will introduce more severe load balancing problems and thus more flexibility in the distribution and replacement of domains among processes is mandatory.

Random numbers are supplied by the sprng2.0 library [8] in independent streams per processor. Substituting the sequential random number generator against the sprng2.0 library causes only a small overhead. With these decisions the parallelization is straightforward.

Particles are generated independently in each subdomain and relocated between subdomains according to their movement. All the electromagnetic field quantities as well as charge densities and currents are distributed accordingly and the halo region is updated as required. Only the node coordinates are replicated in each process to allow for random replacement of particles in all the processes in case some of them are leaving the computational domain. In addition to shortened runtimes this approach opens up the possibility to extend the program to larger and/or finer grids because the memory requirements per processor is almost linearly reduced by the number of processors used.

### 3.1 Hard- and Software Considerations

Parallelization strategies and concepts are easy to consider and investigate theoretically, but a little bit more complicated when they have to be implemented. Already the sequential profiling information discussed in 2 shows a significant system variation because it mainly depends on the balance between processor and memory speed. This is even more pronounced in parallelization because besides processor and memory speed even network latency and bandwidth, MPI implementations (cf. [9]), remote memory access policies, and filesystem access influence the runtime.

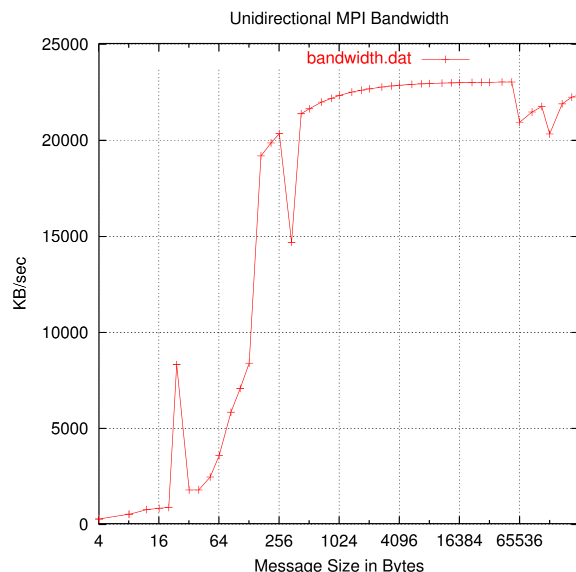


Fig. 2 Unidirectional MPI bandwidth between two nodes connected via 1 Gb interfaces measured with `LLCBENCH` suite [12] for the target cluster.

As implementation has been started on a COMPAQ ES45, has been continued on an IBM p-Series, and has been finished on a Linux cluster portability is an important issue. Therefore MPI [10] in version 1.1 is chosen as software platform. Especially at the current state neither IO-extensions nor extensions to handle one-sided communication nor dynamic process management are incorporated nor do we consider hybrid programming mixing MPI and OpenMP. This is a major drawback because one-sided communication can help to speed up particle calculations in PIC codes as pointed out in [7]. But the final target system has no hardware support for this type of communication (see below).

On the COMPAQ ES45 and the IBM p-Series we use vendor compilers and libraries, on the Linux cluster we use the Intel C compiler suite version 9.1 together with OpenMPI [11] in version 1.2.2. Only performance numbers for the Linux cluster are given below therefore we restrict the hardware specification to this system.

The system consists of Intel Xeon Dual-Core running at 3.2 GHz, 2 MB L2 cache, and 2 GB memory, operating under SUSE LINUX 10.0. The cluster is interconnected with two 1 Gb Ethernet connections one of them is dedicated to the communication traffic and the other one mainly handles IO to NFS mounted filesystems. The Ethernet cards do not provide the possibility for remote memory access. Therefore one-sided communication is not feasible.

 Copyright © 2007 EUROSIM / SLOSIM

Even worse than this is the effective bandwidth this network provides in our current OpenMPI 1.2.2 implementation [11]. As depicted in fig. 2 we measure for an unidirectional `MPI_Send` a bandwidth less than 25 MB/s compared to 128 MB/s nominal bandwidth of a Gigabit Ethernet interface. Considering McClements investigations [9] this should even be worse for other open source MPI implementations and urgently requires further investigations.

### 3.2 Methods and Results

The implementation of a grid decomposition with halo regions is straightforward. A bit more involved is the coding of the particle movement because an unknown number of particles has to be moved to beforehand unknown processes and vice versa. In our first implementation we have used immediate send/receive operations of one integer in order to inform the receiving process and then subsequently to exchange the corresponding amount of particle data. The end of a particle movement period is passed on to all the processes by sending the integer $-1$. While this version was running without problems on a shared memory node this approach got stuck once in a while on the Xeon cluster. We have traced this behaviour down to problems with overtaking messages in our OpenMPI implementation under certain conditions even though MPI clearly prohibits this in its documentation.

For that reason we have changed to `MPI_Isend` in combination with `MPI_Iprobe`. This has the advantage of avoiding one point to point communication per particle package because the information about the number of particles to be received is available in the `status` information returned by `MPI_Iprobe`. Of course we neither exchange particles one by one nor do we use derived datatypes to send them but perform local buffering for performance reasons (see [14]).

Both implementations differ only slightly in their runtimes but the latter has shown not any deadlock up to now.

The results given below refer to calculations on a $90 \times 90 \times 90$ grid in curvilinear coordinates modeling the solar wind around Titan over 40 time steps on the above described Xeon cluster.

Runtimes for different scenarios are depicted in fig. 3. One scenario refers to a test of the nodes in process handling and we compare the same runs executed with one and two threads on each node (compare the single loaded marked curve with others). The other scenarios compare different partitioning directions which are either a decomposition along $x$, along $y$ or a combination of both directions.

As we have already mentioned above and proved with the STREAM benchmark, the Intel Xeon architecture has deficiencies in memory accesses with a ratio of 0.54 between memory and peak performance. This situation is even worse when using

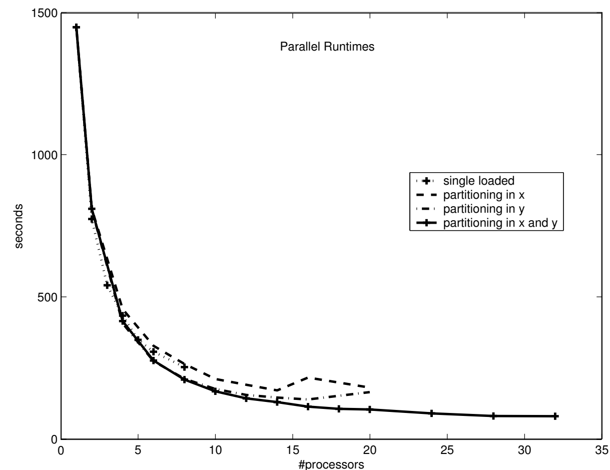


Fig. 3 Parallel runtimes for solar wind interactions with Titan as described in the text for different numbers of processors. Shown are curves for runs with one process per core (single loaded) and with two processes per node for different partitioning strategies.

both cores of the Dual-Core Xeon, as this ratio drops below 0.3. As pointed out in section 2 our overall code is memory bound and thus we expect a significant performance degradation when running with two processes on a dual-core compared to running the same parallel application on two different nodes. Surprisingly enough, this is not the case as can be seen in fig. 3. A first assumption for this unexpected behaviour points toward the Gigabit Ethernet interconnections with their poor bandwidth (s. fig 2) which certainly introduce an overhead that may compensate for the pure memory accesses in dual loaded nodes. In fact profiling with mpiP [13] reveals that this is not the case, as both singe loaded and dual loaded runs both show 11% MPI overhead, thereof most time is spent in `MPI_Wait` (59%) and `MPI_Iprobe` (16%) which are called in functions which exchange particles. Another aspect is a very effective Hyper-Threading for the Xeons. In order to investigate this further we have started four parallel threads on one dual-core node and have compared this to a run with two parallel threads on the same node. Four threads should not run faster on two cores but slower due to the increased competition among the threads. But we have observed that four threads on two cores run as fast as two threads indicating that scheduling and thread and data management introduce an almost neglectable overhead on the Xeon dual-core nodes.

As pointed out in section 3 a simple decomposition of the computational domain provides a meaningful strategy for parallelization. Runtimes for partitionings in $x$ and $y$ are compared in fig. 3 together with timings for a combined partitioning in $x$ as well as in $y$. The $x$ direction is organized in outmost loops and $y$ in middle loops of a $90 \times 90 \times 90$

ISBN 978-3-901608-32-2

Copyright © 2007 EUROSIM / SLOSIM

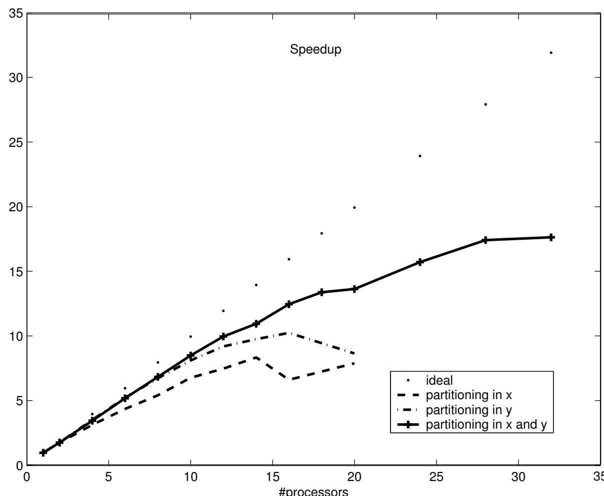


Fig. 4 Speedup for solar wind interactions with Titan as described in the text for different numbers of processors.

grid. We expected partitioning in $x$ to be faster with $8100 = 90 \times 90$ grid points fitting well into cache in most computations. To our surprise this is not the case and a partitioning in $y$ has proven to be faster instead.

To investigate this behaviour we have compared these partitioning schemes with the help of the hardware performance monitor on an IBM p-Series. In fact this comparison proves the importance of the $L2$ cache for runtimes. In contrast to our estimation the $x$ partitioning requires twice the $L2$ traffic and a significant larger amount of page faults with IO traffic than the $y$ partitioning thereby reducing the hardware Flops rate by 12% and increasing the runtime by the same rate. But the difference between these partitioning schemes is smaller than 12% on the Xeon cluster. If we assume a similar behaviour of the memory systems for the Xeon cluster and the IBM p-Series, the following question arises: Where does the $x$ partitioning gain time on the Xeon cluster? Again a profiling with mpiP [13] provides its usefulness. The MPI overhead in the $x$ partitioning is 7% and thus smaller than in the $y$ partitioning with 11%. The profiling data indicates that most time consuming MPI functions are called during particle movement in step (iv). We have traced the MPI calls in this program unit. Many particles have velocities in $x$ direction therefore in the $x$ partitioning scheme most data traffic is directed from threads with domains with low indices to neighbouring threads with domains with higher indices. Therefore there is less data traffic in a partitioning in $y$ direction with respect as well the frequency as the amount. Nonetheless and very surprising the MPI overhead is less in case of a partitioning in $x$ than in $y$ direction. This has to be investigated further.

As can be seen in fig. 3 and even better in fig. 4 - the speedup is displayed there - partitioning in $y$ will become more and more superior to a partitioning in $x$ the more processors we use. Nonetheless both partitionings prove their weaknesses in case we use more than 12 processors. This is as could be expected because chunk sizes get too small. Much better but still unsatisfying are the speedups measured for combined partitionings in $x$ and $y$ directions. But more important than the speedup are of course savings in runtime and as a consequence reduced waiting times for researchers working with this program code. In the beginning a calculation of the solar wind around Titan required more than 46 minutes for 40 time steps. The same calculation can now be accomplished in less than 2 minutes on 8 dual-core Xeon nodes.

## 4 Conclusion

We present the performance optimization and parallelization of a hybrid simulation model (PIC code) that has been developed at the Institute for Theoretical Physics. This model is applied to calculate the solar wind interaction with stellar objects.

By removing C++ operators, introducing block allocation for particles, and substituting replicated calculations against precalculated arrays the sequential runtime has been halved. In addition to this, sequential profiling information reveals valuable information for a parallelization strategy. We have achieved a reasonable speedup with a decomposition in static subdomains supported by a single ghost cell. First we noticed that Hyper-Threading on dual-core Xeon processors effectively handles multiple threads. Therefore it is possible to spawn two independent threads on a dual-core Xeon without any performance loss.

We have compared partitionings in two different directions namely $x$ and $y$ and have analyzed their differences. A partitioning in $x$ direction shows performance degradation by 12% due to increased L2 cache traffic and more communication traffic during particle movement with respect to as well the frequency as the amount of data. In contrast to these observations the MPI overhead is smaller in the $x$ partitioning scheme and performance degradation is much less pronounced than expected from pure numbers of the hardware performance monitor.

As expected we will achieve better results if we combine partitionings in $x$ and $y$ direction than with a single direction because the combination shows a better ratio between inner cells and ghost cells. We achieve an increase in speedup up to 32 processors but more than 16 processors should not be used in production runs because the effectivity significantly drops with more processors. We expect better speedup numbers for runs on a hardware with faster network interfaces which allow for one-sided communication and perhaps even with a

more effective open source MPI implementation.

In conclusion, our work provides researchers at the Institute for Theoretical Physics now with results after 1 day compared to 3 weeks before this work started. Half of these savings are due to sequential optimization of the program code and the remaining is due to parallelization. We will continue our work in both directions and extend it to a third direction as are mathematical improvements.

## 5 References


[1] T. Bagdonat, and U. Motschmann, J. Comput. Phys. 183 (2002) p. 470.

[2] T. Bagdonat: Hybrid Simulation of Weak Comets, Dissertation, TU Braunschweig (2005).

[3] T. Bagdonat, U. Motschmann, K.-H. Glassmeier and E. Kührt, ASSL Vol. 311: The New Rosetta Targets. Observations, Simulations and Instrument Performances, edited by Colangeli, L., Mazotta Epifani, E., and Palumbo, P., (2004) p. 153.

[4] A. Bößwetter, T. Bagdonat, T. Motschmann, and K. Sauer, Ann. Geophys. 22 (2004) p. 4363.

[5] Simon, S., Bagdonat, T., Motschmann, U., and K.-H. Glassmeier, Ann. Geophys. 24 (2006) p. 407.

[6] McCalpin, J. D.: Memory Bandwidth and Machine Balance in Current High Performance Computers, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter (1995).

[7] C. Othmer, and J. Schüle, Comp. Phys. Comm. 147 (2002) p. 741.

[8] M. Mascagni, and A. Srinivasan (2000), ACM Transactions on Mathematical Software, 26 (2000) p. 436.

[9] E. McClements: Performance Comparison of Open Source MPI Implementations: http://www2.epcc.ed.ac.uk/dissertations/dissertations-0506/2688821-9h-dissertation1.1.pdf.

[10] M. Snir, S. Otto, S. Huss-Ledermann, D. Walker, and J. Dongarra: MPI: The Complete Reference (Vol. 1) - The MPI Core, The MIT Press, Cambridge (1998).

[11] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine in Proc. Fifth Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, Barcelona (2006).

[12] http://icl.cs.utk.edu/projects/llcbench.

[13] J. Vetter, and C. Chambreau: mpiP: Lightweight, Scalable MPI Profiling, http://sourceforge.net/projects/mpip.

[14] J. Träff, R. Hempel, R. Ritzdorf, and F. Zimmermann in J. Dongarra, E. Luque, and T. Margalef: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, New York (1999).


*ISBN 978-3-901608-32-2*

*Copyright © 2007 EUROSIM / SLOSIM*