FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

A Comparison of two Parallization Strategies for TRACE

*Michael Gerndt, Olaf Neuendorf**
*Joachim Prümmer, Harry Vereecken**

KFA-ZAM-IB-9425

November 1994
(Stand 22.11.94)

(*) Institut für Erdöl und Organische Geochemie (ICG 4)

A Comparison of two Parallelization Strategies for TRACE

Michael Gerndt[1], Olaf Neuendorf[2],
Joachim Prümmer[1], Harry Vereecken[2]

[1] Zentralinstitut für Angewandte Mathematik (ZAM)
[2] Institut für Erdöl und Organische Geochemie (ICG 4)
Forschungszentrum Jülich
D-52425 Jülich
{m.gerndt, o.neuendorf, j.pruemmer, h.vereecken}@kfa-juelich.de

Abstract

In this report we compare two different methods of parallelization of a finite element code describing water flow in soils. The first method uses Domain Decomposition based on a parallel Schwarz algorithm. The second method uses a Data Partitioning approach pursued in High Performance Fortran (HPF). Experiments with the parallel versions were performed on the Paragon XP/S 10 at KFA.

1 Introduction

In the consequence of the agricultural and industrial growth of the last decades many environmental problems appeared. Worldwide the protection of water resources is one of the major issues in the present day environmental policies. This involves a detailed understanding of the transport of chemicals in soils and aquifers. Therefore mathematical models are increasingly used as the most important tool to quantify the transport of these pollutants and to understand the underlying processes.

The common mathematical approach is to describe these processes as a set of deterministic ordinary and partial differential equations which are solved numerically. In the last years it has been recognised that a deterministic approach is not able to describe the behaviour of various pollutants in natural systems. Detailed measurements of soil and aquifer properties have shown a considerable spatial variability which influences substantially the behaviour of pollutants. Taking account of the intrinsic variability leads to stochastic partial differential equations. For the numerical solution of this type of equations one often needs a discretization with a large number of nodal grid points (more than 10^6 unknowns), depending on the extend of the area of interest and the correlation scale of the parameters. Additionally, in variably saturated systems (soils with shallow groundwater table) the partial differential equations describing the transport of reactive (sorptive) solutes are nonlinear and therefore more difficult to solve in terms of numerics and computer requirements. The combination of both problems requires powerful computer systems to obtain a solution.

Recent developments in computer systems with parallel architecture have stimulated the interest in using this type of computers to solve such problems. Various methods and techniques are

available in the literature which allow to treat these types of numerical problems on massively parallel computer systems (MPP). In this report we compare two different methods of parallelization of a finite element code describing water flow in soils. The first method uses Domain Decomposition based on a parallel Schwarz algorithm. The second method uses a Data Partitioning approach pursued in High Performance Fortran (HPF) [7]. Although first HPF compilers for massively parallel systems are available we manually implemented the parallel version since unstructured grid applications are not supported in the current design of HPF.

2 Application

The usual approach to model the transport of pollution in the soil/aquifer system is to start with the well known Richards equation describing the flow of water [6]

$$\begin{array}{l} F\frac{\partial h}{\partial t} = \vec{\nabla} * (\mathbf{K}\vec{\nabla}(h+z)) + S \\ F = \frac{d\Theta}{dh} \end{array} \quad (1)$$

where F is the storage term (L^{-1}), h is the pressure head (L), $\mathbf{K}$ the hydraulic conductivity tensor (LT^{-1}), z the vertical coordinate (L), S the sink/source term (T^{-1}), Θ the moisture content (L^3/L^3), dimensions in brackets. The parameter $\mathbf{K}$ and the functional relationship between Θ and h are known to vary in space thereby influencing considerably water flow and solute transport. They are often considered as stationary random space functions, described by their first and second statistical moments. To solve flow problems using Eq. (1), initial and boundary conditions need to be specified. Depending on the type of problem three different types of boundary conditions can be imposed: Dirichlet, Neumann, and variable, which means either Dirichlet or Neumann, depending on the value of h.

To perform the time discretization the method of finite differences is used, while for the space discretization the Bubnov-Galerkin finite element method with linear hexaedric elements is applied. This leads to the non linear matrix equation

$$\mathbf{A}(\vec{h}) * \vec{h} = \vec{y} \quad (2)$$

where $\mathbf{A}$ is the coefficient matrix depending upon the solution and $\vec{y}$ a vector containing known information resulting from the boundary conditions. A fully implicit approach updating the coefficient matrix and a load vector, combined with Picard's iteration method to solve for the non-linear iteration is used. This results in an outermost time loop and an inner non-linear loop, as shown in Figure 1. A third iteration loop is necessary to account for the variable boundary condition. The resulting system is solved by the Conjugate Gradients method using diagonal scaling as preconditioner [4].

After solving for the water transport, the solute transport can be calculated using the convection/dispersion equation:

$$\begin{array}{l} \vec{\nabla}(\mathbf{D}\vec{\nabla}c - c\vec{u}) = \partial c/\partial t + S(c,t) \\ c \quad - \quad concentration\ in\ water \\ \vec{u} \quad - \quad pore\ velocity \\ \mathbf{D} \quad - \quad dispersion/diffusion\ tensor \\ S \quad - \quad sink/source\ term \end{array} \quad (3)$$

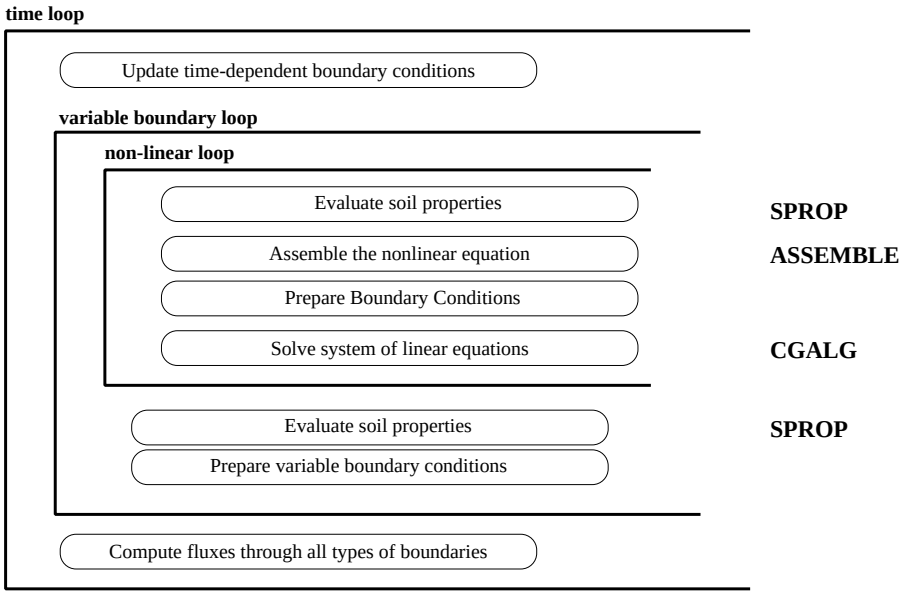

Figure 1: Sequential program structure

This second step is not discussed in this paper. The interest is focused upon the possibility to calculate the water flow on a parallel computer using two different methods of parallelization. The methods presented for water flow are however applicable to the convection/dispersion equation.

3 Parallelization Strategies

For the application described in Section 2 we implemented two different parallel versions based on Domain Decomposition and Data Partitioning. Domain Decomposition and Data Partitioning have in common that the nodes of the finite element grid are distributed onto the processors of a parallel machine. Thus those processors are responsible for the nodes they own. The main difference between the two strategies is that Domain Decomposition has a coarser level of communication than Data Partitioning.

In Domain Decomposition a processor computes a solution for the linear equation regarding only the nodes it owns. Afterwards the solutions on the borders are compared with those of the neighboring processors. This is iterated until some criterion is satisfied.

In Data Partitioning the global solution for the linear equation system is computed in parallel. Thus in each iteration the solution vector is consistent to the one computed by the sequential version of the program.

3.1 Domain Decomposition Approach

The Domain decomposition technique is based upon the idea, that a physical domain of interest Ω can be divided into a number of subdomains Ω_i, while for each of them the same differential equations have to be solved. It was the idea of Schwarz (1890) to use overlapping subdomains

therby exchanging appropriate subdomain boundary information in order to obtain a solution for the overall domain.

We define on the domain Ω a boundary value problem [2, 5] described by Eq. (4)

$$Lu = f(\vec{x}, t), \vec{x} \in \Omega \quad (4)$$

where u is the pressure head and L the partial differential operator

$$L = \frac{\partial}{\partial t} - \vec{\nabla}\mathbf{K}\vec{\nabla}.$$

The domain Ω is divided into p subdomains

$$\Omega = \bigcup_{i=1}^{p} \Omega_i \quad (5)$$

with

$$L_i u_i^k = f_i(\vec{x}, t) \quad (6)$$

and L_i and f_i are restrictions of L and f on Ω_i, respectively, and k is an iteration index. The boundary conditions for the subdomains are given by

$$u_i^k(\vec{x}, t) = \gamma_i^k(\vec{x}, t), \ \ \vec{x} \in \partial\Omega_i/\partial\Omega \quad (7)$$

where γ_i^k are considered as pseudo-Dirichlet boundary conditions, which are defined below by Eq. (11).

A natural way to solve Eq.(4) is to apply either finite difference or finite element techniques, requiring the definition of a numerical grid. For purpose of simplicity we will introduce a simple rectangular grid G on the domain $\Omega = [0, 1] \times [0, 1] \times [0, 1]$ where

$$G = \{(x_i, y_i, z_h), \ x_i = i/n, \ y_i = j/n, \ z_h = h/n, \ 0 \leq i, j, h \leq n\} \quad (8)$$

with n a positive integer representing the number of intervals (or elements). Following Rodrigue we define a sequence of integers such that

$$0 = l_1 < l_2 < r_1 ... < l_p < r_{p-1} < r_p = n \quad (9)$$

where p is the number of subdomains. These integers delimit the right and left boundaries of the subdomains

$$\Omega_i = [x_{l_i}, x_{r_i}] \times [0, 1] \times [0, 1] \ \ 1 \leq i \leq p. \quad (10)$$

The parallel Schwarz procedure solves Eq.(6) on each of the subdomains independently, while the boundary conditions defined by Eq. (7) are updated in the following way

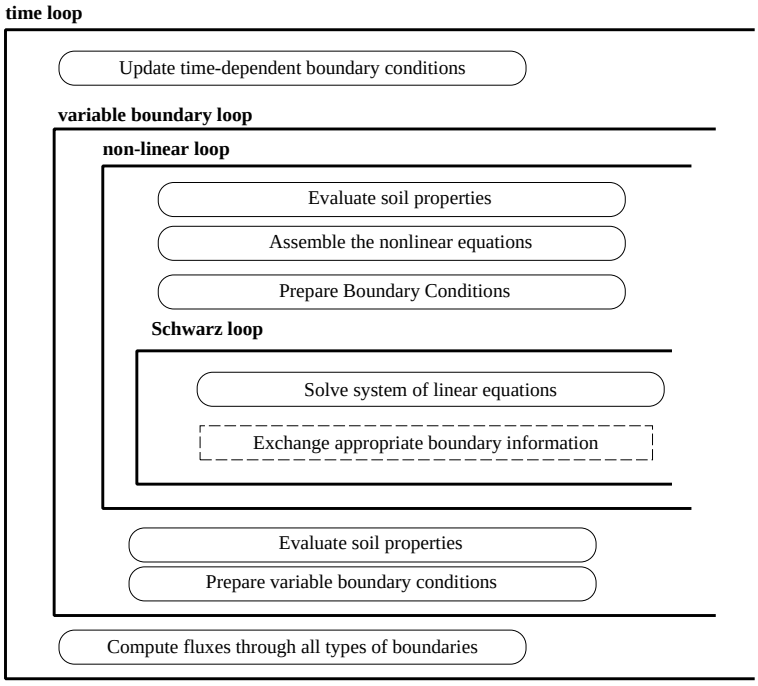

Figure 2: Parallel program structure with Domain Decomposition

$$\begin{aligned} u_i^k(x_{l_i}, y, z, t) &= u_{i-1}^{k-1}(x_{l_i}, y, z, t), \quad 0 \le y, z \le 1; \quad i = 2, ..., p \\ u_i^k(x_{r_i}, y, z, t) &= u_{i+1}^{k-1}(x_{r_i}, y, z, t), \quad 0 \le y, z \le 1; \quad i = 1, ..., p-1 \end{aligned} \tag{11}$$

Eq. (11) defines the pseudo-Dirichlet boundary conditions. Although the overlap of the subdomains may be varied, only the information at the boundary itself has to be exchanged.

Figure 2 shows a flow chart of the relevant part of the TRACE code, where the Schwarz algorithm was implemented. The major additional feature according to the sequential version is the introduction of a communication loop (Schwarz loop). Therefore just a slight change of the code is necessary, which is one of the great advantages of this method. The innermost part of the program is the solution of the linear equation system by the conjugant gradients method. The number of CG iterations was limited to a maximum of 5 to avoid an unproportional high effort for the first steps of the Schwarz loop. This Schwarz loop has to update its boundary vector in such a way that neither the effort of this update nor the number of the Schwarz iterations itself is too high. The effort for an update is mainly governed by the number of CG iterations while again a higher number of Schwarz iterations leads to a rise in the number of CG iterations.

The Schwarz loop is repeated until the difference between the old and the new boundary vector falls under a prespecified criterium. All the rest of the program remains unchanged except for the input/output (I/O). The I/O has to be modified anyway, no matter which strategy of parallelization is applied.

3.2 Data Partitioning Approach

In the Data Partitioning approach the sequential algorithm is parallelized by executing its operations simultaneously on the processors if these operations are independent. The resulting parallel program performs almost the same number of floating point operations and the computed result is equal to the result of the sequential program.

The operations are distributed to the processors with respect to data locality. Similar to the Domain Decomposition approach, the global grid (Eq. 8) is subdivided into subgrids assigned to the processors. Whereas the current implementation of the Domain Decomposisiton approach only supports one-dimensional splitting, the Data Partitioning implementation already allows splitting in each dimension.

The data structures in the sequential program reflect the global grid. All arrays are distributed to the processors according to the decomposition of the grid and the operations are assigned to the processors where the data reside.

This approach is the basic design concept of HPF. The current version of HPF supports regular distribution of arrays to processors. This is not sufficient for this application since the arrays are one-dimensional and the three dimensional grid is mapped in an arbitrary way to these arrays. This mapping makes it necessary to specify irregular distributions for the arrays, i.e. to map array elements individually to the processors. Upcoming HPF compilers are not able to do automatically what we did by hand for this program because irregular distribution are not supported and runtime overhead is reduced taking application properties into account.

In almost all phases of the sequential algorithm shown in Figure 1 the computation for individual finite-element nodes requires information of neighbouring nodes. Since neighbouring nodes may reside on other processors these data have to be communicated prior to the operation or the operation has to be performed where the data reside and the result has to be communicated. The implementation applies both alternatives to reduce communication overhead.

Some operations in the sequential code require global communication among all nodes, e.g. the computation of the residuum of the global linear equation system's solution. All processors have to know the global value to make the same decision whether another CG-iteration has to be executed.

Figure 3 gives an overview of the resulting parallelized program. Neighbour communication as well as global communication is spread over the entire code. The most critical communication with respect to parallel program efficiency is the gather and the global sum in each CG-iteration. A CG-iteration basically consists of a single matrix-vector multiply for the subgrid. Therefore the communication overhead - which is dominated by the message passing latency and thus only dependend on the partitioning strategy and nearly constant in the number of processors - more and more dominates the computation when the size of the subgrid is reduced.

Before the parallel program can be executed the following tasks have to be performed:

1. computation of subregions done by the *distributor*
2. computation of object distributions done by the *object partitioner*
3. rearranging input data done by the *data partitioner*

Figure 4 shows the global organization of these steps and the related tools. The user has to only

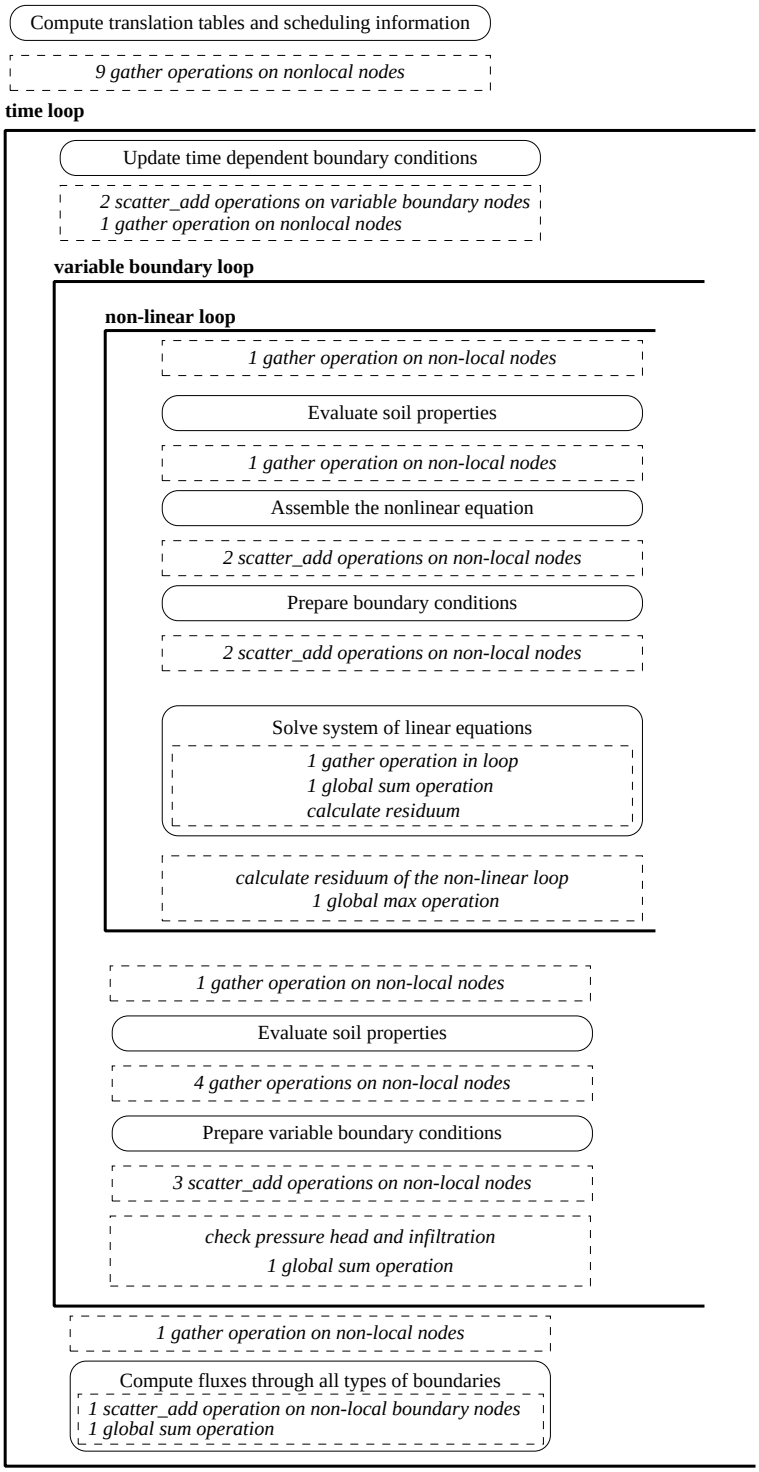

Figure 3: Parallel program structure with Data Partitioning

specify the desired distribution strategy, i.e. how the global domain should be distributed on the processors. The other steps are performed automatically.

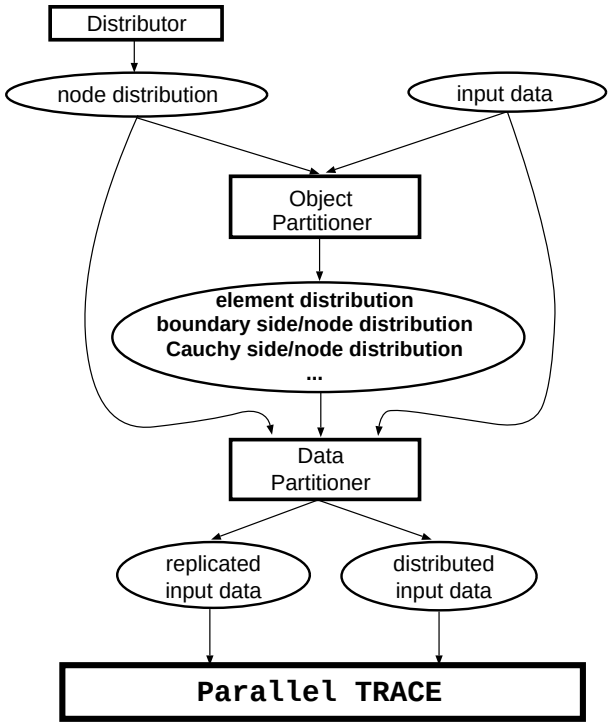

Figure 4: Overall organization of the Data Partitioning version

The most performance critical decision is the selection of the distributions. Although the parallel code handles arbitrary distributions correctly, the distribution of nodes should be done in a way to minimize the communication overhead.

For this application we developed a tool, the *distributor*, which divides the three–dimensional problem region into regular subregions. Input is the extension of every axis (x–, y– and z–axis) and the number of blocks per axis. According to this user–specification the distributor computes the number of nodes and the global node numbers assigned to each processor, i. e. the *node distribution*.

From the node distribution several distributions are derived automatically by a tool called *object partitioner*. The partitioner computes the distribution of the elements based on the majority rule. An element is assigned to that processor which owns most of its nodes. If there is no unique processor with this property an arbitrary processor is selected. We chose this strategy in order to reduce nonlocal accesses to nodes although it may lead to load imbalances.

All other distributions are based on the node and element distributions. These distributions are the boundary side distribution, boundary node distribution, cauchy side distribution, cauchy node distribution, neumann side distribution, neumann node distribution, variable side distribution, variable node distribution and dirichlet node distribution. All these objects are assigned to that processor which owns the original object, i. e. the element or the node.

According to the distributions the input data are rearranged and copied to the parallel file system. This enables each processor to read the information for its own array elements efficiently.

The developed tool, the *data partitioner*, is executed on one processor of the Paragon system. It reads the different distributions and the original input data of the sequential program. From these files it generates two files, one file on the host system containing the data replicated in all processors and another file containing the information of distributed arrays. In the second file the data of each distributed array are arranged into contiguous blocks for each processor.

3.2.1 Implementation of the Data Partitioning version

The Data Partitioning parallelization strategy leads to a code written according to the SPMD programming model (Single Program Multiple Data). Each processor executes the same program on the data which correspond to the assigned subregion.

The parallel code is parameterized in the number of processors as well as the distribution of nodes and elements. A deep understanding of the code was neccessary to find out where accesses to nonlocal objects occur.

The implementation is based on the PARTI library [1], developed at NASA/ICASE by Joel Saltz et al. It supports distributions of arrays, computation of processor–local indices, analysis of communication patterns, and communication of nonlocal array elements.

In a first phase of the parallel program, called the *inspector*, array distributions are specified in each processor via a list of the global array indices assigned to the processor. Based on the distributions, communication patterns and local indices are computed.

Each processor passes a list of the array elements for which it will be responsible (`locnd`) and the number of items in this list (`NNP`, the number of nodal points) to the inspector routine `IFBUILD_TRANSLATION_TABLE`.

```
ND_TTAB = IFBUILD_TRANSLATION_TABLE ( 1 , LOCND , NNP )
```

The call to `IFBUILD_TRANSLATION_TABLE` returns the translation table for the nodes: `ND_TTAB`. This table is used in another inspector routine, `FLOCALIZE`, to compute the local indices from the global ones and to compute the communication schedules to resolve nonlocal accesses. A call to `FLOCALIZE` looks like this:

```
CALL FLOCALIZE(ND_TTAB,ND_SCHED,OFFX,LOFFX,NOFFX,NON_LOC,NNP,JB)
```

On each processor P, `FLOCALIZE` is passed:

1. a pointer to a distributed translation table (`ND_TTAB`)
2. a list of global indices of distributed array elements that are accessed in processor P (`OFFX`), and
3. the number of global indices, `NOFFX`

`FLOCALIZE` returns:

1. a schedule that can be used in PARTI gather and scatter procedures (`ND_SCHED`) to resolve nonlocal accesses in OFFX,

2. a list of locally indexed array references for which processor P is responsible (LOFFX), and

3. the number of distinct off–processor references found in OFFX (NON_LOC).

The pre–computed communication schedules are then used in communication operations to perform the actual exchange of array element values. This is called the *executor* part. PARTI provides gather, scatter, and scatter_add operations to fetch, distribute and combine information for nonlocal elements.

For example, the GATHER routine has to be inserted in the code whereever nonlocal references appear. For example, array X stores the x–coordinate of the nodes. If processor P accesses local nodes as well as nonlocal nodes and thus array elements of X it has to allocate memory for the array elements for local nodes as well as for copies of nonlocal nodes. The doubleprecision FORTRAN version of GATHER is DFGATHER.

```
CALL DFGATHER ( ND_SCHED , X  ( NNP + 1 ) , X  ( 1 ) )
```

As the first parameter DFGATHER takes the precomputed schedule of the nodes (ND_SCHED), the second parameter is the buffer for nonlocal nodes X (NNP + 1), and the third parameter are the local nodes of the processor X (1).

Besides these operations on one–dimensional arrays also operations on two–dimensional arrays were needed in this application and were partly developed in cooperation with ICASE during the parallelization.

Example:

As an example we look at a typical loop of the program where such non–local read and write accesses occur.

```
CALL DFGATHER ( ND_SCHED , X ( NNP + 1 ) , X ( 1 ) )
  .
  .
DO M = 1, NEL
  DO IQ = 1, 8
    NI        = IEN ( M , IQ )
    XQ ( IQ ) = X ( NI )
  ENDDO

  DO IQ = 1, 8
    NI        = IEN ( M , IQ )
    VX ( NI ) = VX ( NI ) + QRX ( IQ )
  ENDDO
ENDDO
  .
  .
CALL DFSCATTER_ADD ( ND_SCHED , VX ( NNP + 1) , VX ( 1 ) )
```

The loop shown iterates over all elements (NEL is the number of elements). For each element IEN stores the node indices of those nodes which belong to the element. These nodes, however, are not

neccessarily local nodes, and a node can belong to different elements at the same time. Therefore non–local values of X have to be gathered before the loop starts, using the pre–computed schedule of the nodes (NODE_SCHED).

In the second inner loop some node information is added to vector VX. This information has to be made available to the processor who owns the non–local nodes by a scatter_add operation. It means that these non–local values are added to those values that the owner computed itself.

4 Comparison by a Single Test Case

To compare both methods, a simple flow domain was chosen with a geometry to enable the calculation of the whole domain on one processor of the Intel Paragon. The number of Finite Element nodes is 9216 with 6 x 6 x 256 in the three directions. The type of the boundaries is Dirichlet except for the top, where a variable Neumann/Dirichlet type with rainfall respectively evaporation is imposed. For simplicity a homogeneous permeability distribution was chosen. This test run performs 14 time steps. For more realistic runs simulation times of several hundred time steps have to be performed.

The overlap in the Domain Decomposition version is three elements with a one dimensional structure of four and eight subdomains while the size of the subdomains is exactly the same for each processor to assure a unique load balance distribution. According to the geometry the domain was subdivided in the z-direction. Note that in the example of Eq. (10) the x-direction was chosen. The distribution in the Data Partitioning version is also in blocks in the z-direction.

Table 1 presents the execution times, Table 2 the speedup, and Table 3 the efficiency of the two program versions. The first row are the values of the Domain Decomposition version (DD), the second row those of the Data Partitioning program (DP). Simulation runs of DD on more than eight processors were not feasible due to the manual generation of the input files. This is presently the most limiting step in the application of the Domain Decomposition method.

In the Data Partitioning version SETUP precomputes the communication patterns. Although there is no similar phase in the Domain Decomposition version we present this overhead separately in the timing table. This phase is executed once when the parallel program is started. Due to the long runtime of the time loop this overhead is neglectable but has to be taken into account for correctness.

The difference in execution time on one processor between DD and DP is mainly caused by a difference in output instructions in the time loop. DP is based on a cleanedup version of the original sequential code for the time loop. Both versions have some minor differences in one of the iteration loops in terms of output statements.

Comparing the speedup and efficiencies, the Data Partitioning method seems to be slightly better than the Domain Decomposition method. This is mainly caused by the redundant computations due to the overlap and by the additional effort needed in the Schwarz iterations. Clearly the linear equation has to be solved more frequently in DD than in DP. Improvement of the Domain Decomposition method can be obtained by imposing the convergence of the Schwarz iteration or by introducing a load balancing algorithm.

In comparison to DD, DP has a considerable amount of memory overhead due to the PARTI library. The additional memory is used for storing the distributions, computing the commu-

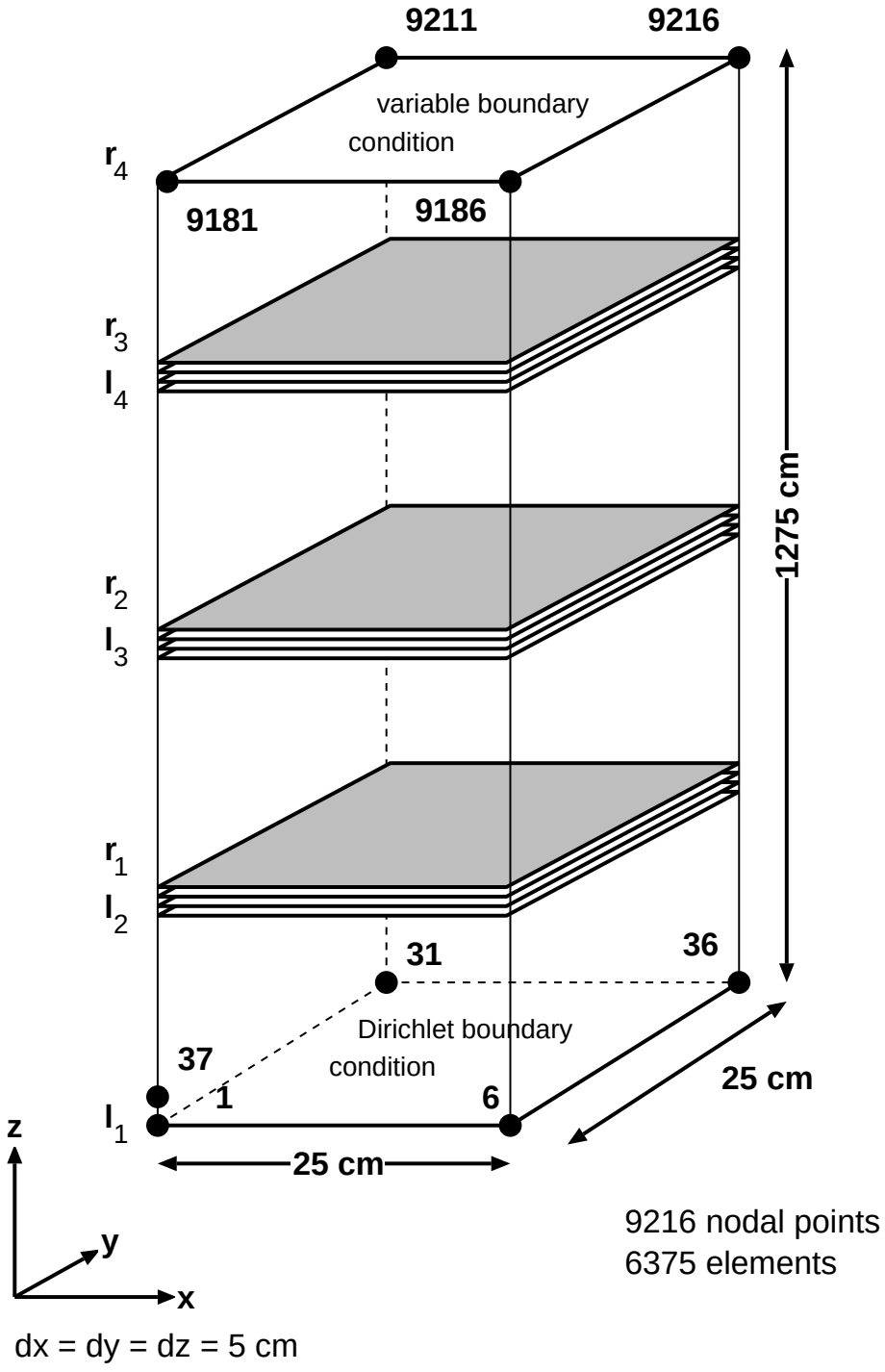

Figure 5: Test domain for 4 CPUs

Proc.		1	4	8	16	32
SETUP	*DD*	na	na	na	na	na
	DP	na	2	1	1	2
SPROP	*DD*	620	160	85		
	DP	622	156	78	39	20
ASEMBL	*DD*	881	228	121		
	DP	880	235	116	57	29
CGALG	*DD*	285	100	59		
	DP	274	82	46	29	22
time loop	*DD*	2069	567	315		
	DP	1948	517	264	139	80

Table 1: Execution time in seconds

Proc.		1	4	8	16	32
SPROP	*DD*	1.0	3.88	7.29		
	DP	1.0	3.99	7.95	15.95	31.10
ASEMBL	*DD*	1.0	3.86	7.28		
	DP	1.0	3.74	7.59	15.44	30.34
CGALG	*DD*	1.0	2.85	4.83		
	DP	1.0	3.34	5.96	9.45	12.45
time loop	*DD*	1.0	3.65	6.57		
	DP	1.0	3.77	7.38	14.01	24.35

Table 2: Speedup

Proc.		1	4	8	16	32
SPROP	*DD*	100	97	91		
	DP	100	100	99	100	97
ASEMBL	*DD*	100	97	91		
	DP	100	94	95	96	95
CGALG	*DD*	100	71	60		
	DP	100	84	74	59	39
time loop	*DD*	100	91	82		
	DP	100	94	92	88	76

Table 3: Efficiency

nication schedules, storing the schedules, and for buffering the communicated data. Since the memory is allocated dynamically a more precise estimation of its amount is impossible but the effect may not be neglected in the case of large flow domains.

At present only a simple case using a limited number of processors has been tested. For further comparison larger and more complex problems in terms of decomposition of the domain (two and three dimensional) will have to be examined.

5 Conclusion

In this study two different strategies for parallelization of a finite element code for water and solute transport in porous media are compared. Both the Domain Decomposition and the Data Partitioning method give very good speedups and efficiencies for the case examined. It opens the perspective of being able to solve large time dependent non-linear flow and transport problems with millions of unknowns on massively parallel computer systems.

For both methods the speedup for the solution of the linear equation is the limiting factor. In case of the Domain Decomposition this is caused by the need for an increased number of Schwarz iterations by increasing processor number. This results in a more frequent solution of the linear equation system. In the Data Partitioning method, the number of times the linear equation system is solved remains constant but the time for communication becomes more and more important.

Both parallelizations required a deep knowledge of the application and the applied algorithm. Although researchers are working on the automatic transformation of such codes to parallel programs in the context of HPF this knowledge will still be required for selecting the data distribution and for understanding the resulting performance of the application. Not only the knowledge of the application is required to under the application's performance but the user has to understand also the automatically carried out transformations. Although the Data Partitioning approach for this application is based on the same techniques, HPF compilers will only be able to generate as efficient code if they are able to perform agressive interprocedural optimizations which were easily performed during the manual parallelization.

Although the coding phase of the actual parallelization, i.e. insertion of communication and transformation of array declarations, took more time for the Data Partitioning version, both parallelizations were completely dominated by the task of parallelizing the I/O. The time invested in this task can be seen as overhead since the execution time for I/O is neglectable compared to the overall calculation time.

Parallelization of the I/O is unnecessary when the data structures are allocated in a global shared memory. In [8] we report on some parallelization experiments with this application for scalable shared memory machines. We developed parallel versions for the KSR and for a shared memory software implementation called KOAN on top of an IPSC/2. Our experiments showed speedups similar to those obtained for the Paragon.

Due to the enormous requirements of this application in computation time and memory we are currently investigating a meta-computing approach. We plan to connect a CM-5 at GMD/Bonn and our Paragon via a high-speed link and to run the Domain Decomposition version of TRACE in parallel on both machines.

References

[1] R. Das, J. Saltz, *A manual for PARTI runtime primitives - revision 2*, Internal Research Report, ICASE, 1992

[2] G. Rodrique, *Inner/outer iterative methods and numerical Schwarz algorithms*, Parallel Computing 2, 1985, 205-218

[3] H.A. Schwarz,*Gesammelte Mathematische Abhandelungen*, Vol.2, Springer Verlag, Berlin, 1890, 133-143

[4] H. Vereecken, G. Lindenmayr, A. Kuhr, D.H. Welte, A. Basermann, *Numerical modeling of field scale transport in heterogeneous variably saturated porous media*, KFA/ICG-4 Internal Report No. 500393, Juelich, Germany, 1993

[5] H. Vereecken, O. Neuendorf, G. Lindenmayr, A. Basermann, *A parallel Schwarz domain decomposition method for the numerical solution of transient water flow in heterogeneous porous media*, submitted to Parallel Computing 1994

[6] G.T. Yeh, *3DFEMWATER : A Three Dimensional Finite Element Model of Water Flow Through Saturated-Unsaturated Media*, Report ORNL-6386, Environmental Science Division Publ. 2904, Oak Ridge, USA, 1987

[7] HPFF, *High Performance Fortran Language Specification*, High Performance Fortran Forum, May 1993, Version 1.0, Rice University Houston Texas

[8] R. Berrendorf, M. Gerndt, Z. Labjomri, T. Priol, *A Comparison of Shared Virtual Memory and Message Passing Programming Techniques Based on a Finite Element Application*, to appear in: Proceedings of CONPAR'94, Linz