Deltares

MT3DMS, A Modular Three-Dimensional Multispecies Transport Model

User Guide to the Massively Parallel Processing (MPP) Package and PETSC (PET) Package

Jarno Verkaik
Arthur van Dam
Aris Lourens

1203355-003

Deltares

Title
MT3DMS, A Modular Three-Dimensional Multispecies Transport Model

Project	Reference	Pages
1203355-003	1203355-003-BGS-0001	24

Keywords
MT3DMS, groundwater flow, contaminant transport, parallel computing, distributed memory, MPI

Summary
This report is a user guide for using the distributed memory parallel processing packages in MT3DMS v5.30. It describes the numerical implementation, the program design and input instructions. For a large contaminant transport model it is shown that computational time can be reduced significantly (speedup factor 20 for this example).

References
-

Version	Date	Author	Initials	Review	Initials	Approval	Initials
1.0		J. Verkaik	JV	J. Griffioen	JG	H. Passier	HP
		A. van Dam	AD				
		A. Lourens	AL				

State
final

Contents

1 Introduction

Preserving a good groundwater and surface water quality is of great importance for water consumption, agriculture, and the environment. Groundwater and surface water bodies that provide us with clean water are threatened by a large range of contaminants, e.g. nutrients, pesticides, or heavy metals. Several national and international policy programs exist to prevent this happening, for example KRW, WB21, or Natura2000. In order to support the decision makers, accurate and fast simulation models are required for predicting contaminant transport. This involves very detailed models resulting in, inherently, very large computing times.

MT3DMS is a widely used, public-domain, code for modeling contaminant groundwater transport. The last two decades a large number of MT3DMS transport models have been developed at Deltares, becoming more and more detailed, and hence requiring more and more computing time. To speed up MT3DMS, a distributed memory parallelization approach is applied in this project that allows the code to run on high performance computers in a massively parallel way, i.e., using a very large number of processors. This report describes how this is realized.

The focus in this approach was on distributed memory parallelization of the Euler solvers, such as the TVD scheme, and not on the Eulerian-Lagrangian solvers (MOC), requiring different kind of parallelization techniques. Furthermore, parallelization of the implicit solver required some special attention, although this is of less importance for models where the contaminant flow is assumed to be advection dominated. From a software point of view, the MT3DMS code was changed at some points (e.g. GCG solver) although this was kept as minimal as possible. Two new modules/packages were added: the MPP package (massively parallel processing) and the PET package (interface to PETSc solver toolkit).

The layout of this report is as follows. chapter 2 describes the numerical implementation, followed by a brief program design in chapter 3 and input instructions in chapter 4. In chapter 5 results are presented for the test problems and a large application model.

2 Numerical implementation

In section 2.1, first some motivations are given for the chosen parallelization approach, followed by the partitioning strategy in section 2.2, and the data communication mechanism in section 2.3. Section 2.4 and 2.5 describe the parallelization approach for the GCG solver and PETSc solver, respectively. Section 2.6 concludes with a list of supported features.

2.1 Motivations for distributed parallel computing

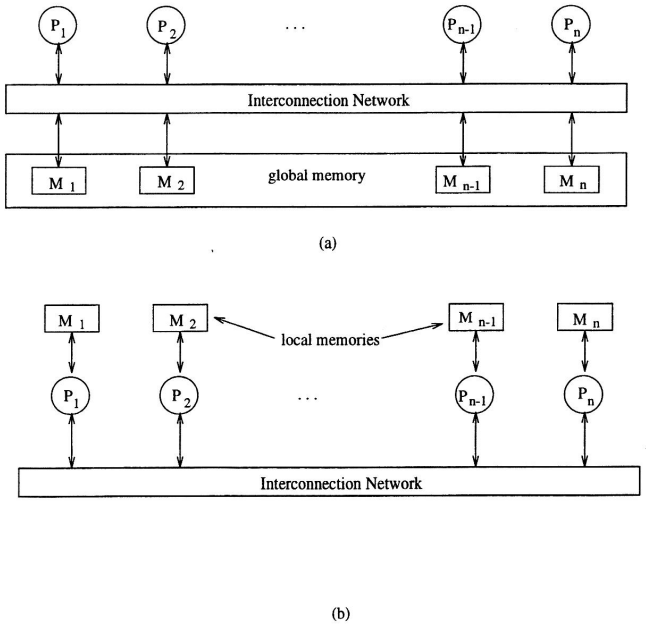

Figure 2.1: shared-memory (a) versus distributed memory (b).

This report will not go into detail on the wide spectrum of existing parallelization techniques. For an introduction, the reader is referred to further readings on parallel computing, e.g. Grama et. al (2003).

A commonly used difference in parallelization is made by memory organization: shared-memory versus distributed memory. In a shared-memory (or global memory) computer architecture (Figure 2.1a), each processor can access each memory location of the entire memory system. In a distributed-memory (or local memory), each processor has its own local memory as illustrated in Figure 2.1b. A processor can only access its own local memory, and only obtain data from another processor through communication using the interconnection network (the network that connects the processors with each other).

Shared memory parallelization is usually achieved by the OpenMP standard, and distributed memory by a Message Passing Interface (MPI) implementation, e.g. MPICH (Gropp et al. 2009). The shared memory approach usually involves implicit (or data) parallelization, meaning that computations are parallelized by setting compiler directives, e.g. for DO-loops in the code. The distributed memory approach usually involves explicit parallelization, meaning that the problem is physically distributed over different processes. Some advantages of distributed memory parallelization are:

- Divide-and-conquer is a natural approach for splitting the large problem into smaller sub-problems;
- Memory size is not limited;
- Use of a very large amount of processors with a good scalability (massively parallel processing);
- Upgrading hardware (clusters/super computers) with more processors is usually cheaper than for shared memory machines;
- MPI is portable to shared memory machines and multi-core processors.

Some downsides are:

- Speed of the network (interconnect) can be bottleneck for the performance (latency);
- Coding can be hard since communication between processors must be programmed explicitly;
- Implicit solver convergence rates can be an issue, especially for dispersion dominated groundwater flow.

Distributed memory parallelization is closely related to so-called domain decomposition methods, see e.g. www.ddm.org. Furthermore, the distributed memory approach does not exclude shared memory. Since coding is quite complementary, both methods can be combined in one single code. As a hybrid approach, for example, one could think of distributed MPI processes[1] where within each MPI process the sub-problem is solved on a multi-core processor using OpenMP.

2.2 Partitioning

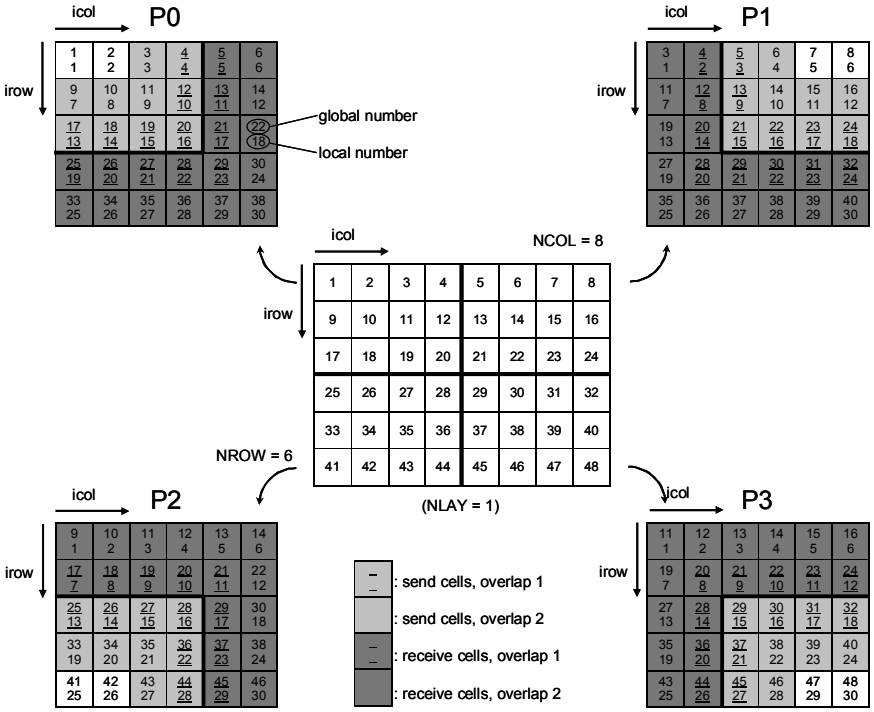

Figure 2.2: Example showing how a grid of eight columns and six rows is partitioned for four processes P0 – P3.

In the distributed approach, the MT3DMS grid is divided over the number of available processes, say $M = M_c \times M_r$, where M_c and M_r are the number of partitions in column- and row direction, respectively. Each partition has the same number of layers, since the assumption is that for large applications the number of rows and columns is much greater than the number of layers. The blocks are numbered for upper-left (P_0 Master process) to the lower right P_{M-1} (Slave processes). The partitioning is done such that a) each process has

1. *From now on, we will speak of communication between MPI processes, instead of processors. This is because more then one MPI processes can run on a single processor.*

more or less the same number of cells (work load) and b) the total length of interfaces between the processes is minimal. By this, each process has the same amount of work and the communication of data between the processes is minimal. The MPP package supports three options for basic partitioning:

1 Basic partitioning algorithm, uniform in column and row direction;
2 Basic partitioning algorithm correcting for active cells in column and row direction;
3 User defined partitioning.

These options are explained in more detail below. First, a non-overlapping partition is determined.

Basic partitioning algorithm: uniform

For example, consider a MT3DMS computational grid with $N^c = 8$ columns, $N^r = 6$ rows and $N^l = 1$ layer[2], as shown in Figure 2.2. Given four processes ($M = 4$), this shows the resulting partitions for the case $M_c = M_r = 2$. The partitioning algorithm used is very straightforward to determine M_c and M_r. First, all the candidate options (dividers) for M_c and M_r are determined such that $M = M_c \times M_r$ for a given M. For this example, there are three dividers $M_c \times M_r$: 1) 1 × 4, 2) 2 × 2, and 3) 4 × 1. Second, the optimal block size b is computed by $b = \text{int}(\sqrt{N / M})$, where $N = N^c \times N^r$. For this example $b = 3$. The smallest dimension, N^r in this case, is divided by b, resulting in the optimal number of blocks in row direction $M_r = 2$). Then, the most optimal divider is selected, hence $M_c = M_r = 2$. In this method, each block p has (assuming even dimensions for simplicity) $\hat{N}_p^c = N^c / M_c$ columns and $\hat{N}_p^r = N^r / M_r$ rows.

Basic partitioning algorithm, correction for active cells

Given the determined M_c and M_r from the above basic partitioning algorithm, the user can automatically optimize the block dimensions $\hat{N}_p^c$ and $\hat{N}_p^r$, and hence the load balance, by specifying an integer grid with weights for each cell (> 0 or 0). For example, consider Figure 2.2. for the case a weight grid is used having all ones, except the last two columns. This example corresponds to $N_{act} = 36$ active cells, where the last two columns contain inactive cells only. Then, the cumulative row sums are 6, 12, 18, 24, 30, 36, 36, 36 and cumulative column sums are 6, 12, 18, 24, 30, 36. For determining $\hat{N}_p^c$, a (active cell) block size is introduced by $b_{act} = \text{int}(N_{act} / M_c)$, which equals 18 for this example. For each column direction, $j = 1,...,M_c$, the index in the row sum array is searched that is closest to $j * b_{act}$ and this index we take as the end column. Hence, for this example this is the third column, shifting the vertical interface with one column to the left. The row direction is treated in a similar way.

2. *In the MT3DMS code these are referred to as NCOL, NROW, and NLAY, respectively.*

User defined partitioning
Instead of automatic partitioning, the user can specify the non-overlapping partitions entirely by input. This is done by specifying M_c and M_r, and the starting columns (M_c times) and starting rows (M_r times) for the partitions.

Amount of overlap
To couple the resulting blocks with each other, an amount of overlap has to be introduced. Because of the third-order stencil that is used for the advection TVD scheme, an overlap of two so-called ghost cells is added, see shaded cells in Figure 2.2 . Each non-overlapping partition $\widehat{\Omega}_p$ has $\widehat{N}_p = \widehat{N}_p^c * \widehat{N}_p^r * N^l$ grid cells, including ghost-cells. Introducing the overlap of two results in overlapping partitions Ω_p having $(\widehat{N}_p^c + 2\delta^E + 2\delta^W) * (\widehat{N}_p^r + 2\delta^N + 2\delta^S) * N^l$, where $\delta^E = 1$ if the block has a east neighbour, if not $\delta^E = 0$, etc. The overlapping dimensions for block p are denoted by $N_p^c = \widehat{N}_p^c + 2\delta^E + 2\delta^W$ for the number of columns, and $N_p^r = \widehat{N}_p^r + 2\delta^N + 2\delta^S$ for the number of rows. The total number of nodes are given by $N_p = N_p^c * N_p^r * N^l$. Each process is responsible for the computation of the cells in $\widehat{\Omega}_p$. To do this in parallel, each process needs to synchronize the concentrations in the ghost cells. The following section describes in more detail how this is accomplished.

2.3 Communication of data

Communication of data can be *local* or *global*. Local communication (LC) only involves the exchange of data between neighbouring processes. For MT3DMS, these data are concentrations for the TVD scheme and search directions for the implicit solver. Global communication (GC) involves the exchange of data for all processes (all-to-all, one-to-all, all-to-one). This is necessary for synchronizing transport time step, mass budgets, scaling factors and Euclidean norms in the implicit solver. In general, local communication is less expensive than global communication, especially for a large number of processors.

Local communication
Local communication is illustrated in Figure 2.2. Before computing the new concentration subject to advection, all processes first have to update their concentrations in the “receive” ghost cells (dark gray shaded). This is done by sending the concentrations for the “send” ghost cells that lie in the non-overlapping partition (light gray shaded). For example, Master process P0 sends to:

- P1 (east):
  - Overlap 1: cells 4, 12, 20
  - Overlap 2: cells 3, 11, 19
- P2 (south):
  - Overlap 1: cells 17, 18, 19, 20
  - Overlap 2: cells 9, 10, 11, 12
- P3 (South-East):
  - Overlap 1: 20
  - Overlap 2: 11, 12, 19

At the same time process P1 sends concentrations to P0 (West), P2 (South-East), P3 (South), process P2 sends to P0 (North), P1 (North-West), P3 (East), and process P3 sends to P0 (North-West), P1 (North) and P2 (West). After sending concentrations, P0 receives concentrations from three neighbours:

- P1 (east):
  - Overlap 1: cells 5, 13, 21
  - Overlap 2: cells 6, 14, 22
- P2 (south):
  - Overlap 1: cells 25, 26, 27, 28
  - Overlap 2: cells 33, 34, 35, 36
- P3 (South-East):
  - Overlap 1: 29
  - Overlap 2: 30, 37, 38

At the same time process P1 receives concentration from P0 (West), P2 (South-East), P3 (South), process P2 receives from P0 (North), P1 (North-West), P3 (East), and process P3 receives from P0 (North-West), P1 (North) and P2 (West).

Local communication is done with MPI using non-blocking send and non-blocking receive.

Global communication
Global communication is necessary to synchronize parameters that need to be evaluated over all processes, such e.g. the minimal transport time step. For this example, each process computes the transport time step satisfying the Courant stability constraint for its partition, sends this time step to all other processes (all-to-all communication), and each process computes the minimum and uses this value as the current transport time step ($\min(x_p),\ p = 0,...,M-1$). The all-to-all summation ($\sum_{p=0}^{M-1} x_p$) is used for the total mass and the inner products for vectors necessary for the implicit solver. The all-to-all maximum communication ($\max(x_p),\ p = 0,...,M-1$) is used in the implicit solver for the scaling and closure parameters. Furthermore, observation point data is gathered and written by the master process P0 as an user option (all-to-one communication).

2.4 GCG implicit solver

The Generalized Conjugate Gradient (GCG) implicit solver is by default used in MT3DMS for solving the linear algebraic system $\mathbf{Au} = \mathbf{b}$, where $\mathbf{A}$ is a general symmetric/non-symmetric coefficient matrix that is real and non-singular, and has a sparse structure containing 7- or 19 diagonals (full dispersion tensor). The vector $\mathbf{u}$ is the concentration solution vector to be solved, and $\mathbf{b}$ the right-hand side vector. The GCG solver is parallelized by a so-called Schwarz domain decomposition approach, see e.g. Saad (2000). Partitioning results in a reordering of coefficients, and for the example of $M = 6$ processes where $M_c = 2$ and $M_r = 3$, the linear system can be written as (assuming non-overlap notation for simplicity):

$$\mathbf{Au} = \left[\begin{array}{cc:cc:cc} \mathbf{A}_{0,0} & \mathbf{A}_{0,1}^{\mathrm{E}} & \mathbf{A}_{0,2}^{\mathrm{S}} & \mathbf{A}_{0,3}^{\mathrm{SE}} & & \\ \mathbf{A}_{1,0}^{\mathrm{W}} & \mathbf{A}_{1,1} & \mathbf{A}_{1,2}^{\mathrm{SW}} & \mathbf{A}_{1,3}^{\mathrm{S}} & & \\ \hdashline \mathbf{A}_{2,0}^{\mathrm{N}} & \mathbf{A}_{2,1}^{\mathrm{NE}} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3}^{\mathrm{E}} & \mathbf{A}_{2,4}^{\mathrm{S}} & \mathbf{A}_{2,5}^{\mathrm{SE}} \\ \mathbf{A}_{3,0}^{\mathrm{NW}} & \mathbf{A}_{3,1}^{\mathrm{N}} & \mathbf{A}_{3,2}^{\mathrm{W}} & \mathbf{A}_{3,3} & \mathbf{A}_{3,4}^{\mathrm{SW}} & \mathbf{A}_{3,5}^{\mathrm{S}} \\ \hdashline & & \mathbf{A}_{4,2}^{\mathrm{N}} & \mathbf{A}_{4,3}^{\mathrm{NE}} & \mathbf{A}_{4,4} & \mathbf{A}_{4,5}^{\mathrm{E}} \\ & & \mathbf{A}_{5,2}^{\mathrm{NW}} & \mathbf{A}_{5,3}^{\mathrm{N}} & \mathbf{A}_{5,4}^{\mathrm{W}} & \mathbf{A}_{5,5} \end{array}\right] \left[\begin{array}{c} \mathbf{u}_0 \\ \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \mathbf{u}_4 \\ \mathbf{u}_5 \end{array}\right] = \left[\begin{array}{c} \mathbf{b}_0 \\ \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \\ \mathbf{b}_4 \\ \mathbf{b}_5 \end{array}\right] = \mathbf{b}\,, \tag{1}$$

where $\mathbf{A}_{i,i}$ is the interior coefficient matrix for block i, $\mathbf{A}_{i,j}^{*}, i \neq j$ are the connection matrices, where the superscript * denotes the interface. For example, $\mathbf{A}_{0,1}^{\mathrm{E}}$ means that block 0 has an Eastern interface to block 1. For the case of a 7-point stencil (no full dispersion tensor), the connection matrices with superscript SE, SW, NE, NW are all equal zero. The vectors $\mathbf{u}_i$ and $\mathbf{b}_i$ are the computed concentration vector and right-hand side vector on block i, respectively. In the additive Schwarz method, the (left) preconditioned system $\mathbf{Q}^{-1}\mathbf{Au} = \mathbf{Q}^{-1}\mathbf{b}$ is solved by choosing the parallel preconditioner $\mathbf{Q}$ the block-diagonal of $\mathbf{A}$ (block-Jacobi). This means that the subdomain solutions, hence solutions of the form $\mathbf{Q}_{i,i}\mathbf{v}_i = \mathbf{A}_{i,i}\mathbf{v}_i = \mathbf{w}_i$ can entirely be solved in parallel. The resulting GCG-Schwarz method with block-Jacobi preconditioning uses the GCG preconditioners (Jacobi, SOR, MC) to solve the subdomain problem $\mathbf{A}_{i,i}\mathbf{v}_i = \mathbf{w}_i$ with only one preconditioning step, and hence inaccurately (Brakkee, et al., 1998). A derivation of the GCG algorithm is given by Jea and Young (1983)[3].

Parallelization of the GCG solver only involves a few basic operations:

- Applying a mask array for computing inner (vector-vector) products
  $$\langle \mathbf{x}, \mathbf{x} \rangle_{\mathrm{mask}} := \sum_{i=1}^{N_p} m x_i^2, \quad m(i) = \begin{cases} 1, & i \in \widehat{\Omega}_p \\ 0, & \text{else} \end{cases},$$
  where N_p is the number of nodes including the ghost cells, and $\widehat{\Omega}_p$ is the non-overlapping partition.
- Local communication (LC): exchange vectors for ghost cells.
- Global communication (GC): global sum, maximum over all processes

The GCG algorithm (CG and Lanczos/ORTHOMIN) in pseudo-code is given for each process by highlighting the MPP modifications:

$\gamma = \max_i |u_i^{(0)}|$; GC: maximum of γ;

Scale: $\mathbf{b} := \mathbf{b}\gamma^{-1}$; $\mathbf{u}^{(0)} := \mathbf{u}^{(0)}\gamma^{-1}$;

$\|\mathbf{b}\|_2^2 = \langle \mathbf{b}, \mathbf{b} \rangle_{\mathrm{mask}}$;

$\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{Au}^{(0)}$; LC of $\mathbf{r}^{(0)}$;

3. *Here, the original paper is referred to and not the MT3DMS manual since the manual has errors in the presented algorithms.*

$\left\|\mathbf{r}^{(0)}\right\|_2^2 = \left\langle \mathbf{r}^{(0)}, \mathbf{r}^{(0)} \right\rangle_{\text{mask}}$;

GC: sum of $\left\|\mathbf{b}\right\|_2^2$ and $\left\|\mathbf{r}^{(0)}\right\|_2^2$;

$\left\|\mathbf{b}\right\|_2 := \sqrt{\left\|\mathbf{b}\right\|_2^2}$; $\left\|\mathbf{r}^{(0)}\right\|_2 := \sqrt{\left\|\mathbf{r}^{(0)}\right\|_2^2}$;

Stopping test: $\left\|\mathbf{r}^{(0)}\right\|_2 / \left\|\mathbf{b}\right\|_2 < \min(1E-6, \text{CCLOSE})$;

Solve $\mathbf{Q}\boldsymbol{\delta}^{(0)} = \mathbf{r}^{(0)}$; LC of $\boldsymbol{\delta}^{(0)}$;

if ($\mathbf{A}$ non-symmetric) **then**

$\mathbf{p}^{(0)} = \boldsymbol{\delta}^{(0)}$; $\boldsymbol{\eta}^{(0)} = \boldsymbol{\delta}^{(0)}$; $\mathbf{q}^{(0)} = \boldsymbol{\delta}^{(0)}$;

$\widetilde{\alpha}_0 = \left\langle \boldsymbol{\delta}^{(0)}, \boldsymbol{\eta}^{(0)} \right\rangle_{\text{mask}}$; GC: sum of $\widetilde{\alpha}_0$;

$n = 0$;

while $\varepsilon > \text{CCLOSE}$ **do**

$\widetilde{\alpha}_n = \left\langle \boldsymbol{\delta}^{(n)}, \boldsymbol{\eta}^{(n)} \right\rangle_{\text{mask}}$; GC: sum of $\widetilde{\alpha}_n$

if n > 0 **then**

$\alpha_n = \dfrac{\widetilde{\alpha}_n}{\widetilde{\alpha}_{n-1}}$;

$\mathbf{p}^{(n)} = \boldsymbol{\delta}^{(n)} + \alpha_n \mathbf{p}^{(n-1)}$;

$\mathbf{q}^{(n)} = \boldsymbol{\eta}^{(n)} + \alpha_n \mathbf{q}^{(n-1)}$;

end if

Solve $Q\tilde{\boldsymbol{\delta}} = A\mathbf{p}^{(n)}$;

Solve $Q^T \tilde{\boldsymbol{\eta}} = \mathbf{q}^{(n)}$;

LC of $\tilde{\boldsymbol{\delta}}$ and $\tilde{\boldsymbol{\eta}}$;

$\hat{\lambda} = \left\langle \tilde{\boldsymbol{\delta}}, \mathbf{q}^{(n)} \right\rangle_{\text{mask}}$; GC: sum of $\hat{\lambda}$;

$\lambda_n = \dfrac{\widetilde{\alpha}_n}{\hat{\lambda}}$;

$\mathbf{u}^{(n+1)} = \mathbf{u}^{(n)} + \lambda_n \mathbf{p}^{(n)}$;

$\boldsymbol{\delta}^{(n+1)} = \boldsymbol{\delta}^{(n)} - \lambda_n \tilde{\boldsymbol{\delta}}$;

$\boldsymbol{\eta}^{(n+1)} = \boldsymbol{\eta}^{(n)} + \lambda_n A^T \tilde{\boldsymbol{\eta}}$;

$\varepsilon = \max_i \left| u_i^{(n+1)} - u_i^{(n)} \right|_{\text{mask}}$; GC: maximum of ε;

$n = n + 1$;

end do

else if ($\mathbf{A}$ is symmetric positive definite)

$\mathbf{p}^{(0)} = \boldsymbol{\delta}^{(0)}$;

$n = 0$;

while $\varepsilon > \text{CCLOSE}$

if n > 0 **then**

$\widehat{\alpha} = \left\langle \boldsymbol{\delta}^{(n)}, \tilde{\boldsymbol{\delta}} \right\rangle_{\text{mask}}$; GC: sum of $\widehat{\alpha}$;

$\alpha_n = -\dfrac{\widehat{\alpha}}{\tilde{\lambda}_{n-1}}$;

$\mathbf{p}^{(n)} = \boldsymbol{\delta}^{(n)} + \alpha_n \mathbf{p}^{(n-1)}$;

end if

$\widehat{\lambda} = \left\langle \boldsymbol{\delta}^{(n)}, \mathbf{p}^{(n)} \right\rangle_{\text{mask}}$; GC: sum of $\widehat{\lambda}$;

Solve $Q\tilde{\boldsymbol{\delta}} = A\mathbf{p}^{(n)}$;

LC of $\tilde{\boldsymbol{\delta}}$;

$\tilde{\lambda}_n = \left\langle \mathbf{p}^{(n)}, \tilde{\boldsymbol{\delta}} \right\rangle_{\text{mask}}$; GC: sum of $\tilde{\lambda}_n$;

$\lambda_n = \dfrac{\widehat{\lambda}}{\tilde{\lambda}_n}$;

$\mathbf{u}^{(n+1)} = \mathbf{u}^{(n)} + \lambda_n \mathbf{p}^{(n)}$;

$\boldsymbol{\delta}^{(n+1)} = \boldsymbol{\delta}^{(n)} - \lambda_n \tilde{\boldsymbol{\delta}}$;

$\varepsilon = \max_i \left| u_i^{(n+1)} - u_i^{(n)} \right|_{\text{mask}}$; GC: maximum of ε;

$n = n + 1$;

end do

end if

Unscale: $\mathbf{b} := \mathbf{b}\gamma$; $\mathbf{u}^{(n+1)} := \mathbf{u}^{(n+1)}\gamma$;

The current implementation of the GCG solver is inefficient for the case $\mathbf{A}$ is a diagonal matrix and the flow is inherently explicit. In this case, $\mathbf{A}$ can be easily inverted and new concentration entries of $\mathbf{u}^{(1)}$ can be directly computed by simply $u_i^{(1)} = b_i / a_{i,i}, i = 1,...N_p$ for all non-zero diagonal entries $a_{i,i}$. This can significantly save computing time since this avoids unnecessary initialization overhead for the preconditioning.

2.5 PETSc implicit solver

Instead of the parallel GCG solver, a parallel PETSc solver can be selected if this is desired (Balay et al., 2008). PETSc stands for Portable, Extensible Toolkit for Scientific Computation, and this toolkit contains a wide range of parallel solver packages. PETSc has been widely used for more than two decades now for a large number of application fields.

For MT3DMS, an interface to PETSc 3.1 is constructed, using the KSP (Krylov Subspace Methods) package. For the case $\mathbf{A}$ is symmetric positive definite, the CG method is used, otherwise, the commonly used GMRES (Generalized Minimal Residual method, Saad, 2000). In PETSc, these methods are set by KSPCG and KSPGMRES, respectively. By default, the GMRES method is restarted after 100 inner iterations. The parallel preconditioner $\mathbf{Q}$ is chosen block-Jacobi, which is similar to the parallel GCG solver. The subdomain solve $\mathbf{A}_{i,i}\mathbf{v}_i = \mathbf{w}_i$ is obtained by an incomplete LU factorization with zero fill-on (ILU(0)). Similar to the parallel GCG solver, the subdomain solution is obtained by preconditioning only.

The general set-up for PETSc is as follows:

1. Initialize PETSc variables;
2. Build linear system;
3. Set solver settings;
4. Solve linear system.

Special attention was needed for:

- *Stopping test*. The GCG stop criterion is based on the ℓ_1-norm of the difference in solution, PETSc is by default based on the ℓ_2-norm of the residual. Although the PETSc manual states that the stop criterion can be customized by the KSPSetConvergenceTest object, it did not seem to be possible to reproduce the GCG stopping test. Hence, PETSc converges if $\left\|\mathbf{r}^{(k)}\right\|_2 < \max(\mathrm{CCLOSE} * \|\mathbf{b}\|_2, 10^{-50})$, and diverges if $\left\|\mathbf{r}^{(k)}\right\|_2 > 10^5 * \|\mathbf{b}\|_2$.
- *Node numbering.* PETSc requires that the linear system is filled according to Eq. (1), meaning that besides the MT3DMS node numbering n^1 (referred to as global node number in Figure 2.2), and the local partioning node numbering n^2 (referred to as local node number in Figure 2.2), a third node numbering n^3 is introduced by $n^3 = n^2 + \sum_{i=0}^{p-1} \widehat{N}_i$ for partition p and $\widehat{N}_i$ is the number of non-overlapping nodes.
- *Dry or inactive cells.* Cells that are inactive are treated such that the corresponding matrix row i of $\mathbf{A}$ only has a diagonal entry set to -1, and $u_i = b_i = u_i^{(0)}$ for the inactive cell i. For each flow time step, cells can become dry or inactive as determined by Flow Model Interface package. If this happens, the matrix connectivity's change and this must be re-evaluated.

2.6 Supported features

Table 2.1 shows for each MT3DMS package if it is supported or not in combination with the MPP package.

Table 2.1: Supported and not supported packages for MPP.

Package	Supported	Restrictions
Basic Transport (BTN) package	Y	
Advection (ADV) package	Y	particle tracking MOC, MMOC or HMOC not supported
Dispersion (DSP) package	Y	
Sink and Source Mixing (SSM) package	Y	recirculation well option not supported
Chemical Reaction (RCT) Package	Y	
Generalized Conjugate Gradient (GCG) package	Y	
Flow Model Interface Package (LKMT3)	Y	Multi-node Well (MNW) package not supported
Transport Observation (TOB) package	N	
Hydrocarbon Spill Source (HSS) package	N	

3 Program design

The MT3DMS v5.30 code is modified to support distributed memory parallelization (MPP package) and parallel PETSc solvers (PET package). Code adjustments are kept as minimal as possible, to remain the modularity and such that the code can be merged with the newer MT3DMS versions. Table 3.1 summarizes the code changes, what parts of the code these involve, and the specific reasons for changing. In Table 3.2 – Table 3.7 the new subroutines and functions are described. Figure 3.1 – Figure 3.4 show flowcharts of the main program and the implicit solvers. Some code adjustments (CA) require further explanation, see Table 3.1 for the numbering.

CA 1-6 (MPP initialization)
These code adjustments are made for the initialization of the MPP package. The basic idea behind these code adjustments is that all MPI processes execute until the point where the grid dimensions are read (BTN5DF), “rewind” the program and then the partitioning takes place. Figure 3.1 shows how this is accomplished. First, all MPI processes do exactly the same: returning from MPPINI1 they have the flag MPPTYP set to MPPINI1, meaning the beginning MPP phase 1. In this phase, the program files are first opened in BTN5OPEN and unit numbers are stored in a temporary array (LIST output file is labelled negative since this file need to be removed later on). Then, the partitioning options are read from INMPP in MPP1INI2. After reading the grid dimensions NCOL and NROW in BTN5DF, all processes call MPP1INI3, where they rewind INMPP and close all other files. Besides closing, the Master process deletes the LIST file. After all processes return with logical MPPREST being true, this is the end of MPP initialization phase 1, and the start of MPP phase 2 (MPPTYP = MPPTYP2). All processes return in the main program to continue label 100, and enter the partitioning subroutine MPP1PART, where they determine their partition and their own NCOL and NROW. In BTN5DF these are set by calling the subroutine MPP1SETRCL, right after the (global) NCOL and NROW have been read.

Although this way of initialization seems a bit cumbersome, it is flexible when the code is extended with a more complex partitioning algorithm, like e.g. MeTiS, see Karypis and Kumar, 1998. In this case, a possibility is that MPP phase 1 is only executed by the Master process and the Master process does the partitioning using calls to the MeTiS library, and partition data files are written for each process. After this, all processes continue with MPP phase 2, and each process reads its MeTiS partition data file in MPP1PART.

CA 24, 38-40 (file readers)
The file readers were modified to support clipping read of data arrays and reading point data.

Table 3.1: Code adjustments of the MT3DMS code v5.30.

Nr.	Code adjustment	Subroutine(s) / Function(s)	Reason
1	MPPREST logical	main program	flag introduced for MPP inititalization
2	subroutine call added to MPP1PART	main program	partitioning of the grid in row and column direction
3	subroutine call added to MPP1INI1	main program	MPI initialize, set flags, etc.
4	subroutine call added to MPP1LXCHINI	main program	initialize local communication data structures
5	subroutine call added to MPP1INI2	main program	read data from input files
6	subroutine call added to MPP1INI3	main program	close and rewind input files
7	INMPP and INPET added	main program	packages MPP and PET added
8	subroutine call added to PET1AL	main program	allocate space or data arrays needed by the PET package
9	check for GCG solver	main program	changed in general check for implicit solver (PETSc solver can also be selected)
10	subroutine call added MPP1RP	main program	MPP package read and prepare: only write to IOUT
11	subroutine call added PET1RP	main program	PETSc package read and prepare
12	subroutine call added MPP1LXCH	main program	local exchange of concentrations CNEW
13	subroutine call added PET1AP	main program	solve PETSc linear system for obtaining new concentrations CNEW
14	subroutine call added MT_MPIFINALIZE	main program	finalize MPI processes
15	WRITESTO logical added	ADV5AL, BTN5RP, BTN5AD, BTN5OT, main program	to assure that only Master process P0 writes to standard output
16	MPP check added (MPP1CHECK)	ADV5AL, BTN5AL, FMI5AL, HSS5AL, RCT5AL, SSM5AL, TOB5AL, GCG5AP	check for unsupported MPP features
17	nodal mask aplied to RMASIO by call MPP1MASK	ADV5SV, SADV5F, ADV5BD, SADV5U, BTN5BD, DSP5BD, RCT5BD, SSM5BD	only account for cells for which the process is responsible
18	nodal mask aplied to TMASS by call MPP1MASK	BTN5AD	only account for cells for which the process is responsible
19	IF-statement splitting	SADV5M, SADV5F, SADV5U, SADV5Q, SAVD5F, SSM5FM, SSM5BD	changed to solve boundary check error for Intel compiler
20	MPP and PET packages added	BTN5OPEN	add MPP and PET packages to the name file
21	subroutine call added to MPP1STORELUN	BTN5OPEN	MPP phase 1: store unit numbers
22	subroutine call added to MPP1FNAME	BTN5OPEN, BTN5RP	MPP phase 2: append file name with process ID (LIST, UCN, CNF, OBS optional)
23	subroutine call added to MPP1SETRCL	BTN5DF	MPP phase 2: set local number of columns and rows
24	point read changed by call to new subroutine CHKLOC	BTN5RP, READPS, READGS, SSM5RP, TOB5RP	read clipped point data: check if used and renumber coordinates
25	subroutine call added to MPP1READOBS	BTN5RP	MPP phase 2: store observation global node numbers (optional merge option)
26	subroutine call added to MPP1WRTOBSHDR	BTN5RP	MPP phase 2: write global node numbers of observation nodes (optional merge option)
27	subroutine call added to MPP1GXCHTIME	BTN5AD	global exchange to compute minimum of transport time step DTRANS
28	subroutine call added to MPP1GXCHMASS1	BTN5AD	global exchange to compute sum of mass TMASS
29	subroutine call added to MPP1GXCHMASS2; changed code order	BTN5BD	combined global exchange to compute sum of masses TMASS en RMASIO
30	subroutine call added to MPP1GXCHOBS	BTN5OT	Master process gathers observation point data from Slaves and writes to output file (optional merge option)
31	subroutine call added to PET1ICBUNDCHG	FMI5RP	store flag indicating that ICBUND had changed (PET package only)
32	diagonal check added: convergence check and call to DIAGMATCHK	GCG5AP	check for diagonal matrix and if true, directly compute CNEW and return
33	subroutine calls added to MPP1CHMAX	GCG5AP	global maximum of SCALE or CHANGE
34	subroutine calls added to MPP1MASKN	GCG5AP	skip summation for cells in overlap
35	subroutine calls added to MPP1LXCH	GCG5AP	local exchange of vectors (search direction)
36	subroutine calls added to MPP1GXCHIP	GCG5AP	global sum of iterior products (vector-vector)
37	dimension checka skipped for FMI package	READHQ, READDS, READPS	check for grid dimensions skipped in case of MPP
38	real array read changed by call to new subroutine RDARRAYR	READHQ, READDS, RARRAY	clipped read of real arrays
39	integer array read changed by call to new subroutine RDARRAYI	READDS, IARRAY	clipped read of integer arrays
40	IOUT > 0 added as a condition to write to IOUT	IARRAY	control output for MPI processes

Table 3.2: New subroutines and functions created for the MPP package (file mt_mpp1.for).

Name	Description
MPP1CHECK	subroutine: check for features that are not supported with MPP
MPP1FNAME	subroutine: append output file name with process rank ID
MPP1GNODE2RANK	function: get process rank ID for the process that is reponsible for a MT3DMS global node number
MPP1GXCHDIAG	subroutine: global sum of integer flag indicating if a matrix is diagonal; used for GCG and PETsc solver
MPP1GXCHIP	subroutine: global sum over all processes used for computing iterior products in GCG solver
MPP1GXCHMASS1	subroutine: global sum of mass over all processes used for TMASS
MPP1GXCHMASS2	subroutine: global sum of mass TMASS and RMASIO and all processes
MPP1GXCHMAX	subroutine: global maximum over all processes used for SCALE and CHANGE in GCG solver
MPP1GXCHOBS	subroutine: Master process gathers observation nodal data from Slave processes and writes to IOBS
MPP1GXCHTIME	subroutine: global minimum over all processes used for DTRANS
MPP1INI1	subroutine: initialize MPI, set MPPTYP integer flag, set WRITESTO logical flag, initialize arrays to store unit numbers
MPP1INI2	subroutine: read data from INMPP, INADV for MIXELM, INGCG for NCRS and INPET for NCRS
MPP1INI3	subroutine: first slave processes rewind INMPP, INADV, INGCG, INPET, and closes other files second Master processes does the same but removes LIST file
MPP1LOADEST	subroutine: estimate work load based on simple counting of active cells
MPP1LXCH	subroutine: local communication using non-blocking send and receive
MPP1LXCHINI	subroutine: initialize local communication data structures: exchange partners, ranks, interface types, nodes
MPP1MASK	subroutine: determine from its local coordinates if process is responsible for this node
MPP1MASKN	subroutine: determine from its local node number if process is responsible for this node
MPP1PART	subroutine: main partitioning subroutine to partitionize the grid in row and column direction
MPP1PARTBAL	subroutine: partioning algorithm by load balancing with pointer grid defined by user and using row and column sums
MPP1PARTDEF	subroutine: default partioning algorithm uniform in row and column direction
MPP1PARTUSER	subroutine: partitioning where the user has specified the blocks
MPP1RP	subroutine: estimate work load; write partition information to output file IOUT
MPP1SETRCL	subroutine: set new NLAY, NROW and NCOL; set clipping area parameters for clipped data read
MPP1STORELUN	subroutine: store unit numbers that are to be closed or rewinded later on
MPP1STOREOBS	subroutine: store observation nodes global MT3DMS node numbers and responsible process rank
MPP1WRTOBSHDR	subroutine: write file header for observation points in IOBS
MPP1XPIDX	subroutine: determine local communication index arrays

Table 3.3: New FORTRAN subroutines and functions created for the PET package (file mt_pet1.for).

Name	Description
PET1AL	subroutine: allocate space for data arrays
PET1AP	subroutine: solve linear system using PETSc library to obtain new concentrations
PET1G2LNODE	function: determine global MT3DMS node number from local node number
PET1ICBUNDCHG	subroutine: check if icbund has changes and hence the grid cell connectivities
PET1L2GNODE	function: determine local node number from global MT3DMS node number
PET1L2PETGNODE	function: determine global PETSC node number from local node number
PET1PETIDX	subroutine: determine connectity indices for the linear system
PET1RP	subroutine: read and prepare data

Table 3.4: New C routines created for the PET package (file mt_pet1.c).

Name	Description
PET1PETCOEFINI	PETSc initialize, allocate PETSc matrix, solution vector and right-hand side vector
PET1PETCOEFINI	fortran interface wrapper for PET1PETCOEFINI
PET1PETCOEFINI	fortran interface wrapper for PET!PETCOEFINI
PET1PETSETCOEF	fill PETSc matrix, initial solution vector and right-hand side vector; assemble system
PET1PETSETCOEF_	fortran interface wrapper for PET1PETSETCOEF
PET1PETSETCOEF_	fortran interface wrapper for PET1PETSETCOEF
PET1PETSOLSET	set PETSc solver settings (CG or GMRES, block-Jacobi, ILU(0), stop criterion, output)
PET1PETSOLSET	fortran interface wrapper for PET1PETSOLSET
PET1PETSOLSET	fortran interface wrapper for PET1PETSOLSET
PET1PETSOLVE	PETSc solve linear system; error handling
PET1PETSOLVE	fortran interface wrapper for PET1PETSOLVE
PET1PETSOLVE__	fortran interface wrapper for PET1PETSOLVE

Table 3.5: Subroutines and functions added to the MT3DMS utilities (mt_utl5.for).

Name	Description
CFN_FINDWORD	subroutine: find word in string array
CFN_LENGTH	function: wrapper for CFN_LENGTH2
CFN_LENGTH2	function: determine string length
CFN_S_TRIM	subroutine: wrapper for CFN_TRIM2
CFN_S_TRIM2	subroutine: trim a string
CFN_UPCASE	subroutine: wrapper for CFN_UPCASE2
CFN_UPCASE2	subroutine: make a string uppercase
CHKLOC	subroutine: read clipped point data and check if used and renumber coordinates
CHKNOTSUPPORTED	subroutine: check for supported packages for clipping read
DEFRDSZ	subroutine: define clipping area parameters
GETIRCL	subroutine: get coordinates from MT3DMS global node number
RDARRAYI	subroutine: integer array reader suitable for clipping data
RDARRAYR	subroutine: real array reader suitable for clipping read

Table 3.6: Subroutine added to the GCG5 package (mt_gcg5.for).

Name	description
DIAGMATCHK	subroutine: check if coefficient matrix only has diagonal entries if so sompute cnew(n) = rhs(n) / a(n,1)

Table 3.7: MPI wrapper routines needed by the MPP and PETSc package (mt_mpp1_mpi.for).

Name	Description
MTMPI_ALLMAXI	subroutine: wrapper for MPI_ALLREDUCE to maximize integer values over all processes
MTMPI_ALLMAXR	subroutine: wrapper for MPI_ALLREDUCE to maximize real values over all processes
MTMPI_ALLMINR	subroutine: wrapper for MPI_ALLREDUCE to minimize real values over all processes
MTMPI_ALLSUMR	subroutine: wrapper for MPI_ALLREDUCE to sum real values over all processes
MTMPI_BARRIER	subroutine: wrapper for MPI_BARRIER to block until all processes reach this subroutine
MTMPI_COMM_RANK	function: wrapper for MPI_COMM_RANK to determine own process rank ID
MTMPI_COMM_SIZE	function: wrapper for MPI_COMM_SIZE te determine number of processes
MTMPI_ERROR	subroutine: MPI error message handling
MTMPI_FINALIZE	subroutine: wrapper for MPI_FINALIZE to finalize MPI
MTMPI_GATHERVI	subroutine: wrapper for MPI_GATHERV to gather integer arrays from all processes to one process
MTMPI_GATHERVR	subroutine: wrapper for MPI_GATHERV to gather real arrays from all processes to one process
MTMPI_INIT	subroutine: call MPI_INIT; zero communication buffers; get NRPROC and MYRANK
MTMPI_IRECVI	subroutine: wrapper for MPI_IRECV for non-blocking receive of an integer array
MTMPI_IRECVR	subroutine: wrapper for MPI_IRECV for non-blocking receive of an real array
MTMPI_ISENDI	subroutine: wrapper for MPI_ISEND for non-blocking send of an integer array
MTMPI_ISENDR	subroutine: wrapper for MPI_ISEND for non-blocking send of a real array
MTMPI_WAITALL	subroutine: wrapper for MPI_WAITALL to wait for all communications to complete

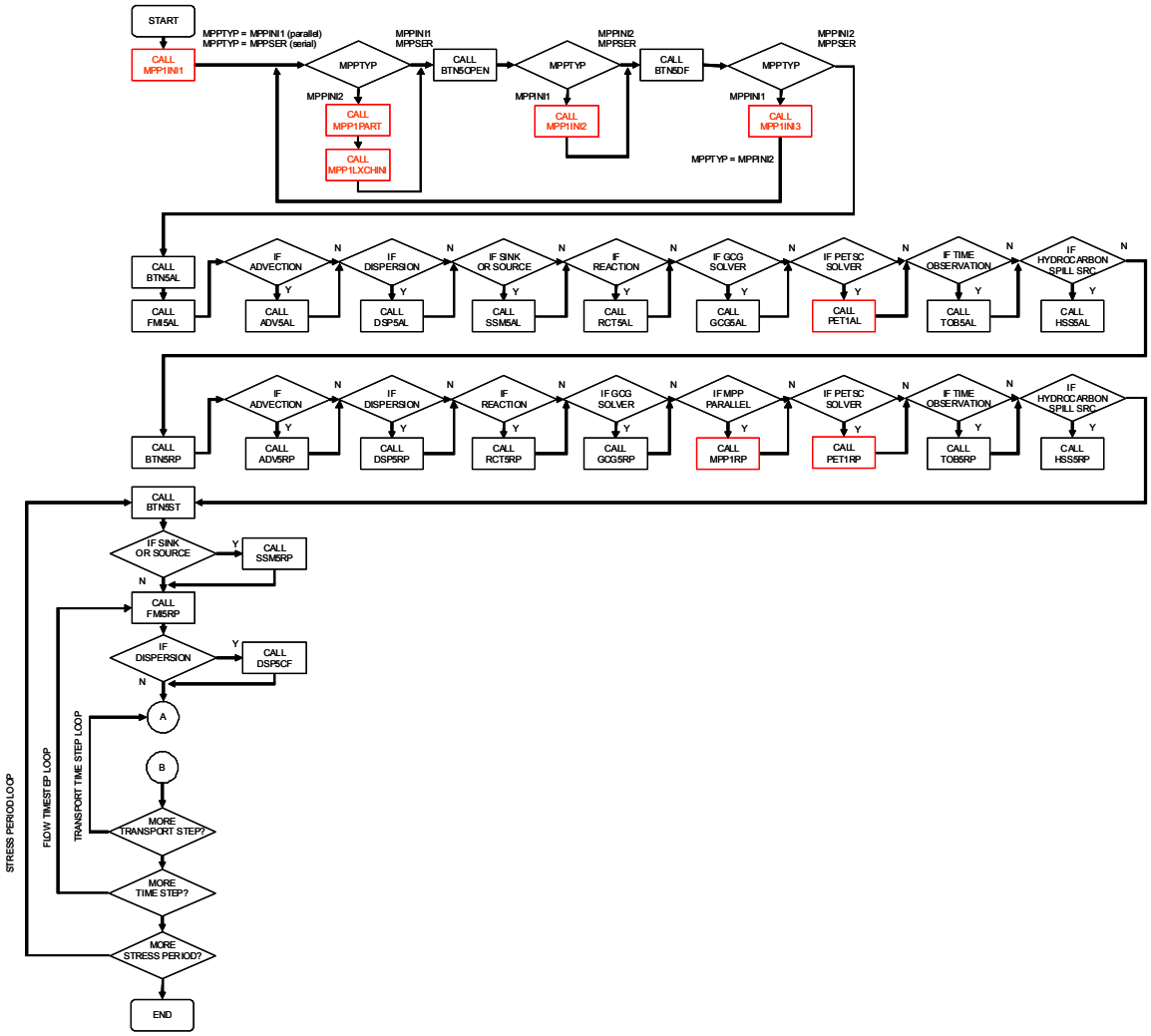

Figure 3.1: Main flow chart of MT3DMS (continued).

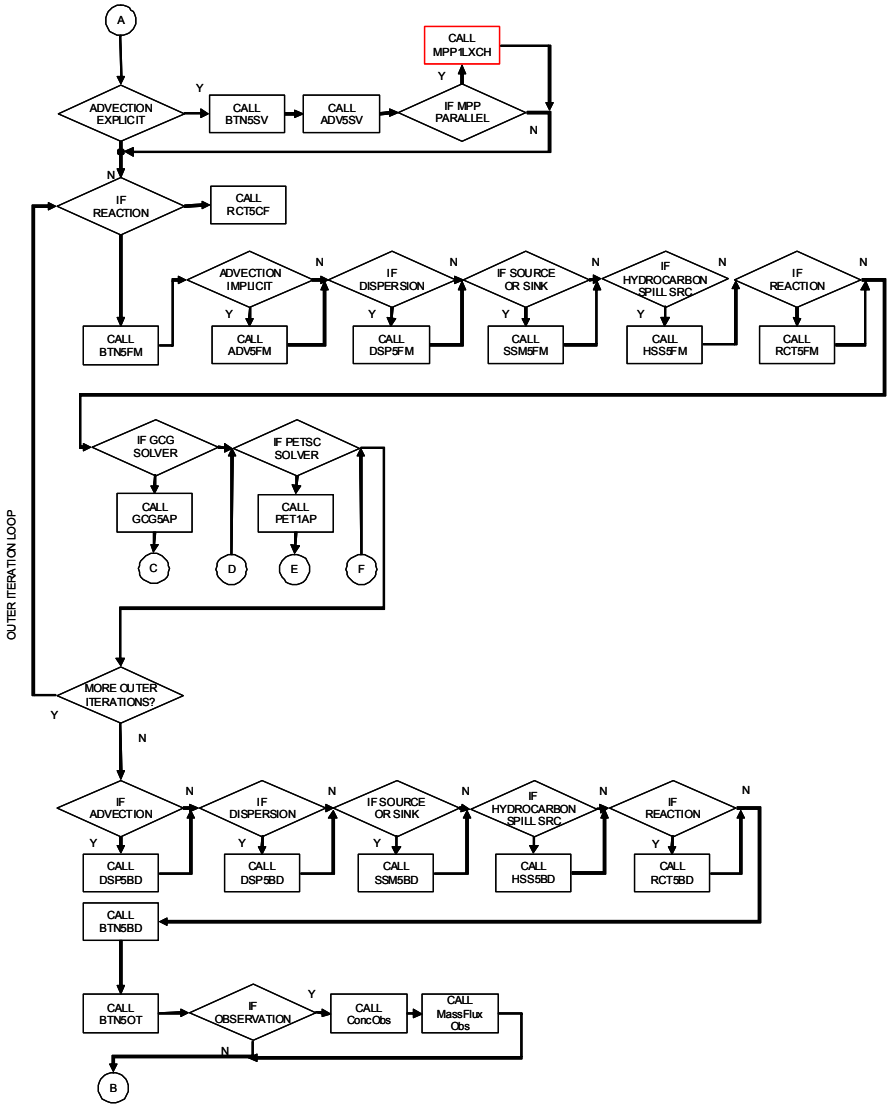

Figure 3.2: Transport time step loop flow chart (continued).

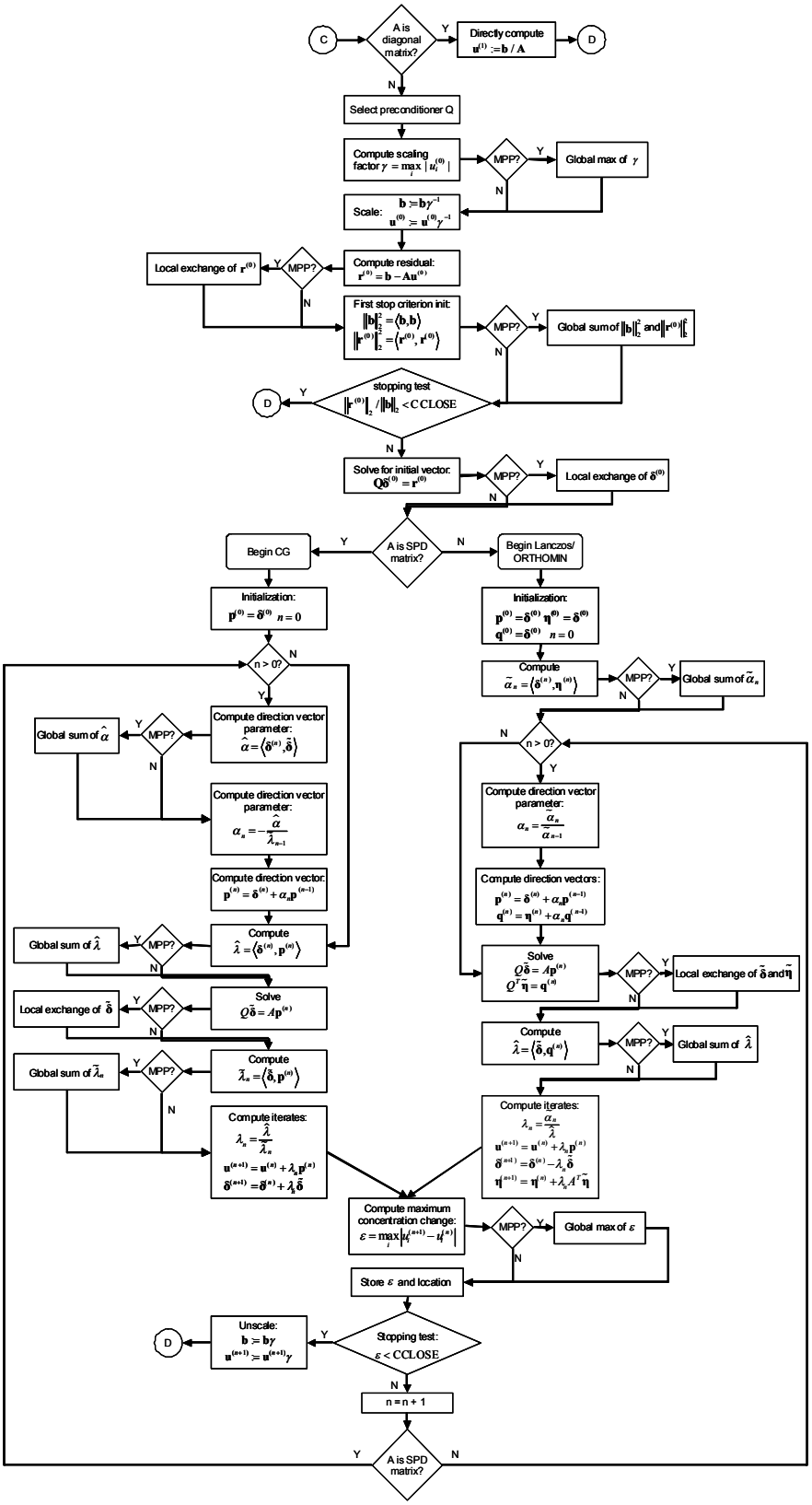

Figure 3.3: Flowchart for the parallel GCG solver (continued).

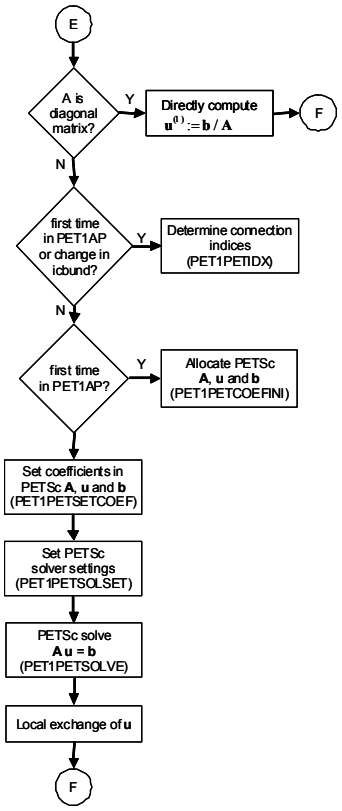

Figure 3.4: Flow chart of the PETSc solver.

4 Input instructions

Input for the Massively Parallel Processing (MPP) package is read on unit INMPP = 5, which is preset in the main program. The input file is required only if the MPP package is used in the simulation. For each simulation, are the input instructions are given by Table 4.1.

Table 4.1: Input instructions for the MPP package

Massive Parallel Processing (MPP) package			
ID	**Variable Name**	**Format**	**Explanation**
G1	MERGEOBS	free	flag indicating that Master process should merge observation points MERGEOBS = 0: each process writes his own observation file MERGEOBS = 1: master process writes observation file
G2	PARTOPT	free	flag indicating the partition method in column and row direction PARTOPT = 0: default algorithm by assuming constant load PARTOPT = 1: user defined blocks PARTOPT = 2: load balancing by user specified integer grid
(Enter G2a if PARTOPT = 1 or 2)			
G2a	NRPROC_NCOL, NRPROC_NROW	free	number of partitions in column and row direction
(Enter G2b and G2b if PARTOPT = 1)			
G2b	NRPROC_COL	free	array of length NRPROC_NCOL with the starting columns of the partitions
G2c	NRPROC_ROW	free	array of length NRPROC_NROW with the starting rows of the partitions
(Enter G2d if PARTOPT = 2)			
G2d	LOADPTR	iarray	array used for load balancing in row and column direction only entries different then 0 are used

Input to the Portable, Extensible Toolkit for Scientific Computation (PET) package is read on unit INPET = 6, which is preset in the main program. The input file is needed only if the PET package is used in the simulation. If the PET package is used, the MPP package must be used too. For each simulation are the input instructions are given by Table 4.2.

Table 4.2: Input instructions for the PET package.

Portable, Extensible Toolkit for Scientific Computation (PET) Package			
ID	**Variable Name**	**Format**	**Explanation**
H1	MXITER, ITER1, NCRS	free	maximum number of outer iterations, maximum of inner iterations; flag for treatment of dispersion cross terms NCRS = 0: dispersion cross terms lumped to right-hand side NCRS = 1: full dispersion tensor
H2	CCLOSE	free	convergence criterion: 2-norm(residual)/2-norm(right-hand side)

Note that in order to use the MPP and PET packages, you will need to have MPI installed on your system. To use PET package you will also need PETSc to have installed too.

5 Test problems

5.1 Examples

The parallel code was tested for its correctness for the supported examples in the MT3DMS v5.30 distribution (21-02-2010). The examples were taken from directory “examples\mf96mt3d\”, where MODFLOW-96 was used to compute the groundwater flow. For this, MODFLOW-96 with the link-MT3D interface (25-05-2003) was compiled with the Intel compiler under Linux. The examples were tested for 1, 2, 3, 4 and 8 processes. As a reference, the examples were run with a compiled clean version of MT3DMS v5.30.

Examples 4, 6, 8 were not tested because they all use MOC advection which is not supported. All tests gave similar results in the parallel case as for the serial case. For one specific example, we will go into more detail.

Example 1: One-dimensional transport in a uniform flow field
This test case models one-dimensional solute transport involving advection, dispersion and some simple chemical reactions in a steady-state uniform flow field. The full problem specification can be found in Sec. 7.1 of Zheng, 1999. Here, advection was solved by the parallelized TVD scheme.

Test case 1a (advection only) has no difference (exactly zero) between serial and parallel runs, both using the GCG and PETSc solver. This is easily explained by the fact that no iterative solvers are invoked when dispersion is absent. Also, the parallel advection solver (ULTIMATE) is explicit in time and, therefore, produces results across multiple partitions that are identical to the results from a serial run.

Test case 1b (advection and dispersion) has no significant difference (up to machine precision) between serial and parallel runs, both using the GCG and PETSc solver. This is achieved by setting the tolerance for the iterative solvers to machine precision (e.g., CCLOSE=1E-16). For less strict values, serial and parallel results are identical up to that tolerance, as is to be expected.

Test case 1c (advection, dispersion and sorption) shows equality between runs up to machine precision, as for test case 1b. Figure 5.1a shows the numerical solutions obtained with the PETSc solver at Tend=2000 s for serial and parallel runs. The results for any number of partitions coincide.

Test case 1d (advection, dispersion, sorption and decay) shows the same behavior. Figure 5.1b shows the numerical solutions with the parallel GCG solver at Tend=2000 s for serial and parallel runs. Again, all results coincide.

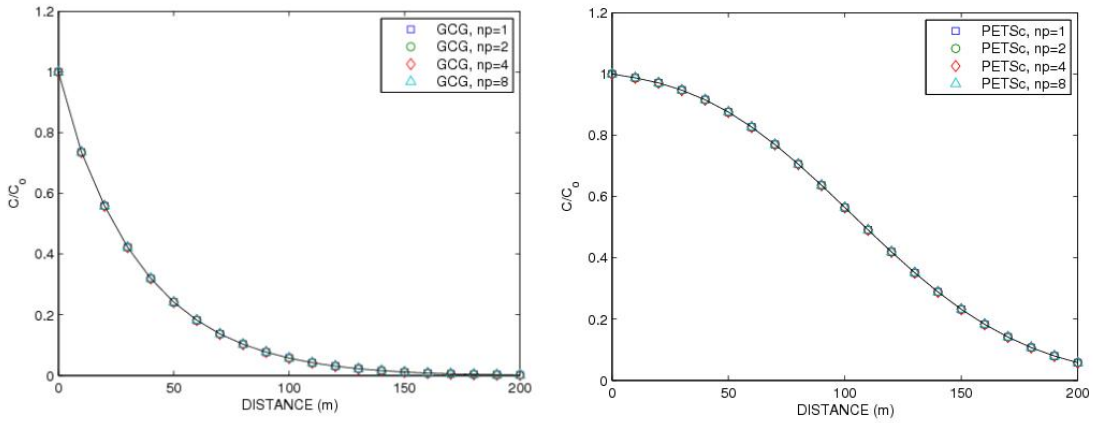

Figure 5.1: Comparison of the calculated concentrations with the analytical solutions for the one-dimensional test problem. The analytical solutions are shown in solid lines with the numerical solutions in different symbols.
a) case 1c, solved by PETSc on 1, 2, 4 and 8 processes.
b) case 1d, solved by GCG on 1, 2, 4, and 8 processes.

5.2 Large application model

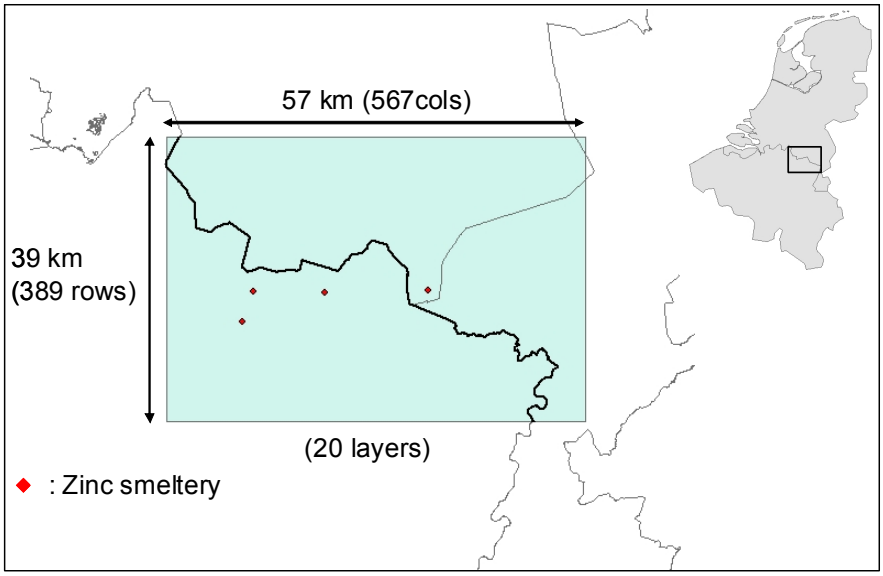

Figure 5.2: Area of interest for the BeNeKempen model and model dimensions.

In cooperation with the Dutch knowledge centers Deltares and Alterra several large contaminant transport models have been developed. The largest regional transport model that uses the maximum amount of computer resources is the recently developed so-called BeNeKempen model for the "Aktief Bodembeheer de Kempen" environmental program. This 3D transboundary model consists of ~5 million grid cells with a horizontal resolution of 100 × 100 m, covering an area of ~2200 km2, which is about 1/20 part of The Netherlands (Figure 5.2). This MT3DMS model simulates groundwater transport of cadmium and zinc originating from historical emissions of four zinc smelters in the vicinity of the Dutch-Belgian border. The typical simulation time used is 90 years.

The performance of the parallel code is benchmarked on the Dutch national compute cluster SARA-LISA[4], having specifications:

- 512 nodes each node 8 or 12 cores (total 4480 cores);
- Intel Xeon quad cores (416 nodes), Intel Xeon six cores (96 nodes);
- 2 GB internal memory per core;
- Fast Infiniband connection, 1600 MB/sec, <6 µs;

and on the Deltares in-house H4 cluster:

- 221 nodes, each node 2 or 4 cores (total 448 cores);
- AMD dual cores (220 x-nodes), AMD quad cores (1 y-nodes);
- 4 GB internal memory, 250 GB local storage;
- Slow 1 Gigabit ethernet connection.

4. *This work was sponsored by the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF) for the use of supercomputer facilities, with financial support from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organization for Scientific Research, NWO).*

The BeNeKempen model was run up to 32 processes on both machines with the MPP package selected in combination with the parallel GCG solver. The parallel GCG-solver was selected, since the PETSc solver introduces a large amount of initialization overhead when the flow is advection dominated, which is the assumption for the BeNeKempen model. True wall-clock time were measured, i.e. the time measured from start till end of simulation that is directly experienced by the user. The results are shown in Figure 5.3. A speedup of more than a factor 20 was obtained with 32 processes on the Dutch national compute cluster SARA-LISA, reducing the computing time from ~10 hours to ~30 minutes. The speedup on the H4-cluster decreases for more than 16 processors, that is probably caused by the slow ethernet interconnect. Overall, the benchmarks show that parallel MT3DMS code can significantly reduce large computing times.

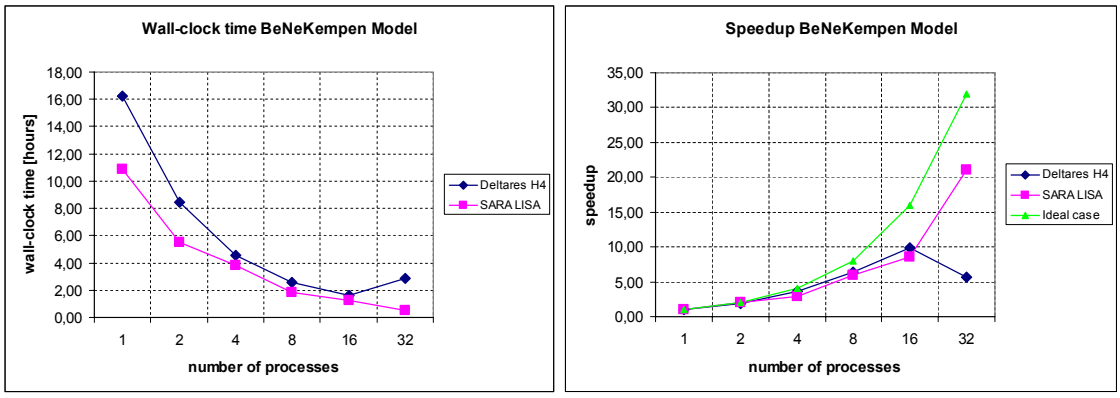

Figure 5.3: Timing results for the parallel MT3DMS code (parallel GCG) on the SARA LISA cluster and the Deltares H4 cluster. Left: measure wall-clock time; right: speed-ups.

Exactly the same results were reproduced in the parallel case as in the serial case, see Figure 5.4 for a typical observation point. Figure 5.5 shows how the default partitioning is done for the case of 16 blocks (4 × 4). Clearly, the load balance is not optimal, since the 12 outer blocks do not have the maximum load since they have a large number of inactive cells.

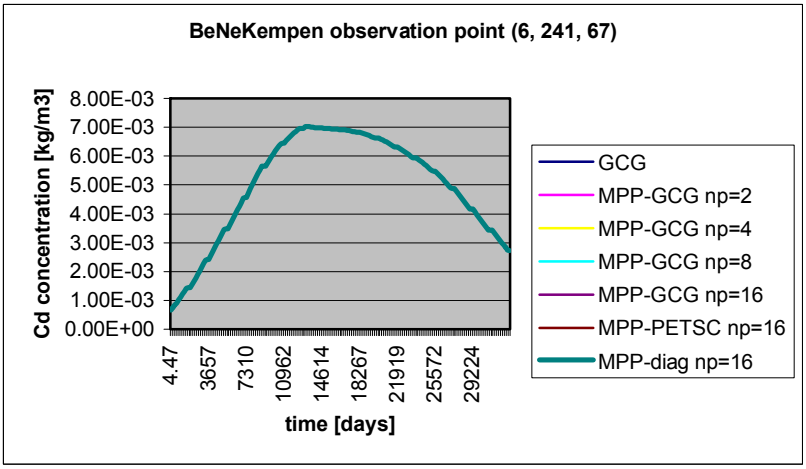

Figure 5.4: Typical observation point showing a cadmium concentration time series for 90 years. Results are shown for the GCG solver, the parallel GCG solver (2, 4, 8, 16 processors), the PETSc solver (16 processors) and the case of no implicit solver for 16 processors (diagonal check activated).

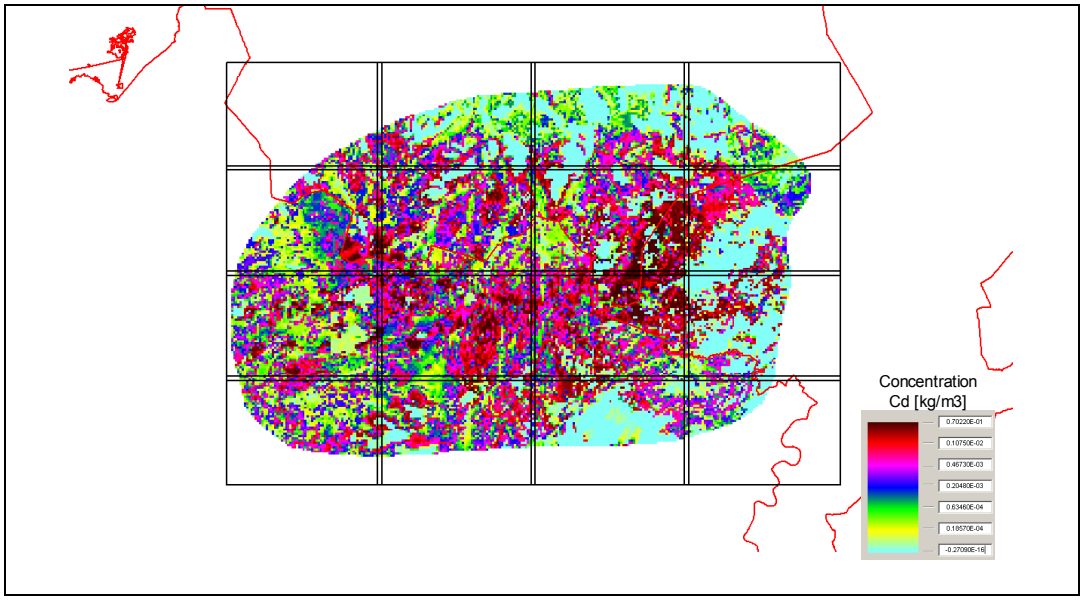

Figure 5.5: Computed cadmium concentration for layer = 1 and T = 90 years, showing the partitioning for the case of 16 processes (upper left block: P0; lower right block: P15).

References

Balay, S. *et al*. (2008). *PETSc Users Manual; revision 3.0.0*, Argonne National Laboratory.

Brakkee, E., C. Vuik and P. Wesseling (1998). 'Domain decomposition for the incompressible Navier-Stokes equations: solving problems accurately and inaccurately'. *Int. J. for Num. Meth. Fluids*, Vol. 26, pp. 1217-1237.

Grama, A., A. Gupta, G. Karypis, V. Kumar (2003). *Introduction to parallel computing*, Pearson, second edition.

Gropp, W. *et. al.* (2009). *MPICH2 User's Guide.* Mathematics and Computer Science Division, Argonne National Laboratory.

Jea, C.J., D.M. Young (1983). 'On the simplification of Generalized Conjugate-Gradient Methods for Nonsymmetrizable Systems', *Linear Algebra and its Applications*, Vol. 52/53, pp. 399-417.

Karypis. G., V. Kumar (1998). *MeTiS - A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, version 4.0.* University of Minnesota, Department of Computer Science / Army HPC Research Center.

Saad, Y. (2000). *Iterative Methods for Sparse Linear Systems*. Boston: PWS Publishing Compagny, second edition.

Zheng, C., P.P. Wang (1999). *MT3DMS, A modular three-dimensional multi-species transport model for simulation of advection, dispersion and chemical reactions of contaminants in groundwater systems; documentation and user's guide.* U.S. Army Engineer Research and Development Center Contract Report SERDP-99-1, Vicksburg, MS, 202 p.