SPE-176045-MS

A Parallel Framework for Reservoir Simulators on Distributed-memory Supercomputers

Hui Liu, Kun Wang, Zhangxin Chen, Kirk E. Jordan, Jia Luo, and Hui Deng

This paper was prepared for presentation at the SPE/IATMI Asia Pacific Oil & Gas Conference and Exhibition held in Nusa Dua, Bali, Indonesia, 20–22 October 2015.

Abstract

This paper presents our work on developing a platform for high performance reservoir simulations, which is developed to support the implementation of various reservoir simulators on distributed-memory parallel systems. This platform employs MPI (Message Passing Interface) for communications and OpenMP for shared-memory computation. It provides structured grids due to its simplicity and cell-centered data for each grid cell. The platform has a distributed matrix and vector module and a map module. The map connects the grid and linear system modules. Commonly-used Krylov subspace linear solvers are implemented, including the restarted GMRES method and the BiCGSTAB method. It also has an interface to a parallel algebraic multigrid solver, BoomerAMG from HYPRE. Parallel general-purpose preconditioners and special preconditioners for reservoir simulations are also developed. The numerical experiments show that our platform has excellent scalability and it can simulate giant models with hundreds of millions of grid cells using thousands of CPU cores.

Introduction

Nowadays, various processes have been applied by the oil and gas industry to enhance oil recovery. The simulations of these processes are becoming more and more complicated. In the meantime, geological models are more and more complex. To obtain high resolution results for these oil recovery processes, models with millions of grid cells are usually employed. Their simulations may take days or even weeks to finish one run using regular workstations and personal computers. The long simulation time could be a problem to reservoir engineers that design new production processes, since tens of simulations may be required to find optimal solutions. Fast computational methods and parallel reservoir simulators should be investigated.

Reservoir simulations have been studied for decades and various models and methods have been developed by researchers. Coats studied black oil, compositional and thermal models, and he also investigated grid effects, linear solvers and stability issues [1, 6, 7, 16, 35, 19]. Kaarstad et al. [12] implemented a reservoir simulator for a two-dimensional two-phase oil-water model, which could solve problems with up to one million grid cells. Rutledge et al. [9] developed a compositional simulator on massive SIMD computers using the IMPES (implicit pressure-explicit saturation)

method. Killough et al. [8] developed a parallel compositional simulator for distributed-memory parallel systems. Shiralkar et al. [10] developed a parallel production qualified simulator, which could run effectively on a variety of computing platforms. Killough et al. [13] also used the locally refined grids in their parallel simulator. Parashar et al. implemented a parallel simulator that handles multiple fault blocks with multiple physics [15]. Dogru et al. [14, 17] developed a parallel simulator, which was capable of simulating one billion grid cells. Chen et al. also developed parallel simulators for distributed-memory parallel systems, which showed excellent scalability on IBM Blue Gene/Q [5]. Zhang et al. developed a parallel platform for adaptive finite element and adaptive finite volume calculations, which was also applied to reservoir simulations [18, 24]. For many reservoir simulations, most of the simulation time is spent on the solution of linear systems resulted from Newton methods and it is well-known that the core of speeding up linear solvers is to develop efficient preconditioners, especially physics-based preconditioners. Many preconditioners have been proposed and they are applied to reservoir simulations, including point- wise and block-wise incomplete factorization (ILU) methods for general linear systems [26, 27, 41], domain decomposition methods [42, 41], constrained pressure residual (CPR) methods for the black oil model, compositional model and extended black oil models [28, 29, 30], multi-stage methods [32] for black oil simulations, and multiple level preconditioners [69, 70] and fast auxiliary space preconditioners (FASP) for black oil and polymer simulations [33, 37]. Chen et al. studied parallel reservoir simulations and developed a family of CPR-like preconditioners for black oil simulations and compositional simulations [2]. High performance GPUs are also applied to accelerate reservoir simulations. Chen et al. developed GPU-based linear solvers and preconditioners to speed the Krylov subspace linear solvers, algebraic multigrid solvers and preconditioners [51, 54, 30, 36, 11, 34, 44, 56]. These techniques have been applied to black oil simulations [54, 30]. Klie and Saad implemented their GPU-based linear solvers and applied them to reservoir simulation [43, 45]. Natoli et al. developed a GPU-based parallel algebraic multigrid solver and a GPU-based reservoir simulator, which was much faster than existing CPU-based simulators and could handle black oil models with tens of millions of grid cells [20, 21, 22, 23].

In this paper, we present our work on developing a parallel platform that is designed for large-scale reservoir simulations on distributed-memory parallel systems. This platform is written by C, MPI (Message Passing Interface) and OpenMP. MPI is employed to handle communications among computation nodes and OpenMP is employed to do shared-memory paral- lelization on a computation node. For now, it provides grid management, data management, linear solvers, preconditioners, distributed matrices and vectors, visualization, parallel input and output, key words parsing and well modeling modules. This platform supports structured grids, the finite difference methods and the finite volume methods. The dynamic load balancing module for grid partitioning is crucial for parallel computing [64, 67, 47, 63, 48]. In our platform, it is completed by ParMETIS [67], Zoltan [63], and the Hilbert space-filling curve (HSFC) method [64]. The ParMETIS is a graph partitioning package using topological information of a grid. In the partitioning process, a dual graph of a grid is generated, where each grid cell presents a vertex of the dual graph and an edge exists in the dual graph if two cells are neighbors. Zoltan is a geometry partitioning package using geometric information, such as coordinates, which is developed by Sandia National Laboratories. The HSFC partitioning method, a geometric partitioning method, is an in-house partitioning method. The HSFC method serves as the default partitioner. Commonly used Krylov subspace solvers and AMG solvers are implemented, including the restarted GMRES solver, BiCGSTAB solver [25], and classic AMG solvers [61, 62]. General preconditioners, including ILU(k), ILUT, domain decomposition [42] and AMG [61, 62], and special preconditioners, including the classical CPR [28, 29, 41] preconditioner and other CPR-like preconditioners, are implemented. Based on the platform, a black oil simulator

and a thermal simulator have been developed. Numerical experiments show that our platform is capable of calculating problems with hundreds of millions of grid cells and it has excellent scalability on distributed-memory parallel computers.

Grid Management

For now, only regular structured hexahedral grids are supported, due to the reason that these grids have simple geometry and they are easy to generate. Each cell of a grid is a hexahedron as shown in Figure 1, which has six faces and twelve edges. A hexahedron can be described by its edge lengths, h_x, h_y and h_z. The structured grids have simple topology and a two-dimensional cross section of a hexahedral grid is shown in Figure 2, which is also an example of a structured grid for a two-dimensional domain. For a two-dimensional structured grid, each interior grid cell has four neighbors and each boundary cell may have one or two neighbors. A three-dimensional structured grid has similar topology, where each interior cell has six neighbors and each boundary cell has three, four or five neighbors, depending on its location. A simple reservoir can be described as $\Omega = [x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$. If the domain Ω in the x, y and z directions are divided into n_x, n_y and n_z intervals, then the grid has $N_g = n_x \times n_y \times n_z$ cells. The grid can be uniform or non-uniform. If the grid is uniform, each cell is the same as others. These structured grids support finite difference methods, finite volume methods and finite element methods, and they have been widely applied by most commercial reservoir simulators.

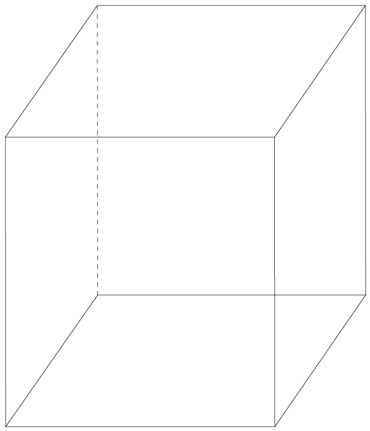

Figure 1—A hexahedron

Figure 2—Structured grid

In the platform, geometric information of each cell is stored, including the coordinates of each vertex, its centroid coordinates, the area of each face, the boundary conditions of each face and its volume. The area of each face and the volume of each cell can also be calculated using stored coordinates. Other geometric information can be calculated by stored information, such as an outer normal. Each cell has a unique global index. The default index of cell (*i, j, k*) is calculated as

$$idx(i,j,k) = n_x * n_y * k + n_x * j + i, \tag{1}$$

which is numbered from the bottom layer of a reservoir to the top layer of the reservoir. Here the *i, j* and *k* are the integer coordinates of the cell in the *x, y* and *z* directions, respectively. This numbering style is also noted as natural numbering.

Another numbering style that is used by most reservoir simulators is

$$idx(i,j,k) = n_x * n_y * (n_z - k) + n_x * j + i, \tag{2}$$

which is numbered from the top layer to the bottom layer.

Figure 3 presents our basic data types and MPI-related data types used by the platform, which is generated automatically by project management tools, such as autoconf and m4. They can be redefined by configure options when being compiled. The summary of data structure for a cell, which is defined as CELL, is shown by Figure 4. In this data structure, information for tracing neighbors, boundary type of each face and cell type are also included. The data structure for grids, GRID, is presented in Figure 5. It stores the coordinates of each vertex, vertex indices, cell indices, distribution of cells on each MPI task, a mapping between the global index of a cell and its local index, well data and MPI data. Other types of grids may be developed in the future, such as unstructured grids and local refinement of grids.

```
typedef double                  FLOAT;

#if USE_LONG_LONG
typedef signed long long int    INT;
typedef unsigned long long int  UINT;
#elif USE_LONG
typedef signed long int         INT;
typedef unsigned long int       UINT;
#else
typedef signed int              INT;
typedef unsigned int            UINT;
#endif

typedef signed short            SHORT;
typedef unsigned short          USHORT;
typedef char                    CHAR;
typedef FLOAT                   COORD[3];      /* space coordinates */
#undef  TRUE
#define TRUE                    (1)
#undef  FALSE
#define FALSE                   (0)
typedef int                     BOOLEAN;

/* MPI type */
#define PRSI_MPI_FLOAT    MPI_DOUBLE

#if USE_LONG_LONG
#define PRSI_MPI_INT      MPI_LONG_LONG
#define PRSI_MPI_UINT     MPI_UNSIGNED_LONG_LONG
#elif USE_LONG
#define PRSI_MPI_INT      MPI_LONG
#define PRSI_MPI_UINT     MPI_UNSIGNED_LONG
#else
#define PRSI_MPI_INT      MPI_INT
#define PRSI_MPI_UINT     MPI_UNSIGNED
#endif

#define PRSI_MPI_SHORT    MPI_SHORT
#define PRSI_MPI_CHAR     MPI_CHAR
#define PRSI_MPI_BOOLEAN  MPI_INT

/* CSR format */
typedef struct mat_csr_t_
{
    INT   num_rows;
    INT   num_cols;
    INT   num_nonzeros;
    INT   *Ap;
    INT   *Aj;
    FLOAT *Ax;
} mat_csr_t;
```

Figure 3—Basic data types

```
typedef struct CELL_
{
    COORD      ctrd;                    /* centroid coordinate */
    FLOAT      area[6];                 /* area of faces */
    FLOAT      vol;                     /* volume */

    void       *nb[6];                  /* neighbours */
    INT        vert[8];                 /* local index of vertices */
    INT        index;                   /* local index */
    INT        idx[3];                  /* index in x, y and z direction */
    INT        regn;                    /* region mark */

    USHORT     bdry_type[6];            /* boundary type */
    USHORT     type;                    /* cell type */

} CELL;
```

Figure 4—Data structure of CELL

```
typedef struct GRID_
{
    COORD           *vert;              /* coordinates of vertices */
    CELL            *cell;
    INT             *num_cells;         /* number of cells in each process */
    RNEIGH          *rnghr;             /* remote neighbours */
    USHORT          *type_vert;         /* vert types */
    INT             *L2Gmap_vert;       /* Local to global map of vertices */
    INT             *L2Gmap_cell;       /* Local to global map of cell indices */

    FLOAT           lif;                /* Load imbalance factor */
    INT             nregns;             /* number of marks, from 0 */
    INT             nverts;             /* number of vertices in the submesh */
    INT             nfaces;             /* nuber of faces */
    INT             ncells;             /* number of cell indices in the submesh */
    INT             nrngbr;             /* number of remote neighbours */

    INT             nverts_owned;       /* owned by current submesh */
    INT             nfaces_owned;       /* owned by current submesh */
    INT             nfaces_remote;      /* equals to number of remote neighbours */

    INT             nverts_global;      /* number of vertices in the global mesh */
    INT             nfaces_global;      /* number of vertices in the global mesh */
    INT             ncells_global;      /* number of cells in the global mesh */

    FLOAT           bbox[3][2];         /* bounding box */
    INT             ncx, ncy, ncz;      /* grid size in x, y, z directions */
    FLOAT           *vx, *vy, *vz;      /* partition of x, y, and z directions.
                                         * or coordinate of vertex in each direction */

    BOOLEAN         uniform;            /* uniform in each direction or not */
    BOOLEAN         zdownward;          /* z direction heads downward ordering */

    /* Well, process (nprocs - 1) owns all wells  */
    WELL            **well;
    WELL_CINFO      *well_cinfo;
    CELL            **_perf_cell;       /* pointer to cell which has perferation */
    INT             nperfs;             /* number of perferations */
    INT             nperfs_global;
    INT             nwells_global;      /* number of wells */
    BOOLEAN         well_assembled;
    BOOLEAN         destroy_well;       /* if grid or simulator destroy wells */

    MPI_Comm        comm;
    int             rank;
    int             nprocs;

    /* grid */
    int             id;
    GRID_TYPE       type;
    BOOLEAN         assembled;

} GRID;
```

Figure 5—Data structure of GRID

Grid Partitioning

Here we assume that N_p MPI tasks exist. Let $\Omega = [x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$ be the reservoir domain, and the domain has been partitioned into n_x, n_y and n_z intervals in the x, y and z directions. Let $\mathbb{G}$ be the structured grid, which has $N_g = n_x \times n_y \times n_z$ cells. We also use $\mathbb{G}$ to represent the union of all cells,

$$\mathbb{G} = \{B_1, B_2, \cdots, B_{N_g}\}, \tag{3}$$

where B_i, is the i-th cell of $\mathbb{G}$. The grid $\mathbb{G}$ is distributed in N_p MPI processors, and each processor has a subset of $\mathbb{G}$. Let $\mathbb{G}_i$ be the sub-grid owned by the i-th processor, which satisfies the following conditions:

$$\begin{cases} \mathbb{G}_i \neq \emptyset \ (i = 1, \cdots, N_p) \\ \mathbb{G}_i \cap \mathbb{G}_j = \emptyset \ (i \neq j) \\ \cup \mathbb{G}_i = \mathbb{G} \ (i = 1, \cdots, N_p). \end{cases} \tag{4}$$

For any cell B_i, it belongs to some sub-grid, whose neighboring cells may belong to different sub-grids.

In reservoir simulations, each cell has similar calculations. Therefore, we assume each cell has the same amount of calculations. The workload of each MPI task can be modeled by the size of its sub-grid, $|\mathbb{G}_i|$, or simply a number of grid cells in the sub-grid. When discretizing the reservoir models, information from neighboring cells is always required, if a neighbor belongs to another MPI task, and then communications are necessary.

Now let us model the communication volume for each MPI task, which can be modeled by a dual graph. Here we assume that only information from neighboring cells is required. We see a cell as a vertex of a graph, and if two cells are neighbors, there exists an edge between these two cells. The dual graph of a grid can be constructed. Figure 6 is the dual graph of a mesh in Figure 2. The dual graphs for a three-dimensional structural grid and unstructured grid can be constructed the same way. The dual graph can be presented by a sparse matrix, such as a CSR (Compressed Sparse Row) matrix. For any interior cell, its communication pattern can be shown in Figure 7, where a black spot represents an interior cell, the blue spots are the neighbors in the same sub-grid and the red spots are the neighbors in other sub-grids. If the neighbors are in the same sub-grid, then information required by the cell can be accessed directly, which requires no communication. Therefore, this kind of neighbors are called local neighbors. If the neighbors are in other sub-grids, communications are required to obtain information. These neighbors are called remote neighbors. Boundary cells have similar communication patterns. We can see that the communication volume for any sub-grid can be modeled by the total number of remote neighbors it has. The communication pattern can be modeled by the distribution of remote neighbors.

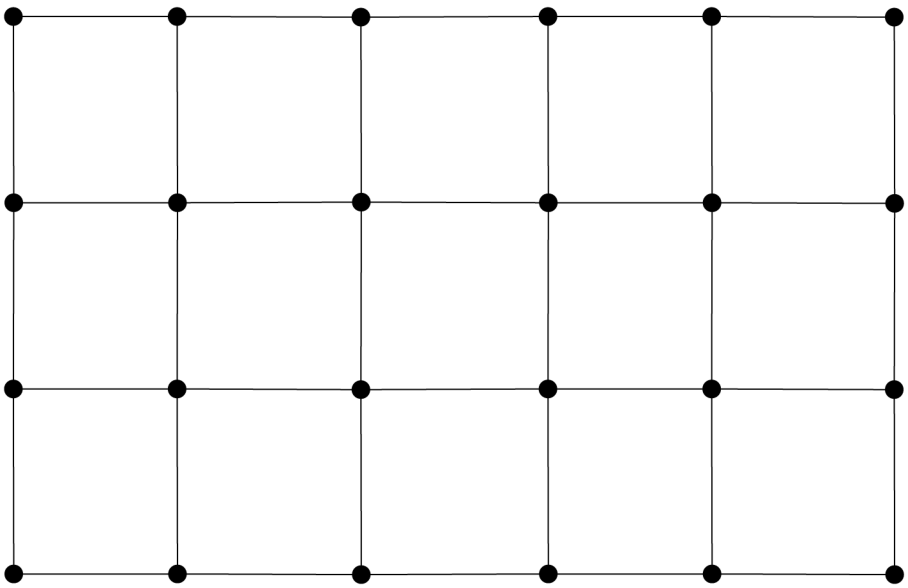

Figure 6—Dual graph of two-dimensional mesh in Figure 2

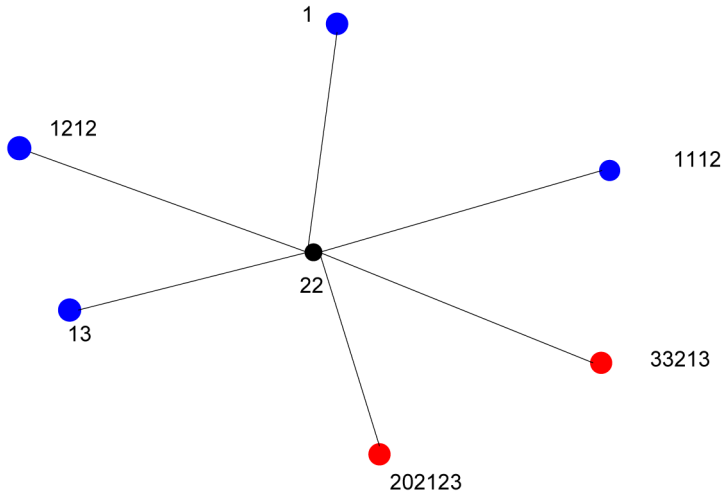

Figure 7—Communication pattern of interior cell (black spot)

Topological Methods

The goal of grid partitioning is that each processor has equal workload and the communication is minimized. The graph partitioning package ParMETIS is employed to partition the dual graph of a grid. By passing the dual graph using a CSR matrix, partition results can be calculated, whose communications are minimized approximately.

Hilbert Space-filling Curve Method The geometry information based methods are also efficient grid partitioning methods, including a recursive coordinate bisection method, a recursive inertial bisection method and the space-filling curve method [64]. The assumption of geometry information based method is that two cells that are near each other have large possibility to communicate with each other. This is true for grid based computational methods, such as finite element methods, finite volume methods and finite difference methods. The Zoltan package provides most of these methods. We implement the Hilbert space-filling curve method as a built-in module, which serves as the default partitioning method. The reasons that the Hilbert space-filling curve is applied are that a Hilbert space-filling curve has good spatial locality, the Hilbert space-filling curve method is fast and the partitioning quality is also good. Hilbert space-filling curves were designed by Hilbert, which define maps between $[0,1]^n$ and $[0,1]$. Examples of two-dimensional Hilbert space-filling curves are shown in Figure 8, where curves of levels 1, 2 and 3 are presented.

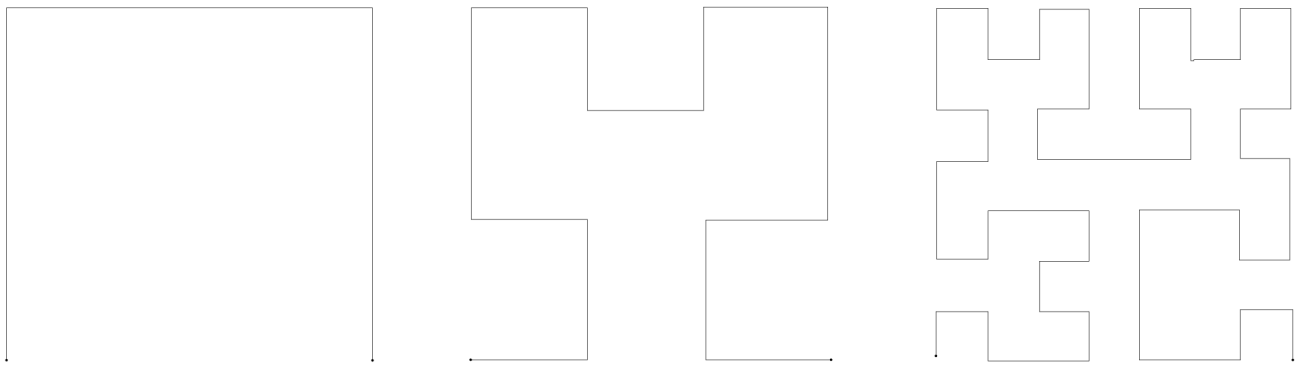

Figure 8—Hilbert space-filling curves, two dimension, level 1, 2 and 3

The algorithm of the Hilbert space-filling curve method for grid partitioning is demonstrated in Algorithm 1, which has three steps. The first step is to map a computational domain Q to a subset of $(0,1)^3$,

which can be obtained by a linear mapping. Then, for any cell, its new centroid coordinates belong to $(0,1)^3$. The Hilbert method defines a map from $[0,1]^3$ to [0,1]. We can employ the map to calculate the values of the new centroid coordinates in (0,1). The third step is to partition (0,1) into N_p sub-intervals such that each sub-interval has the same amount of cells (or workload).

Algorithm 1—Hilbert space-filling curve method

1: Map a computational domain Ω to a subset of $(0,1)^3$.
2: For any cell, calculate its mapping that belongs to $(0,1)$ using the Hilbert method.
3: Partition the interval $(0,1)$ into N_p sub-intervals and each sub-interval has the same amount of cells.

Hilbert Mapping

Let n ($n \geq 2$) be the dimension of a Hilbert curve and m be the level of the Hilbert curve. In this paper, three-dimensional Hilbert curves are applied. Let D_m be the coordinate set of the m-th level Hilbert curve, which is defined as $D_m = \{(x_n, \ldots, x_2, x_1) | 0 \leq x_i < 2^m, 1 \leq i \leq n\}$. $(x_n, \ldots, x_1)$ $(\in D_m)$ is a coordinate of the Hilbert curve, and x_i $(1 \leq i \leq n)$ is called the i-th component of the coordinate. Logical operation Λ for two coordinates is defined as

$$(x_n, \cdots, x_2, x_1) \wedge (y_n, \cdots, y_2, y_1) = (x_n \wedge y_n, \cdots, x_2 \wedge y_2, x_1 \wedge y_1), \tag{5}$$

where Λ is the regular exclusive or operation (xor).

Let $(a_1 a_2 \ldots a_k)_d$ represent a number system, where $0 \leq a_i < \mathrm{d}$ $(1 \leq i \leq k)$, and k can be any positive integer. The number is a binary number for $d = 2$, and a decimal number for $d = 10$. The and operator is denoted by &. Let us define

$$p_n^i(a_1, a_2, \cdots, a_n) = (\sum_{j=1}^{i} a_i) \; mod \; 2 = (\sum_{j=1}^{i} a_i) \; \% \; 2, \tag{6}$$

where a_i equals 0 or 1 $(1 \leq \mathrm{i} \leq n)$. The $p_n^i(a_1, a_2, \cdots, a_n)$ equals 1 or 0. With the help of p_i, the function f_n is defined as

$$f_n(a_1, a_2, \cdots, a_n) = (b_1 b_2 \cdots b_n)_2 = j, b_1 = a_1, b_i = \begin{cases} a_i \; if \; p_n^{i-1} = 0 \\ 1 - a_i \; if \; p_n^{i-1} = 1 \end{cases}, \tag{7}$$

where a_i equals 0 or 1 and j is a decimal number. The function f_n maps a vector $(a_1, a_2, \ldots, a_n)$ to a decimal number j. Its inverse function b_n is defined as

$$b_n(j) = b_n((a_1 a_2 \cdots a_n)_2) = (b_1, b_2, \cdots, b_n), b_1 = a_1, b_i = \begin{cases} a_i, \; if \; a_{i-1} = 0 \\ 1 - a_i, \; if \; a_{i-1} = 1 \end{cases}, \tag{8}$$

where $j (0 \leq j < 2^n)$ is a decimal number and $j = (a_1 a_2 \ldots a_n)_2$. The function maps a decimal number (scalar) to a vector.

Let the level of Hilbert order be m. For any point x, it can be mapped to an integer vector such that $(x_n, \ldots, x_1) \in \mathrm{D_m}$. Each component x_i $(1 \leq \mathrm{i} \leq \mathrm{n})$ is written as $x_\mathrm{i} = (x^m{}_i x^{m-1}{}_i \ldots x^1{}_i)_2$. The calculated Hilbert order is stored as $(\mathrm{r_m r_{m\text{-}1}} \ldots \mathrm{r_1})_2 \mathrm{n}$, which will be mapped to [0,1] in the end. The mapping is described in Algorithm 2. We assume GH^0 and GH^1 are known, which can be calculated by algorithms in [71] and be stored.

Algorithm 2—Hilbert mapping algorithm

(1) For $x \in [0,1]^n$, map x to domain D_m by multiplying 2^m. Let $k = m$.
(2) If $k = 0$, terminate the procedure. Or we have $r_k = f_n(x_n^k, x_{n-1}^k, \cdots, x_1^k)$.
(3) For each integer $i(1 \le i \le n)$, if x_i^k equals to 1, then $x_i = x_i - 2^{k-1}$.
(4) For each integer $i(1 \le i \le n)$, if the i-th component of $G_n^{r_k,1}$ is 1, then $x_i = 2^{k-1} - 1 - x_i$.
(5) If $G_n^{r_k,0}$ has two components equal to 1 in i-th and j-th position, then swap x_i and x_j.
(6) $k = k - 1$, goto (1).
(7) $h = \frac{(r_m r_{m-1} \cdots r_1)_{2^n}}{2^{n^m}}$ $(h \in [0,1])$.

Partitioning Quality

The quality of a partition has several measurements. The workload is measured by the load imbalance factor. Since we assume each cell has equal calculations, the workload of each processor can be approximated by the number of cells it owns. The load imbalance factor f_p is defined as the following formula:

$$f_p = \frac{N_p \times \max_{1 \le i \le N_p} |\mathbb{G}_i|}{\sum_{i=1}^{N_p} |\mathbb{G}_i|}. \tag{9}$$

The load imbalance factor f_p is always greater than or equal to one. The ideal case is that f_p equals one.

Let f_i be the number of faces in the sub-grid $\mathbb{G}_i|$ and b_i be the number of remote faces which are shared by a remote cell. b_i is proportional to the communication volume of the i-th processor involved. The maximum local surface index is defined to model the maximal communications that one processor involves,

$$r_M = \max_{0 \le i < n} \frac{b_i}{f_i}. \tag{10}$$

The global surface index is defined to model the overall communications of all processors,

$$r_G = \frac{\sum_{i=1}^{N_p} b_i}{\sum_{i=1}^{N_p} f_i - \sum_{i=1}^{N_p} b_i}. \tag{11}$$

Usually the partitioning quality of a grid from graph partitioning tools is better than that from geometric methods, such as the Hilbert space-filling curve method. However, geometric methods are much faster than graph methods.

Data Management

The cell-centered data is supported, which is natural to reservoir simulation, since each cell represents a block of a real oil and gas field and we can attach properties to the cell, such as porosity, permeability, pressure, temperature, density and viscosity. This setting is also friendly to the design of parallel linear solvers and preconditioners. Its data structure, DOF, is shown in Figure 9. Two types are provided, which support cell properties and well properties.

```
typedef struct DOF_TYPE_
{
    INT           np_cell;        /* number of DOFs per cell */
    INT           np_well;        /* number of DOFs per well */

} DOF_TYPE;

typedef struct DOF_
{
    char          *name;        /* name of DOF */
    GRID          *g;           /* the mesh */
    DOF_TYPE      *type;        /* type of DOF */
    FLOAT         *data;
    INT           *idata;
    INT           count_cell;   /* data count per cell */
    INT           count_well;   /* data count per well */
    INT           count_perf;   /* data count per perferation */
    INT           dim;

    BOOLEAN       assembled;

} DOF;
```

Figure 9—Data structure of DOF

In reservoir simulations, sometimes grid cells have different properties, such as oil and water saturations, porosity and permeability. For sequential reservoir simulators, the input and output modules (reading data from a data file and writing data to a data file) are trivial, where simple writing and reading functions from operating systems or C language are enough. However, for parallel computing, each MPI task has part of the whole grid, and each processor reads and writes the file simultaneously to setup the models, which may introduce conflicts when more than one processor exist. The platform provides parallel input and output modules using MPI-IO, which supports integer and floating-point numbers. The initialization of reservoir models and the restart of reservoir simulations are implemented using these parallel input and output functions.

Distributed Matrices and Vectors

A linear system, $Ax = b$ ($A \in R^{N \times N}$), is assembled in each Newton iteration. In the platform, each matrix and vector are distributed among all MPI tasks. Each row of the distributed matrix has a unique global row index, which ranges from 0 to N — 1 and is numbered from the 0-th MPI task to the (N_p — 1)-th MPI task consecutively. Each row also has a local index on each MPI task. The global indices of a vector is numbered the same way.

A map, MAP, is defined to store communication information among cell data, matrices and vectors. It includes DOF information, off-process entries, communication pattern, locations of data required by other MPI tasks and locations of data received from other MPI tasks. A distributed floating-point vector is defined in Figure 10, which has buffer that holds data entries, the number of local entries, the number of total entries and reference to mapping information.

```
typedef struct VEC_
{
    MAP      *map;
    FLOAT    *data;
    INT      nlocal;      /* entries belong to current proc */
    INT      localsize;   /* total entries */
    INT      id;

    BOOLEAN  assembled;

} VEC;
```

Figure 10—Data structure of VEC

The data structure of a distributed matrix is more complex than a vector, which requires entries for each row and some other additional information. It is represented in Figure 11. The MAT_ROW stores non-zero entries of each row and its storage format is similar to a CSR matrix, which has a value of an entry, and the global index and local index of an entry. The MAT has communication information, MPI information and additional information, such as mapping between off-process entries and global indices.

```
/* struct for a matrix row */
typedef struct MAT_ROW_
{
    FLOAT   *data;     /* data */
    INT     *cols;     /* local col indices, INT[ncols] */
    INT     *gcols;    /* global col indices, INT[ncols] */
    INT     ncols;     /* number of nonzero cols */

} MAT_ROW;

typedef struct MAT_
{
    /* data */
    MAT_ROW    *rows;

    MAP        *map;
    COMM_INFO  *cinfo;
    INT        *O2Gmap;

    INT        nlocal;       /* local entries */
    INT        localsize;    /* total local entries */
    INT        nglobal;      /* global matrix/vector size */
    INT        *part;        /* distribution information */

    int        rank;
    int        nprocs;
    MPI_Comm   comm;
    INT        id;

    int        refcount;
    BOOLEAN    assembled;

} MAT;
```

Figure 11—Data structure of MAT

Since we have communication information, matrices and vectors, the commonly used matrix-vector operations and vector operations can be implemented directly, which are listed as follows:

$$y = \alpha Ax + \beta y, \tag{12}$$

$$z = \alpha Ax + \beta y, \tag{13}$$

$$y = \alpha x + \beta y, \tag{14}$$

$$z = \alpha x + \beta y, \tag{15}$$

$$\alpha = \langle x, y \rangle, \tag{16}$$

$$\alpha = \langle x, x \rangle^{\frac{1}{2}}, \tag{17}$$

where matrix A is a distributed matrix, α and β are scalars, and x and y are vectors.

It is well-known that a proper matrix reordering technique improves the efficiency and robustness of linear solvers and preconditioners. There are many techniques that can reduce the bandwidth of a given matrix, such as the reverse Cuthill- MaKee reordering method. There are some techniques that can reduce fill-in of the ILU methods, such as the minimum degree (MD) reordering method. Proper reordering techniques are important to the development of efficient linear solvers and preconditioners. For the CPR-like preconditioners for reservoir simulations, the coupling between the pressure unknowns and other unknowns are required to weaken. The alternative block factorization (ABF) strategy [31] and the Quasi-IMPES strategy [68] are implemented. Other matrix-vector operations are also implemented, such as matrix (vector) creation and destroying, entries adding, assemble, matrix sorting, and sub-matrix extraction.

Linear Solvers and Preconditioners

For the linear system, $Ax = b$, derived from a nonlinear system, Krylov subspace solvers including the restarted GM- RES(m) solver, the BiCGSTAB solver, and algebraic multi-grid (AMG) solvers are commonly used. The Krylov subspace solvers mentioned here are suitable for arbitrary linear systems while the algebraic multi-grid solvers are efficient for positive definite linear systems. The parallel AMG solver is the BoomerAMG solver from HYPRE [61, 62].

Linear systems from reservoir simulations are nonsymmetric and not positive definite. The in-house parallel Krylov subspace linear solvers are implemented with the help of operations described above. The data structure of our solver, SOLVER, is shown in Figure 12, which includes parameters, matrices, right-hand sides, solutions, and preconditioner information. The data structure SOLVER_OEM defines a framework, which stores interfaces to solver creation, destroying, entries adding, assembling and solutions. Other solvers can be coupled with our platform right away.

```
typedef struct SOLVER_
{
    FLOAT                residual;
    FLOAT                rtol;
    FLOAT                atol;
    FLOAT                btol;
    int                  nits;
    INT                  maxit;
    INT                  restart;

    MAT                  *A;
    VEC                  *rhs;
    VEC                  *x;
    MAP                  *map;

    /* solver */
    struct SOLVER_OEM_   *oem;

    /* preconditioner */
    SOLVER_PC            pc;
    PC_TYPE              pc_type;

    int                  rank;
    int                  nprocs;
    MPI_Comm             comm;
    BOOLEAN              assembled;
    BOOLEAN              oem_created;

} SOLVER;

typedef struct SOLVER_OEM_
{
    int (*RegisterOptions)(void);
    int (*Create)(SOLVER *solver);
    int (*Destroy)(SOLVER *solver);
    int (*AddMatrixEntries)(SOLVER *solver, INT nrows, INT *rows,
            INT ncols, INT *cols, FLOAT *values);
    int (*AddRHSEntries)(SOLVER *solver, INT n, INT *ni, FLOAT *values);
    int (*Assemble)(SOLVER *solver);
    int (*Solve)(SOLVER *solver, BOOLEAN destroy);

} SOLVER_OEM;
```

Figure 12—Data structure of SOLVER

Several preconditioners are developed, including general purpose preconditioners and physics-based preconditioners for reservoir simulations only.

Restricted Additive Schwarz Method For the traditional ILU methods, the given matrix A is factorized into a lower triangular matrix L and an upper triangular matrix U; a lower triangular linear system and an upper triangular linear system are required to solve:

$$LUx = b \Longleftrightarrow Ly = b, Ux = y. \tag{18}$$

For the lower triangular system $L_y = b$, the system needs to solved row-by-row from the top row to the bottom row. For the upper triangular system $U_x = y$, the system needs to be solved row-by-row from the bottom row to the top row. The algorithms are serial and it is well-known that they have limited scalability. However, the RAS method is parallel. The restricted additive Schwarz (RAS) method [39], one of the domain decomposition preconditioners, was developed by Cai et al, which works for arbitrary non-singular square matrices.

For any given square matrix A, whose order is N, it can be represented as a directed (or undirected) graph G. In our platform, the graph is partitioned into N_p non-overlapping subsets, and each MPI task owns a subset W_i^0 . For the subset W_i^0 , a 1-overlap subset W_i^1 is obtained by including all its immediate neighbor vertices in G [39]. Repeating this process, a δ-overlap subset W_i^δ is defined. For each W_i^δ , a restriction operator R_i^δ is defined. Let N_i^δ be the dimension of W_i^δ ; then R_i^δ is an $N_i^\delta \times N$ matrix. The submatrix A_i is defined as

$$A_i = R_i^\delta A R_i^{\delta^T}, \tag{19}$$

which is an $N_i^\delta \times N_i^\delta$ matrix. A restriction operator D_i^δ for each δ-overlap subset W_i^δ to its original subset 0-overlap subset W_i^0 can be defined, which is similar to R_i^δ . Each processor solves its local linear system simultaneously:

$$A_i R_i^\delta x = R_i^\delta b, \tag{20}$$

which is solved by the ILU(k), ILUT(p, tol) and ILUC methods in our platform. Then the solution is projected to W_i^0 using the restriction operator D_i^δ , which needs no communication.

The data structure of the RAS preconditioner is shown in Figure 13. The pars stores parameters of the RAS preconditioner, such as overlap, a local solver (ILUK, ILUT and ILTC), the level of ILUK, memory control and tolerance of ILUT, and filter tolerance. The RAS_DATA has a local problem stored by the lower triangular matrix L and the upper triangular matrix U, communication information of different sub-domains (cinfo), memory buffer and prolongation (restriction) operation information. The default parameters of the RAS preconditioner is shown in Figure 14. The default local solver is ILU(0). Default parameters for ILUT(p, tol) is -1 and 1e-3. If p is -1, the factorization subroutine will determine dynamically. For the preconditioning system $Mx = b$, only an overlapped portion of b is required to exchange during solution process, and thus the RAS method has excellent scalability. It also has good convergence.

```
typedef struct RAS_PARS_
{
    INT         overlap;
    INT         iluk_level;
    INT         ilut_p;
    int         solver;
    FLOAT       ilut_tol;
    FLOAT       filter_tol;
    INT         ilutc_drop;

} RAS_PARS;

/* RAS */
typedef struct RAS_DATA_
{
    COMM_INFO           *cinfo;
    mat_csr_t           L;
    mat_csr_t           U;

    FLOAT               *frbuf;
    FLOAT               *fxbuf;
    FLOAT               *fbbuf;
    INT                 *ras_pro;     /* ras prolongation */
    INT                 num_ras_pro;

    RAS_PARS            pars;

} RAS_DATA;
```

Figure 13—Data structure of RAS preconditioner

```
static RAS_PARS ras_pars_default =
{
    /* overlap */       1,
    /* k */             0,
    /* ilut_p */        -1,
    /* solver */        ILUK,
    /* ilut_tol */      1e-3,
    /* filter tol */    1e-4,
    /* drop */          0,
};
```

Figure 14—Default parameters of RAS preconditioner

Algebraic Multigrid Methods If A is a positive-definite square matrix, the AMG methods [58, 52, 53, 55,4] have proved to be the most efficient methods for positive definite linear systems and are also scalable [61, 40]. Its structure of an algebraic multigrid solver is shown in Figure 15. A coarse grid is chosen when entering a coarser level. The Cleary-Luby-Jones-Plassman (CLJP) parallel coarsening algorithm was proposed by Cleary [46] based on the algorithms developed by Luby [50] and Jones and Plassman [49]. The standard RS coarsening algorithm has also been parallelized [61]. Falgout et al. developed a parallel coarsening algorithm, the Falgout coarsening algorithm, which has been implemented in HYPRE [61]. Yang et al. proposed HMIS and PMIS coarsening algorithms for a coarse grid selection [3]. A restriction operator R_l and an interpolation (prolongation) operator P_l are determined. In general, the restriction operator R_l is the transpose of the interpolation (prolongation) operator P_l:

$$R_l = P_l^T.$$

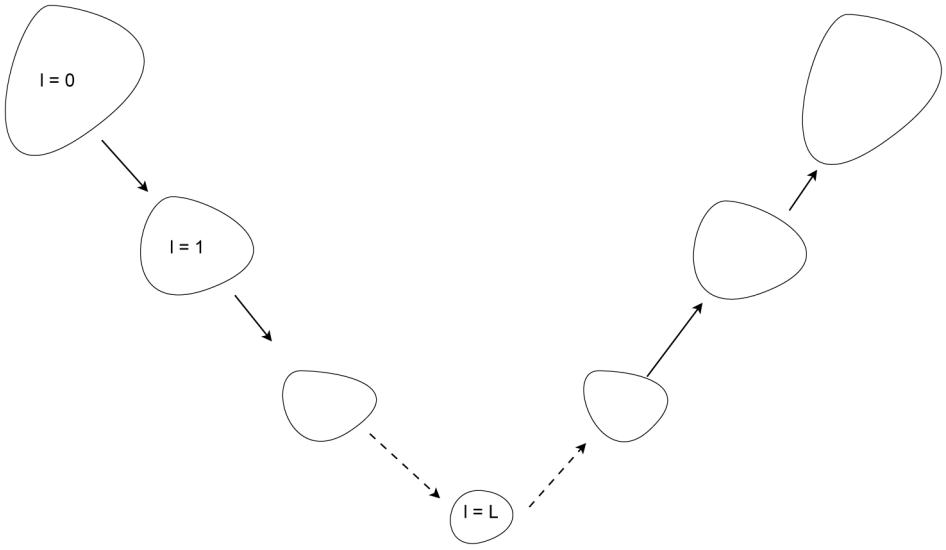

Figure 15—Structure of AMG solver.

The matrix on the coarser grid is calculated by

$$A_{l+1} = R_l A_l P_l. \tag{21}$$

We know that a high frequency error is easier to converge on a fine grid than a low frequency error, and for the AMG methods, the restriction operator, R_l, projects the error from a finer grid onto a coarser grid and converts a low frequency error to a high frequency error. The interpolation operator transfers a

solution on a coarser grid to that on a finer grid. Its setup phase for the AMG methods on each level l ($0 \leq l \leq L$) is formulated in Algorithm 3, where a coarser grid, an interpolation operator, a restriction operator, a coarser matrix and post- and pre-smoothers are constructed. By repeating the algorithm, an L-level system can be built. The solution of the AMG methods is recursive and is formulated in Algorithm 4, which shows one iteration of AMG. The AMG package we use is the BoomerAMG from HYPRE [61, 62].

Algorithm 3—AMG setup Algorithm

1: Calculate strength matrix S.
2: Choose coarsening nodes set ω_{l+1} according to strength matrix S, such that $\omega_{l+1} \subset \omega_l$.
3: Calculate prolongation operator P_l.
4: Derive restriction operator $R_l = P_l^T$.
5: Calculate coarse matrix A_{l+1}: $A_{l+1} = R_l \times A_l \times P_l$.
6: Setup pre-smoother S_l and post-smoother T_l.

Algorithm 4—AMG V-cycle Solution Algorithm: amg_solve(*l*)

Require: $b_l, x_l, A_l, R_l, P_l, S_l, T_l, 0 \leq l < L$

$b_0 = b$
if $(l < L)$ **then**
  $x_l = S_l(x_l, A_l, b_l)$
  $r = b_l - A_l x_l$
  $b_{r+1} = R_l r$
  amg_solve$(l+1)$
  $x_l = x_l + P_l x_{l+1}$
  $x_l = T_l(x_l, A_l, b_l)$
else
  $x_l = A_l^{-1} b_l$
end if
x = x_0

The data structure of the AMG method, which is used as a solver and a preconditioner depending on the settings, is shown in Figure 16. The BMAMG_PARS stores parameters of the AMG method, including the coarsening type, interpolation type, maximal levels, smoother type, and cycle type. The BMAMG-_DATA stores linear system information, such as a distribution pattern of matrices and vectors, mapping information, and related interfaces. Default parameters for the AMG method is shown in Figure 17, where a default six-level AMG method is applied. The detailed explanation of each parameter can be read from the HYPRE user manual.

```
typedef struct BMAMG_PARS_
{
    INT         maxit;
    INT         num_funcs;           /* size of the system of PDEs */
    INT         max_levels;          /* max MG levels */
    FLOAT       strength;            /* strength threshold */
    FLOAT       max_row_sum;         /* max row sum */
    FLOAT       trunc_tol;           /* trunc tol */

    int         coarsen_type;        /* default coarsening type = Falgout */
    int         cycle_type;          /* MG cycle type */
    int         relax_type;          /* relaxation type */
    int         coarsest_relax_type; /* relax type on the coarsest grid */
    int         interp_type;         /* interpolation */
    INT         num_relax;           /* number of sweep */

} BMAMG_PARS;

typedef struct BMAMG_DATA_
{
    MAP                  *map;
    INT                  ilower, iupper;
    HYPRE_IJMatrix       A;
    HYPRE_IJVector       b, x;
    BOOLEAN              assembled;
    HYPRE_Solver         hsolver;

    SolveFcn             setup;
    SolveFcn             solve;
    DestroyFcn           destroy;

    BMAMG_PARS           pars;

} BMAMG_DATA;
```

Figure 16—Data structure of AMG preconditioner

```
static BMAMG_PARS amg_pars_default =
{
    /* maxit */              1,
    /* num_funcs */          -1,
    /* max_levels */         6,
    /* strength */           0.5,
    /* max_row_sum */        0.9,
    /* trunc error */        1e-2,

    /* coarsen_type */       Falgout,
    /* cycle_type */         v-cycle,
    /* relax_type */         gs-h-forward,
    /* coarsest_relax_type */ gs-h-symmetric,
    /* interp type */        cmi,
    /* itr relax */          2,
};
```

Figure 17—Default parameters of AMG preconditioner

CPR-like Preconditioners for Isothermal Models For the black oil and composition models, the pressure unknowns dominate the overall error and their linear systems are hard to solve. The matrices from the pressure unknowns are positive definite. Many preconditioners have been developed to speed the

solution of the linear systems from these models, such as the constrained pressure residual (CPR) method. In this section, we introduce our multi-stage preconditioner for the black oil and compositional models, which is based on the classical CPR preconditioner.

When we discretize the black oil and compositional models, different primary variables (unknowns) may be chosen [59]. In this paper, we assume that the oil phase pressure p_o is always one of the unknowns. Assume that a grid is chosen, which has grid cells, and the grid cells are numbered from 1 to N_g. For each grid cell i ($1 \leq i \leq N_g$), there are several primary variables, one of which is the oil phase pressure $p_{0,i}$. The other variables are denoted by $\overrightarrow{s_i}$. Depending on the reservoir models and states, $\overrightarrow{s_i}$ may contain one, two or more variables. The well unknowns are denoted by $\overrightarrow{w}$, whose dimension equals the number of wells in the reservoir, N_w. Let us define the pressure vector p as follows:

$$p = \begin{pmatrix} p_{o,1} \\ p_{o,2} \\ \cdots \\ p_{o,N_g} \end{pmatrix}, \tag{22}$$

and the global unknown vector x as follows:

$$x = \begin{pmatrix} p_{o,1} \\ p_{o,2} \\ \cdots \\ p_{o,N_g} \\ \overrightarrow{s_1} \\ \overrightarrow{s_2} \\ \cdots \\ \overrightarrow{s_{N_g}} \\ \overrightarrow{w} \end{pmatrix}. \tag{23}$$

The global restriction operator from x to p is defined as

$$\Pi_r x = p. \tag{24}$$

The global restriction operator Π_r applies to the fluid flow variables only (not to the well variables). A prolongation operator Π_P is defined as

$$\Pi_p p = \begin{pmatrix} p_{o,1} \\ p_{o,2} \\ \cdots \\ p_{o,N_g} \\ \overrightarrow{0} \\ \overrightarrow{0} \\ \cdots \\ \overrightarrow{0} \\ \overrightarrow{0} \end{pmatrix}, \tag{25}$$

where $\Pi_p p$ has the same dimension as x.

The matrix A derived from the Newton methods for the black oil and compositional models can be written as equation (26) if a proper ordering technique is applied:

$$\widetilde{A} = \begin{pmatrix} A_{pp} & A_{ps} & A_{pw} \\ A_{sp} & A_{ss} & A_{sw} \\ A_{wp} & A_{ws} & A_{ww} \end{pmatrix}, \tag{26}$$

where the sub-matrix A_{pp} is the matrix corresponding to the pressure unknowns, the sub-matrix A_{ss} is the matrix corresponding to the other unknowns, the sub-matrix A_{ww} is the matrix corresponding to the well bottom hole pressure unknowns, and other matrices are coupled items. The matrix A_{pp} is positive definite for the black oil model. From now on, we assume the matrix A has the same structure as $\widetilde{A}$.

Let us introduce some notation for the preconditioning system $Ax = f$. If A is a positive definite matrix, then we define the notation $\mathscr{M}_g(A)^{-1}b$ to represent the solution x from the AMG methods. If it is solved by the RAS method, then we use the notation $\mathscr{R}(A)^{-1}b$ to represent solution x. The preconditioners we develop are shown from Algorithm 5 to Algorithm 8, which are noted as CPR-PF, CPR-FP, CPR-FPF and CPR-FFPF preconditioners. The algorithm uses two, three and four steps to solve the preconditioning system. Details can be read from [2].

Algorithm 5—The CPR preconditioner

1: $x = \Pi_p \mathscr{M}_g(A_{pp})^{-1}\Pi_r f.$
2: $r = f - Ax$
3: $x = x + \mathscr{R}(A)^{-1}r.$

Algorithm 6—The CPR-FP Preconditioner

1: $x = R(A)^{-1}f.$
2: $r = f - Ax$
3: $x = x + \Pi_p \mathscr{M}_g(A_{pp})^{-1}\Pi_r r.$

Algorithm 7—The CPR-FPF preconditioner

1: Initialize x. {x is 0 usually}
2: $x = \mathscr{R}(A)^{-1}f.$
3: $r = f - Ax$
4: $x = x + \Pi_p(\mathscr{M}_g(A_{pp})^{-1}(\Pi_r r)).$
5: $r = f - Ax$
6: $x = x + \mathscr{R}(A)^{-1}r.$

Algorithm 8—The CPR-FFPF Preconditioner

1: $x = \mathscr{R}(A)^{-1}f.$
2: $r = f - Ax$
3: $x = x + \mathscr{R}(A)^{-1}r.$
4: $r = f - Ax$
5: $x = x + \Pi_p \mathscr{M}_g(A_{pp})^{-1}\Pi_r r.$
6: $r = f - Ax$
7: $x = x + \mathscr{R}(A)^{-1}r.$

The data structure of the CPR preconditioners is shown in Figure 18. The term CPR_PARS stores parameters of the CPR methods, including parameters for the RAS method and parameters for AMG method. The term CPR_DATA includes two linear systems (RAS solver for the whole problem and AMG solver for the pressure problem), prolongation information of the pressure problem, and memory buffers.

```
typedef struct CPR_PARS_
{
    RAS_PARS    ras;
    BMAMG_PARS  amg;
    INT         pres_which;
    INT         pres_loc;

    INT         itr_ras_pre;
    INT         itr_ras_post;

} CPR_PARS;

typedef struct CPR_DATA_
{
    RAS_DATA        ras;
    BMAMG_DATA      amg;

    INT             *pro_pres;    /* prolongation from pressure */
    INT             num_pro_pres;
    VEC             *varbuf;      /* vector r (A), buffer */
    VEC             *vpbbuf;      /* buffer for AMG */
    VEC             *vpxbuf;      /* buffer for AMG */

    CPR_PARS        pars;

} CPR_DATA;
```

Figure 18—Data structure of CPR preconditioners

Numerical Experiments

Two parallel computers are employed to test our simulators. The first one is called Parallel from Westgrid, University of Calgary. The Parallel has 7,056 CPU cores, and there are 528 12-core standard nodes and 60 special 12-core nodes that have 3 general-purpose GPUs each. The 12 cores associated with one compute node share 24 GB of RAM. Parallel uses an InfiniBand 4X QDR (Quad Data Rate) 40 Gbit/s switched fabric, with a two to one blocking factor. The second parallel computer is Blue Gene/Q from IBM. The system, Wat2Q, is located in the IBM Thomas J. Watson Research Center, Yorktown Heights, New York. It has two racks, and each rack has two mid-planes, each of which has 16 computer nodes. Each node has 32 compute cards (64-bit PowerPC A2 processor). One compute card has 17 cores, one of which is for the operation system and the other 16 cores for computation. The IBM Blue Gene/Q system has 32,768 CPU cores for computation. The 64-bit PowerPC A2 processor cores are 4-way simultaneously multi-threaded, and run at 1.6 GHz. The IBM Blue Gene/Q has a strong network relative to compute performance, which means that the system can scale. However, the downside is that the performance of each core is really low compared with processors from Intel. In this section, each CPU core runs at most one MPI task, so the number of CPU cores is equivalent to the number of MPI tasks in the following examples.

The original SPE10 benchmark

The SPE10 benchmark and its refined models are used to test the scalability of our parallel simulators. The models have a sufficiently fine grid, which makes use of classical pseudoisation methods almost

impossible. At the fine geological model scale, the models are described on a regular Cartesian grid, whose dimensions are 1,200 × 2,200 × 170 (ft). The top 70 ft (35 layers) represents the Tarbert formation, and the bottom 100 ft (50 layers) represents Upper Ness. The fine scale cell size is 20 × 10 × 2 (ft) [66]. The grid size of the models is 60 × 220 × 85 cells (1.122×10^6 cells). It has five wells, one injection well and four production wells. The injection rate for the injector is 5,000 bbl/day and the maximal injection bottom hole pressure is 10,000 psi. The four producers are operated at 4,000 psi bottom hole pressure. The original oil-water model in this case has around 2.244 millions of unknowns.

The model is highly heterogeneous, whose permeability is ranged from 6.65e-7 Darcy to 20 Darcy, and the x-direction permeability is shown in Figure 19. The porosity of the SPE10 benchmark, shown in Figure 20, ranges from 0 to 0.5. The relative permeability of the water phase is calculated by

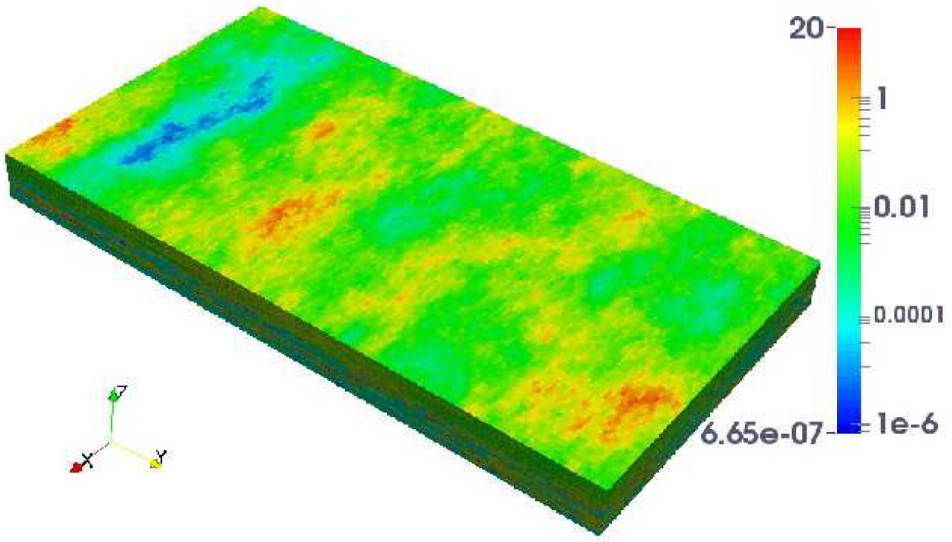

Figure 19—Permeability in X Direction of the SPE10 benchmark

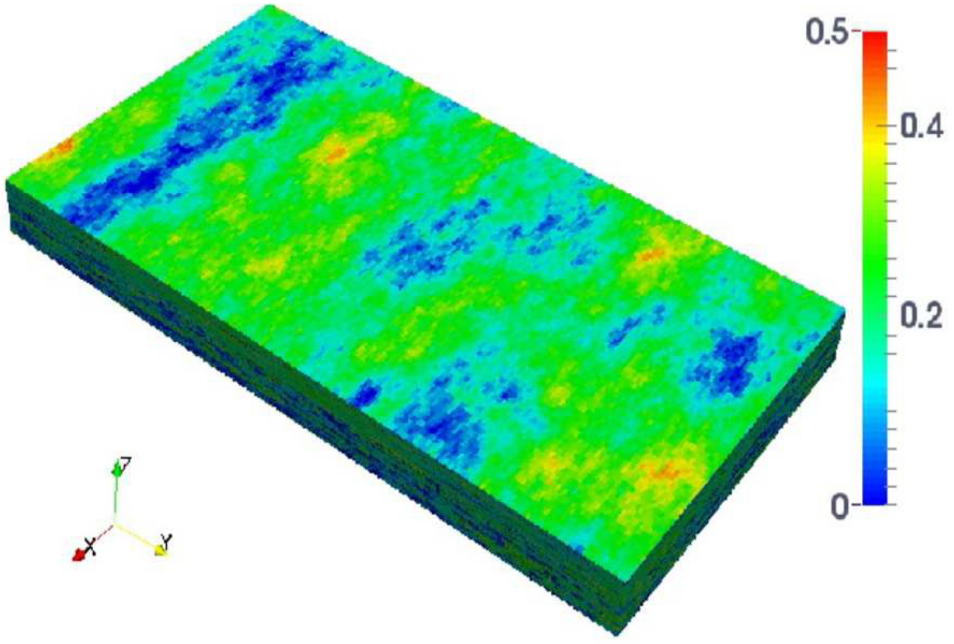

Figure 20—Porosity of the SPE10 benchmark

$$K_{rw}(s_w) = \frac{(s_w - s_{wc})^2}{(1 - s_{wc} - s_{or})^2}, \tag{27}$$

and the relative permeability of the oil phase is calculated by

$$K_{ro}(s_w) = \frac{(1 - s_{or} - s_w)^2}{(1 - s_{wc} - s_{or})^2}, \tag{28}$$

where $s_{wc} = s_{or} = 0.2$. The capillary pressure is ignored. More details can be found in [66].

When the SPE10 benchmark is used for the black oil model, fluid properties are adopted; see Tables 1 and 2. Here B_o is the oil formation volume factor, R_s is the solution gas-oil ratio for saturated oil, C_o is the compressibility of under-saturated oil and C_{vo} is the oil viscosity compressibility of under-saturated oil. The water viscosity is constant, $\mu_w = 0.3$cp.

Table 1—Oil PVT

Pressure psia	Rs MSCF/STB	Bo RB/STB	Viscosity cp	Co psia^{-1}	Cvo psia^{-1}
400	0.0165	1.012	3.5057	1.1388e-6	0.0
4000	0.1130	1.01278	2.9972	1.1388e-6	0.0
10000	0.1810	1.155	2.6675	1.1388e-6	0.0

Table 2—Gas PVT

Pressure psia	Bg SCF/STB	Viscosity cp
400	1.96	0.0140
4000	0.84	0.0160
8000	0.59	0.0175
10000	0.42	0.0195

Example 1.

This case tests the partition quality of the ParMET/S and HSFC methods. The case has a uniform grid with a dimension of $600 \times 600 \times 600$, *which has 216 millions of grid cells. The summaries are listed in Tables 3 and 4.*

Table 3—Partition quality of HSFC method, Example 1, IBM Blue Gene/Q

# procs	Time (s)	# CE (s)	MLSI	ASI	GSI	MIC
16	26.68	1,913,062	0.0071	0.0059	0.0029	8
32	13.80	2,790,851	0.0110	0.0086	0.0043	9
64	7.15	3,695,842	0.0148	0.0113	0.0056	11
128	3.72	5,464,385	0.0215	0.0167	0.0084	13
256	1.96	7,474,494	0.0284	0.0228	0.0115	14
512	1.07	9,792,151	0.0376	0.0297	0.0151	17

Table 4—Partition quality of ParMETIS, Example 1, IBM Blue Gene/Q

# procs	Time (s)	# CE (s)	MLSI	ASI	GSI	MIC
16	207.74	2,488,307	0.0111	0.0076	0.0038	10
32	101.71	3,749,825	0.0161	0.0115	0.0058	13
64	54.59	5,319,779	0.0232	0.0163	0.0082	18
128	31.24	7,063,938	0.0287	0.0215	0.0101	18
256	19.63	9,474,861	0.0377	0.0288	0.0145	20
512	14.60	12,153,852	0.0475	0.0368	0.0187	21

In Tables 3 and 4, CE, MLSI, ASI, GSI, and MIC mean the number of cutting edges, maximal local surface index, average surface index, global surface index and maximal interprocess connectivity, respectively. CE, MLSI, ASI, and GSI represent the communication volume, including the total volume, average volume and maximal volume. MIC represents startup times for group communications, which can model latency. If their values are small, then communications are good and the applications have good scalability. From Tables 3 and 4, we can see, for this example, the partition from the HSFC method has higher quality than that from ParMETIS. The tables also show that the HFSC method is much faster than ParMETIS. Their scalability is compared in Figure 21, from which we can the HSFC method has better scalability than the ParMETIS.

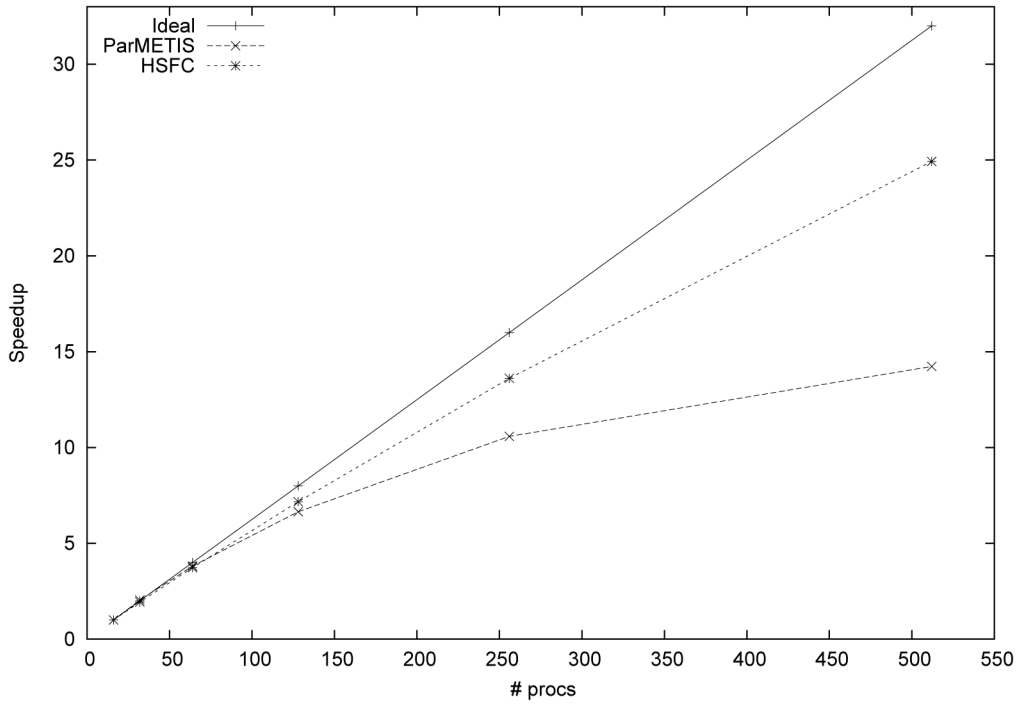

Figure 21—Scalability of partitioners, Example 1, IBM Blue Gene/Q

Example 2.

This case tests the performance of sparse matrix-vector multiplication. The case has a uniform grid with a dimension of $585 \times 585 \times 585$, *which has around 200 millions of rows. This example tests the operation* $z = \alpha Ax + \beta y$. *Its performance is shown in Table 5 and Figure 22.*

Table 5—Performance of SpMV, Example 2, IBM Blue Gene/Q

# procs	32	64	128	256	512	1024	2048
Time (s)	2.211	1.078	0.556	0.269	0.134	0.067	0.0372

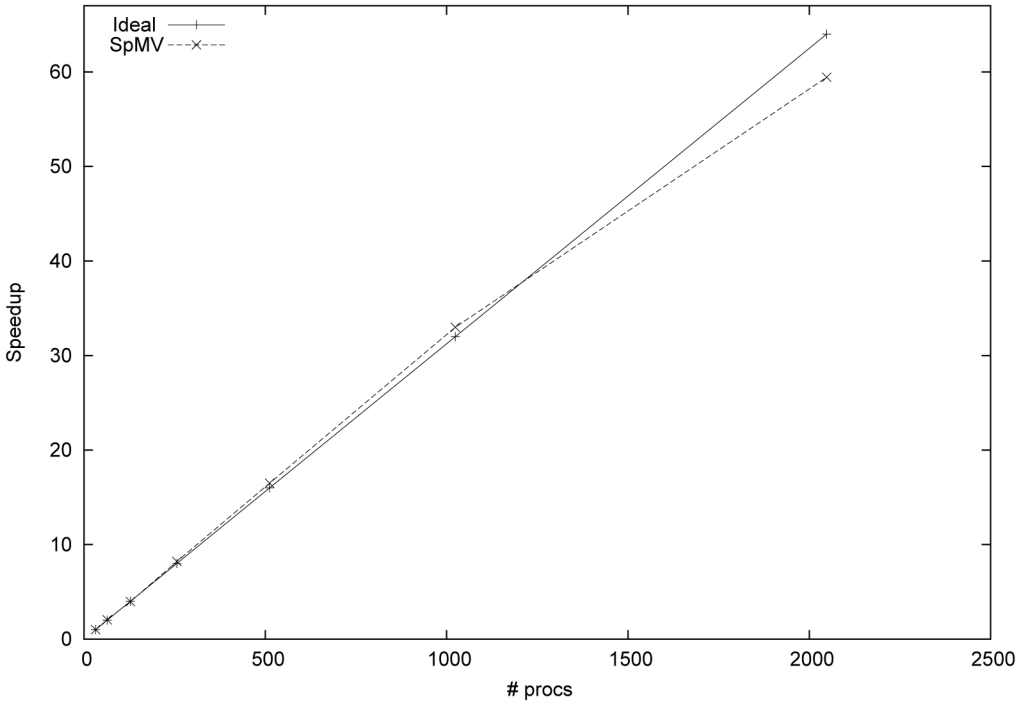

Figure 22—Scalability of SpMV, Example 2, IBM Blue Gene/Q

This example uses up to 128 compute cards, when more than 128 MPI tasks are use, multiple MPI tasks run on one card. For the case with 256 MPI tasks, 2 MPI tasks run on one compute card - up to 64 MPI tasks can be run on a compute card. From Table 22, we can see that when the number of MPI tasks is doubled, the running time of SpMV is reduced by half. The results in the table and the scalability in Figure 22 show that our SpMV has excellent scalability.

Example 3.

This case is a simple Poisson equation defined on domain $[0,1]^3$ *with a Dirichlet boundary condition. The case is run on the Parallel cluster; and up to 128 MPI tasks are used. Each computation node has only one MPI task. The case has a uniform grid with a dimension of* $500 \times 500 \times 500$, *which has 125 millions of grid cells. The restarted GMRES(50) solver and RAS preconditioner are applied. The overlap of the RAS method is one and the sub-problem of the RAS method is solved by ILUT(14, 1e-3). The stopping tolerance is 1e-8. The numerical summaries are listed in Table 6 and scalability results are shown in Figure 23.*

Table 6—Performance of Example 3 on the Parallel cluster

# procs	Griding (s)	Overall (s)	# Itr	Speedup
16	49.19	1258.55	240	16
32	24.18	662.10	240	30.41
64	11.84	338.68	240	59.45
128	5.45	166.54	240	120.91

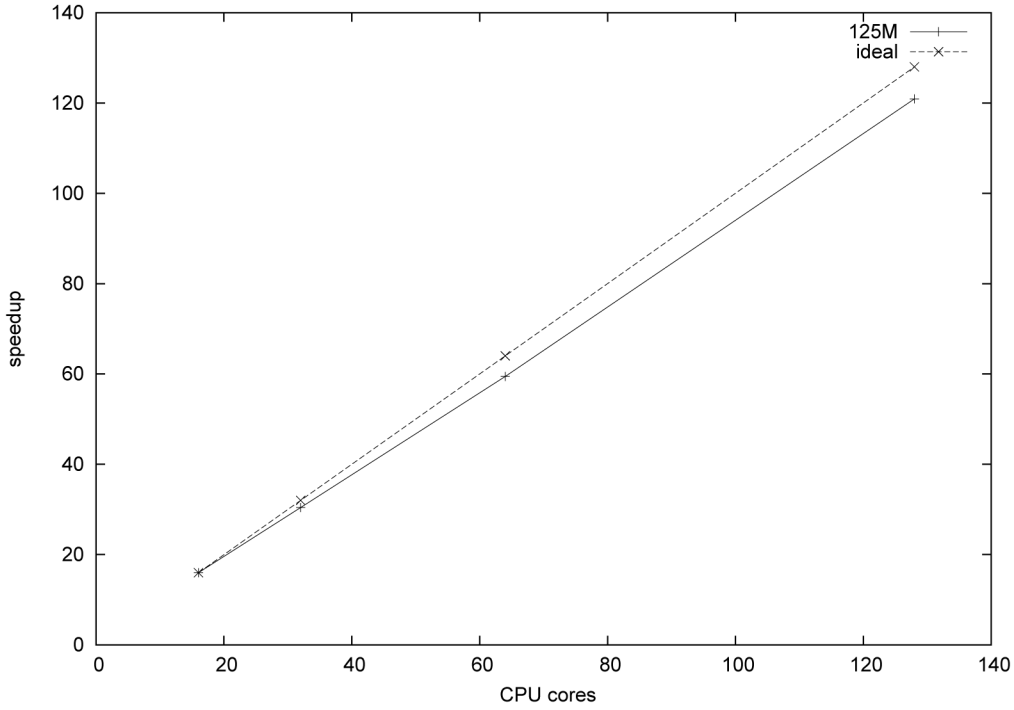

Figure 23—Scalability of Example 3, Parallel cluster

This case tests the scalability of our grid, linear solver and the RAS preconditioner on a regular cluster. The first column of Table 6 is the number of MPI tasks (# procs) applied. The second column is the gridding time (second), including generation of the grid, partitioning, construction of the communication pattern and other modules. The third column is the overall calculation time (second), including the gridding, construction of the linear solver and preconditioner, and solution of the linear system. The forth column is the number of linear iterations of the linear solver. The last column is speedup obtained on the Parallel cluster. Here the 16 cores case is used as the basis case, whose speedup is assumed to be 16. For the gridding, from the table, we can see the grid generation and partitioning have excellent scalability and the running time reduces by a half if the MPI tasks are doubled. For the overall running time, when 32 MPI tasks are used, a speedup of 30.41 is obtained. When using 64 MPI tasks, a speedup of 59.45 is achieved. When using 128 MPI tasks, the speedup is 120.91, which means our simulator is 120.91 times faster than sequential simulators. The number of linear iterations of the GMRES(50) with the RAS preconditioner is robust, which keeps fixed even if many more MPI tasks are employed. This example shows our grid, linear solver and the RAS preconditioner have good scalability on a regular cluster, which is also demonstrated in Figure 23.

Example 4.

This example tests the same equation as in Example 3 with different configuration. The grid dimension is 600x600x600. It has 216 millions of grid cells. The problem is run on the IBM Blue Gene/Q system, and up of 4,096 cores are used, where 256 compute cards are employed. The GMRES(30) solver and RAS precondiiioner are applied. The overlap is also one, and the sub-domain problem is solved by ILUT(11, 1e-3). The stopping tolerance is 1e-8. The numerical summaries are listed in Table 7 and scalability results are shown in Figure 24.

Table 7—Numerical summaries of Example 4 on the IBM Blue Gene/Q

# procs	Griding (s)	Solver (s)	Overall (s)	# Itr	Speedup
512	42.68	444.35	495.72	870	512
1024	21.75	224.08	250.28	870	1014
2048	11.66	113.77	127.7	870	1988
4096	7.96	58.68	67.78	870	3745

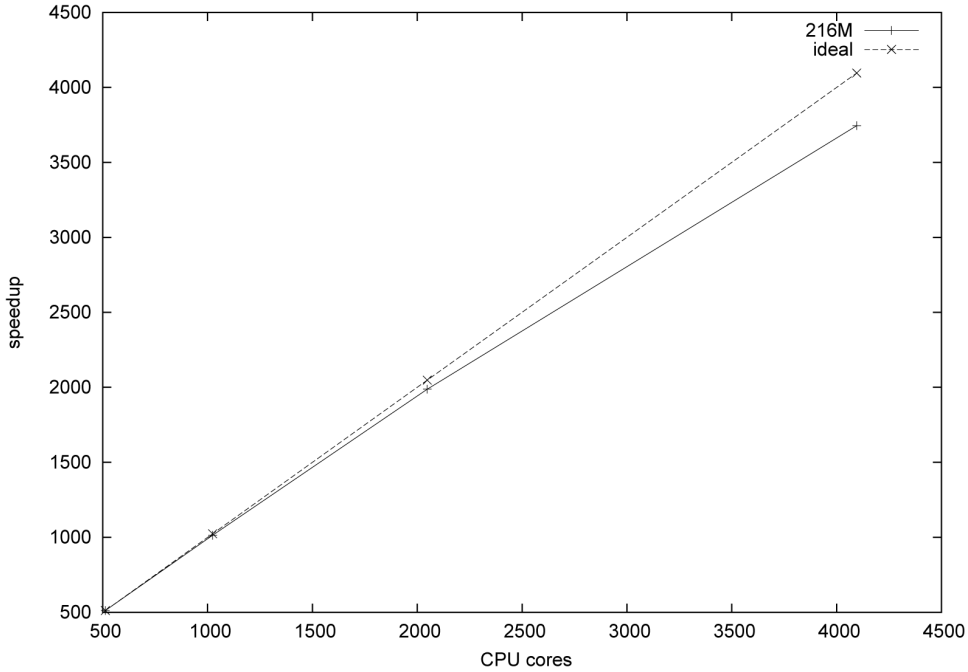

Figure 24—Scalability of Example 4 on the IBM Blue Gene/Q

The first column in Table 7 is the number of MPI tasks (cores) used. The second column is time used for gridding, including generation and partitioning of the grid. The third column is time used by the linear solver and the forth column is the overall time. The fifth column is the number of linear iterations. The last column is speedup obtained. The 512 cores case is the basis case for calculating the overall speedup. When using 1,024 CPU cores, the case is accelerated 1,014.1 times faster. When using 2,048 CPU cores, the case is 1,987.54 times faster. When using 4,096 CPU cores, our simulator is 3,744.6 times faster. For gridding, when we use up to 2,048 MPI tasks, its scalability is excellent. For the case with 4,096 MPI processors, the grid partitioning module introduces some overhead, where a one-dimensional method is applied by the third step of the HSFC partitioning method. The one-dimensional method is completed by one processor and it broadcasts the partitioning results of interval (0,1) to other processors. For the linear solver, its scalability is linear, which is excellent for practical calculation. The linear solver and preconditioner are also robust, whose convergence keeps the same for different MPI tasks. The overall running time from Table 7 shows our platform has good scalability, which is also demonstrated in Figure 24. From Figure 24, we can also see that the scalability of the case with 4,096 MPI tasks is lower than other cases. The reason is that 16 MPI tasks run on one compute card, and they compete memory bandwidth and network.

Example 5.

This example tests a refined SPE10 case for the two-phase oil-water model, where each grid cell is refined into 27 grid cells. This case has around 30 millions of grid cells and around 60 millions of unknowns. The stopping criterion for the inexact Newton method is 1e-3 and the maximal Newton iterations are 20. The BiCGSTAB solver is applied and its maximal iterations are 100. The preconditioner is the CPR-FPF

preconditioner The potential reordering and the Quasi- /MPES decoupling strategy are applied. The simulation period is 10 days. Up to 128 compute cards are used. The numerical summaries are shown in Table 8, and the speedup (scalability) is shown in Figure 25.

Table 8—Numerical summaries of Example 5

# Procs	# Steps	# Newton	# Solver	# Avg. solver	Time (s)	Avg. time (s)
64	50	315	3451	10.9	119167.4	378.3
128	48	286	3296	11.5	49488.7	173.0
256	54	323	4190	12.9	30423.2	94.1
512	52	329	3635	11.0	14276.5	43.3
1024	54	316	3969	12.5	7643.9	24.1

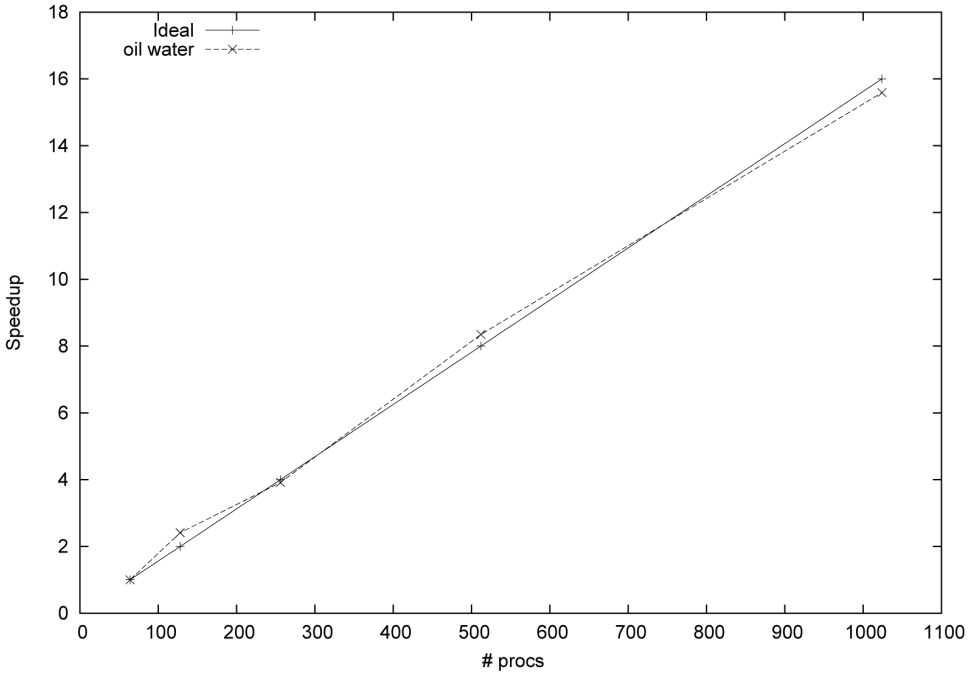

Figure 25—Scalability of Example 5, IBM Blue Gene/Q

In this example, up to 1,024 MPI tasks are employed and the simulation with 64 MPI tasks is used as the base case to calculate speedup and scalability. The numerical summaries in Table 8 show the inexact Newton method is robust, where around 50 time steps and around 300 Newton iterations are used for each simulation with different MPI tasks. The linear solver BiCGSTAB and the preconditioner CPR-FPF show good convergence, where the average number of linear iterations for each nonlinear iteration is between 10 and 13. The results mean our linear solver and preconditioner are effective and efficient. The overall running time and average time for each Newton iteration show our simulator has excellent scalability on IBM Blue Gene/Q, which is almost ideal for parallel computing. The scalability is also demonstrated in Figure 25. The running time and scalability curve also demonstrate our linear solver and preconditioner are scalable for large-scale simulation.

Example 6.

The example tests the scalability of the black oil simulator using a refined SPE10 geological model, where each grid cell is refined to 27 grid cells. The model has 30.3 millions of grid cells. The inexact Newton method is applied and the termination tolerance is 10^{-2}. *The linear solver is BiCGSTAB, whose maximal*

inner iterations are 100. The preconditioner is the CPR-FPF method and the overlap for the RAS method is one. The potential reordering and the ABF methods are enabled. The simulation period is 10 days. The maximal change allowed in one time step of pressure is 1,000 psi and the maximal change of saturation is 0.2. Up to 128 compute cards are used. Summaries of numerical results are shown in Table 9.

Table 9—Numerical summaries of Example 6

# Procs	# Steps	# Newton	# Solver	# Avg. solver	Time (s)	Avg. time (s)
64	33	292	1185	4.0	106265.9	363.9
128	33	296	1150	3.8	50148.3	169.4
256	33	299	1267	4.2	25395.8	84.9
512	33	301	1149	3.8	12720.5	42.2
1024	33	301	1145	3.8	6814.2	22.6
2048	33	298	1200	4.0	3632.0	12.1

Table 9 includes information for the nonlinear method, linear solver and running time. For all simulations, 33 time steps are used and the total Newton iterations are around 300. The results show the inexact Newton method is robust. For the linear solver and preconditioner, their convergence is good, which terminate in around 4 iterations. The results mean the linear solver and preconditioner are robust and effective for this highly heterogeneous model. The running time, average time per Newton iteration and scalability curve in Figure 26 show the scalability of our simulator, linear solver and preconditioner is good. When we use up to 1,024 MPI tasks and each compute card runs up to 8 MPI tasks, the scalability is excellent, which is ideal for parallel computing. However, when 2,048 MPI tasks are used and each compute card has 16 MPI tasks on it, the scalability is slightly lower. Even if this occurs, the IBM Blue Gene/Q supercomputer has better scalability than other parallel systems.

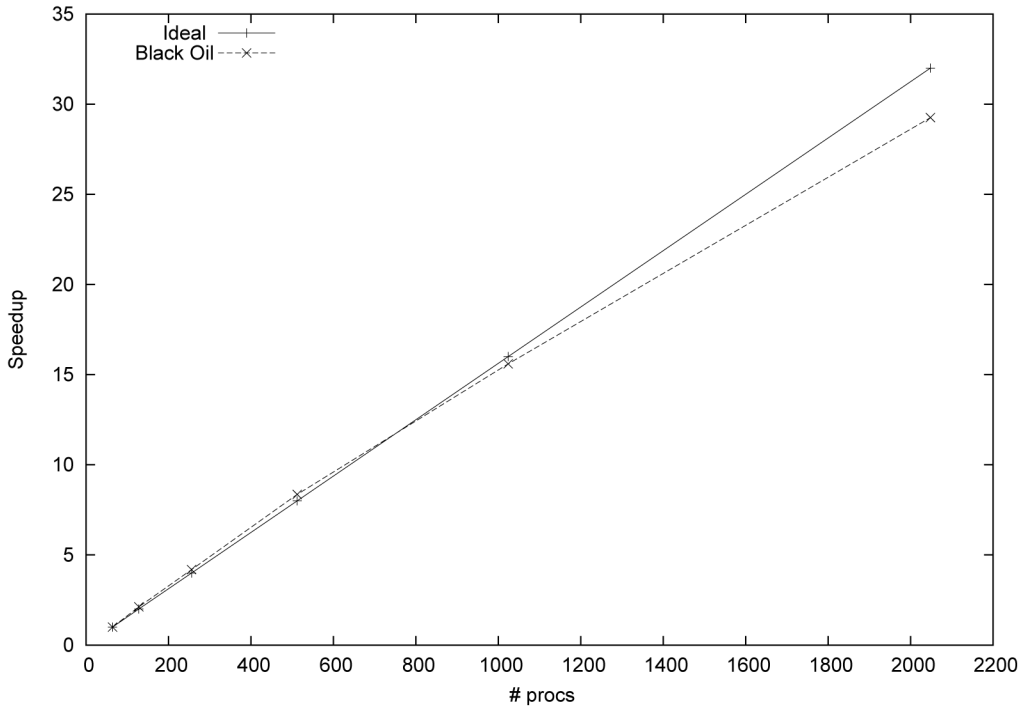

Figure 26—Scalability (speedup) of Example 6 on the IBM Blue Gene/Q

Example 7.

This example tests a refined SPE10 case for the two-phase oil-water model, where each grid cell is refined into 125 grid cells. This case has around 140 millions of grid cells and around 280 millions of unknowns. The stopping criterion for the inexact Newton method is 1e-2 and the maximal Newton iterations are 20. The GMRES(30) solver is applied and its maximal iterations are 100. The preconditioner is the CPR-FPF preconditioner The potential reordering and the Quasi-IMPES decoupling strategy are applied. The simulation period is 2 days. Up to 128 compute cards are used. The numerical summaries are shown in Table 10, and the speedup (scalability) is shown in Figure 27.

Table 10—Numerical summaries of Example 7

# Procs	# Steps	# Newton	# Solver	# Avg. solver	Time (s)	Avg. time (s)
256	36	225	5544	24.6	117288.1	521.2
512	36	226	5724	25.3	57643.1	255.0
1024	35	207	5446	26.3	27370.3	132.2
2048	36	209	5530	26.4	14274.9	68.3

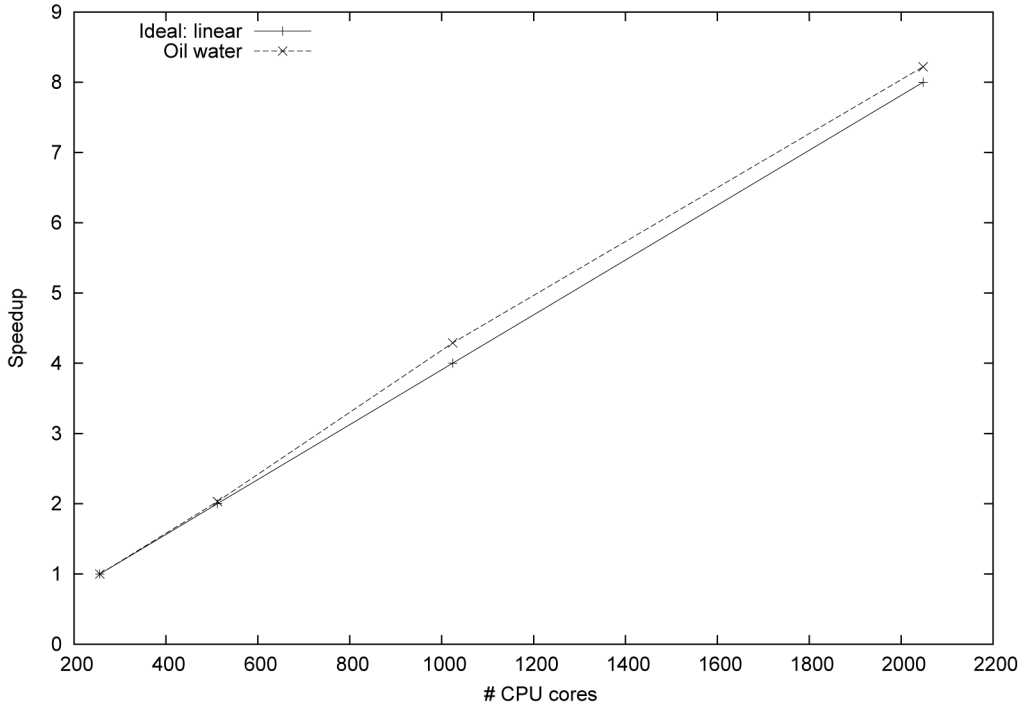

Figure 27—Scalability of Example 7, IBM Blue Gene/Q

In this example, up to 2048 MPI tasks are employed and the simulation with 256 MPI tasks is used as the base case to calculate speedup and scalability. The numerical summaries in Table 10 show the inexact Newton method is robust, where around 36 time steps and around 220 Newton iterations are used for each simulation with different MPI tasks. The linear solver GMRES(30) and the preconditioner CPR-FPF show good convergence, where the average number of linear iterations for each nonlinear iteration is around 26. The overall running time and average time for each Newton iteration show our simulator has excellent scalability on IBM Blue Gene/Q, which is almost ideal for parallel computing and shows slight superlinear scalability. The scalability is also demonstrated in Figure 27. The results show our linear solver and preconditioner are scalable for large-scale simulation.

Example 8.

The example tests a standard black oil case with 100 millions of grid cells and 300 millions of unknowns. The case is run on the Parallel cluster: The tolerance for inexact Newton method is 10^{-3}. *The linear solver is BiCGSTAB and the preconditioner is the CPR-FPF preconditioner The ABF decoupling strategy and potential reordering techniques are enabled. The maximal inner iterations for the linear solver are 100. The overlap for the RAS method is 1. The simulation period is fixed at 20 time steps. Up to 3,072 MP/ tasks are employed. Summaries of numerical results are shown in Table 11 and scalability is presented in Figure 28.*

Table 11—Numerical summaries of Example 8, Parallel cluster

# procs	# Newton	# Solver	Time (min)	Speedup	Efficiency
768	42	84	42.1	1.	100%
1536	42	84	22.1	1.90	95%
2304	42	84	15.2	2.77	92.3%
3072	42	84	11.0	3.83	95.75%

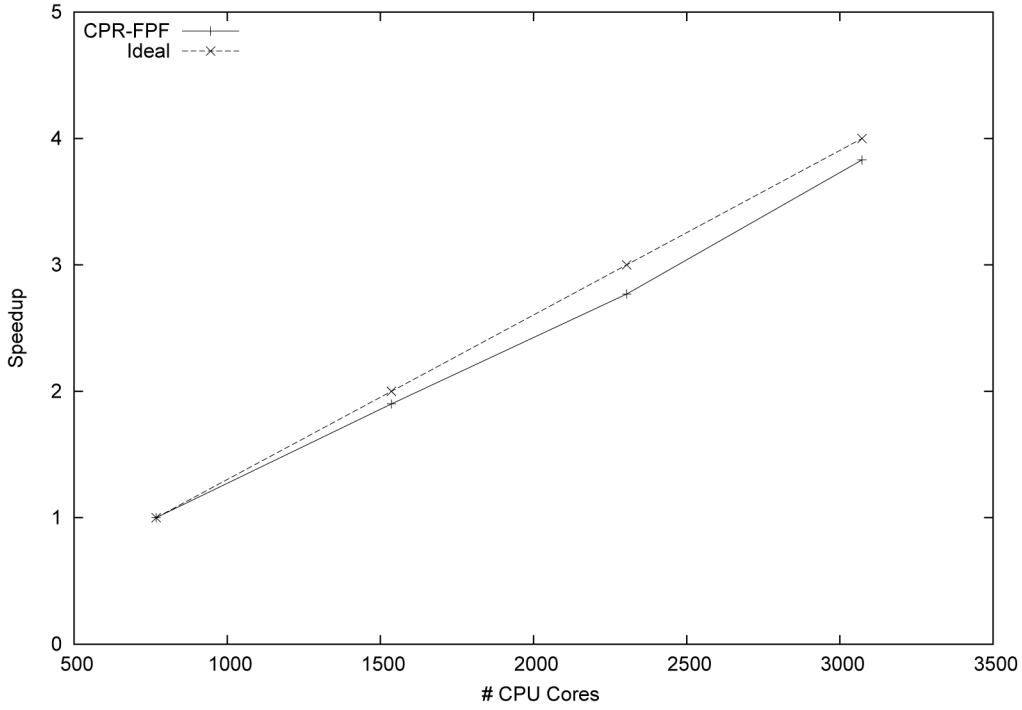

Figure 28—Scalability (Speedup) of Example 8, Parallel cluster

Numerical summaries, including the number of MPI tasks (# procs), the number of Newton iterations (# Newton), the number of linear iterations (# Solver), overall running time, speedup and efficiency, are listed in Table 11. The case with 768 MPI tasks is the base case and up to 3,072 MPI processors are employed. The results indicate that our linear solver and CPR-FPF preconditioner are effective and the CPR-FPF method is an efficient preconditioner for giant reservoir simulations, which converges in two iterations working with the linear solver BiCGSTAB. The preconditioner is also robust with respect to the grid cells and the dimension of linear systems, and is not sensitive to the number of MPI tasks. In the case of 3,072 CPU cores, the linear solver also converges in two iterations. The scalability and efficiency of our linear solvers and preconditioners are demonstrated again. When 1,536 CPU cores are employed, the speedup is 1.90 while the ideal speedup is 2. The efficiency is 95%. When using 3,072 CPU cores, the

speedup is 3.83 compared to the base case, and the efficiency is 95.75%, which is an excellent number for parallel computing. This example shows our linear solver, preconditioner and simulator are scalable, which are also demonstrated in Figure 28. From this figure, we can see the real speedup of this example is very close to the ideal speedup. We can conclude that our preconditioner and solver techniques are efficient for extremely large reservoir simulations, and our platform and black oil simulator are scalable.

Example 9.

This example tests a thermal model, which simulates a SAGD process. The model simulates a dead oil case, which includes a heavy oil phase, water phase and steam (gas phase). Its grid dimension is $400 \times 1000 \times 60$, *which has 24 millions of grid cells. The cell size is* $1 \times 1 \times 1$ *m. The model has five well pairs (one injector and one producer), a total of 10 wells. The standard Newton method is used to solve the coupled nonlinear system and its termination tolerance is 1e-2. The linear solver is GMRES(30) and the preconditioner is the RAS method. The termination tolerance of the linear solver is 1e-3. The potential reordering and ABF decoupling strategies are applied. 128 compute cards are usedfor all cases on the /BMBlue Gene/Q supercomputer The numerical summaries are in Table 12 and its scalability is shown in Figure 29.*

Table 12—Numerical summaries of Example 9, IBM Blue Gene/Q

# procs	Steps	# Newton	# Solver	#Avg. solver	Time (s)	Avg. time (s)
128	51	426	1238	2.9	66906.5	157.05
256	51	426	1238	2.9	34157.3	8O.18
512	51	426	1239	2.91	17724.4	41.61
1024	51	426	1239	2.91	9429.73	22.13

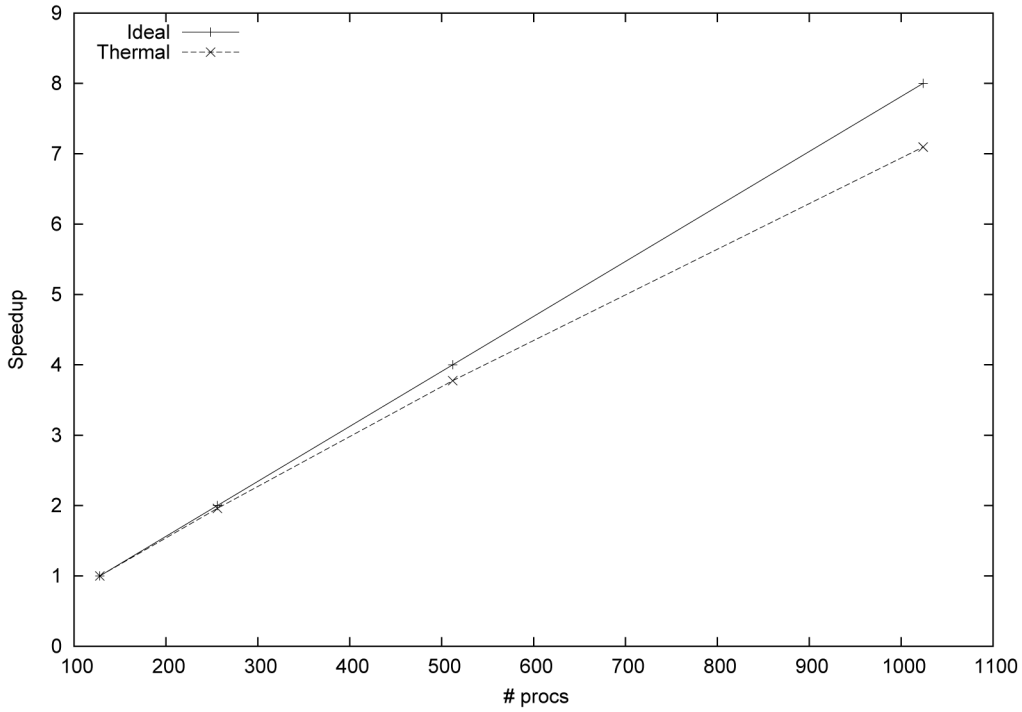

Figure 29—Scalability of Example 9, IBM Blue Gene/Q

The results in Table 12 show the standard Newton method is robust for all cases, where a total of 51 time steps and 426 Newton iterations are used. The linear solver, GMRES(30), and the RAS precondi-

tioner are robust. Each Newton iteration converges in around 2.9 linear iterations, which means the matrix pre-processing techniques, linear solver and preconditioner are effective and efficient. The case with 128 MPI tasks is the base case. When the number of MPI tasks is doubled, the parallel efficiency is around 98%, which is almost ideal. When 512 MPI tasks are used, each compute card has four MPI processors on it, and the efficiency is around 94.3%, which is also excellent for parallel computation. When 1,024 MPI tasks are employed, each compute card has 8 MPI tasks, which shares memory and network. They also read and write data simultaneously, and they send and receive data simultaneously. In this case, these MPI tasks compete memory bandwidth and network, which reduces their scalability. Even if this occurs, we still have an efficiency of 88.71%. From this example, we can see our thermal parallel simulator has good scalability, which is also shown in Figure 29.

Conclusion

Our work on developing an in-house parallel platform is presented in this paper, which provides grids, data, linear solvers and preconditioners for reservoir simulators. Various techniques and methods have been introduced, including the Hilbert space-filling curve method, topological partitioning method, structured grid, distributed matrices and vectors, and multistate preconditioners for reservoir simulations. Examples, including grid management, linear solvers, pressure equations, the black oil and thermal models, are presented to benchmark our platform. Numerical results show that our platform and simulators have excellent scalability and applications based on the platform can be sped up thousands of times faster. This paper also shows parallel computing is a powerful tool for large-scale scientific computing.

Acknowledgements

The support of Department of Chemical and Petroleum Engineering, University of Calgary and Reservoir Simulation Group is gratefully acknowledged. The research is partly supported by NSERC/AIEE/Foundation CMG, AITF Chairs, IBM Thomas J. Watson Research Center, and iCentre Simulation & Visualization. This research is enabled in part by support provided by WestGrid (www.westgrid.ca) and Compute Canada Calcul Canada (www.computecanada.ca).

References

1 K. H. Coats, Reservoir simulation, SPE-1987-48-PEH, *Society of Petroleum Engineers,* 1987.
2 H. Liu, K. Wang, Z. Chen, and K. E. Jordan, Efficient Multi-stage Preconditioners for Highly Heterogeneous Reservoir Simulations on Parallel Distributed Systems, SPE-173208-MS, SPE Reservoir Simulation Symposium held in Houston, Texas, USA, 23-25 February 2015.
3 Hans DS, Yang UM, Heys J, Reducing Complexity in Parallel Algebraic Multigrid Preconditioners, *SIAM Journal on Matrix Analysis and Applications* **27,** (2006), 1019–1039.
4 UM Yang, On the Use of Relaxation Parameters in Hybrid Smoothers, *Numerical Linear Algebra With Applications,* **11,** (2004), 155–172.
5 H. Liu, K. Wang, Z. Chen, J. Luo, S. Wu, B. Wang, Development of Parallel Reservoir Simulators on Distributed- memory Supercomputers, SPE-175573-MS, SPE Reservoir Characterisation and Simulation Conference and Exhibition, Abu Dhabi, UAE, 14-16 September, 2015.
6 K. H. Coats, Simulation of Steamflooding With Distillation and Solution Gas, SPE-5015-PA, *Society of Petroleum Engineers Journal,* 1976, 235–247.
7 K. H. Coats, Reservoir Simulation: State of the Art, SPE-10020-PA, *Journal of Petroleum Technology,* **34**(08), 1982, 1633–1642.
8 J. E. Killough and R. Bhogeswara. Simulation of compositional reservoir phenomena on a distributed-memory parallel computer. *Journal of Petroleum Technology* **43.11** (1991): 1368–1374.

9 J. M. Rutledge, D. R. Jones, W. H. Chen, and E. Y Chung, The Use of Massively Parallel SIMD Computer for Reservoir Simulation, SPE-21213, eleventh SPE Symposium on Reservoir Simulation, Anaheim, 1991.

10 G. Shiralkar, R.E. Stephenson, W. Joubert, O. Lubeck, and B. van Bloemen Waanders, A production quality distributed memory reservoir simulator, *SPE Reservoir Simulation Symposium*. 1997.

11 H. Liu, S. Yu and Z. Chen, Development of Algebraic Multigrid Solvers Using GPUs, SPE Reservoir Simulation Symposium, Feb 2013, Houston, USA.

12 T. Kaarstad, J. Froyen, P. Bjorstad, M. Espedal, Massively Parallel Reservoir Simulator, SPE-29139, presented at the 1995 Symposium on Reservoir Simulation, San Antonio, Texas, 1995.

13 J. E. Killough, D. Camilleri, B.L. Darlow, J. A. Foster, Parallel Reservoir Simulator Based on Local Grid Refinement, SPE-37978, SPE Reservoir Simulation Symposium, Dallas, 1997.

14 A.H. Dogru, H.A. Sunaidi, L.S. Fung, W.A. Habiballah, N. Al-Zamel, K.G. Li, A parallel reservoir simulator for large-scale reservoir simulation, *SPE Reservoir Evaluation & Engineering 5.1* (2002): 11–23.

15 M. Parashar, and I. Yotov, An environment for parallel multi-block, multi-resolution reservoir simulations, Proceedings of the 11th International Conference on Parallel and Distributed Computing Systems (PDCS 98), Chicago, IL, International Society for Computers and their Applications (ISCA), 1998.

16 K. H. Coats, A Highly Implicit Steamflood Model, SPE-6105-PA, *Society of Petroleum Engineers Journal,* **18**(05), 1978, 369–383.

17 A.H. Dogru, L. S. Fung, U. Middya, T. Al-Shaalan, J.A. Pita, A next-generation parallel reservoir simulator for giant reservoirs, SPE/EAGE Reservoir Characterization & Simulation Conference. 2009.

18 L. Zhang, A Parallel Algorithm for Adaptive Local Refinement of Tetrahedral Meshes Using Bisection, Numer. Math.: Theory, *Methods and Applications,* 2009, **2,** 65–89.

19 K. H. Coats, An Equation of State Compositional Model, SPE-8284-PA, *Society of Petroleum Engineers Journal,* **20**(05), 1980, 363–376.

20 K. Mukundakrishnan, K. Esler, D. Dembeck, V. Natoli, J. Shumway, Y Zhang, J. R. Gilman, H. Meng, Accelerating Tight Reservoir Workflows With GPUs, SPE-173246-MS, SPE Reservoir Simulation Symposium, Houston, Texas, USA, Feb 2015.

21 K. Esler, K. Mukundakrishnan, V. Natoli, J. Shumway, D. Dembeck, Y. Zhang and J. Gilman, Parallel Efficiency and Algorithmic Optimality in Reservoir Simulation on GPUs, SPE Large Scale Computing and Big Data Challenges in Reservoir Simulation Conference and Exhibition, 13 September, 2014, Istanbul Turkey.

22 J. Shumway, K. Mukundakrishnan, K. Esler, V. Natoli, Y. Zhang, and J. Gilman, Rapid High-Fidelity Reservoir Simulation with Fine-Grained Parallelism on Multiple GPUs, EAGE Workshop on High Performance Computing for Upstream, 3 September 2014, Crete, Greece.

23 K. Esler, K. Mukundakrishnan, V. Natoli, J. Shumway, Y. Zhang and J. Gilman, Realizing the Potential of GPUs for Reservoir Simulation, ECMOR XIV – 14th European Conference on the Mathematics of Oil Recovery, 8 September 2014.

24 L. Zhang, T. Cui, and H. Liu, A set of symmetric quadrature rules on triangles and tetrahedra, *J. Comput. Math,* 2009, **27**(1), 89–96.

25 Y. Saad, Iterative methods for sparse linear systems, *SIAM,* 2003.

26 J.R. Wallis, Incomplete Gaussian elimination as a preconditioning for generalized conjugate gradient acceleration, *SPE Reservoir Simulation Symposium,* 1983.

27 G.A. Behie, and P. A. Forsyth, Jr., Incomplete factorization methods for fully implicit simulation of enhanced oil recovery, *SIAM Journal on Scientific and Statistical Computing* **5.3** (1984): 543–561.
28 J.R. Wallis, R. P. Kendall, and T. E. Little, Constrained residual acceleration of conjugate residual methods, *SPE Reservoir Simulation Symposium,* 1985.
29 H. Cao, T. Schlumberger, A. Hamdi, J.R. Wallis, H.E. Yardumian, Parallel scalable unstructured CPR-type linear solver for reservoir simulation. *SPE Annual Technical Conference and Exhibition*. 2005.
30 Z. Chen, H. Liu, S. Yu, B. Hsieh and L. Shao, GPU-based parallel reservoir simulators, 21st International Conference on Domain Decomposition Methods, 2012, France.
31 R. E. Bank, T. F. Chan, W. M. Coughran, Jr., R. K. Smith, The Alternate-Block-Factorization procedure for systems of partial differential equations, *BIT Numerical Mathematics* **29.4** (1989): 938–954.
32 T. M. Al-Shaalan, H. M. Klie, A. H. Dogru, M. F. Wheeler, Studies of Robust Two Stage Preconditioners for the Solution of Fully Implicit Multiphase Flow Problems. *SPE Reservoir Simulation Symposium*. 2009.
33 X. Hu, W. Liu, G. Qin, J. Xu, Z. Zhang, Development of a fast auxiliary subspace pre-conditioner for numerical reservoir simulators, SPE Reservoir Characterisation and Simulation Conference and Exhibition. 2011.
34 H. Liu, S. Yu, Z. Chen, B. Hsieh and L. Shao, Sparse Matrix-Vector Multiplication on NVIDIA GPU, *International Journal of Numerical Analysis & Modeling, Series B,* 2012, Volume **3,** No. 2, 185–191.
35 K. H. Coats, Effects of Grid Type and Difference Scheme on Pattern Steamflood Simulation Results, SPE-11079-PA, *Journal of Petroleum Technology,* **38**(05), 1986, 557–569.
36 Z. Chen, H. Liu and B. Yang, Parallel triangular solvers on GPU, Proceedings of International Workshop on DataIntensive Scientific Discovery (DISD), Shanghai University, Shanghai, China, August 1-4, 2013.
37 C. Feng, S. Shu, J. Xu and C. Zhang, A Multi-Stage Preconditioner for the Black Oil Model and Its OpenMP Implementation, 21st International Conference on Domain Decomposition Methods, 2012, France.
38 T. Chen, N. Gewecke, Z. Li, A. Rubiano, R. Shuttleworth, B. Yang and X. Zhong, Fast Computational Methods for Reservoir Flow Models, *Technical report,* University of Minnesota, 2009.
39 A. Elli, and O. B. Widlund. Domain decomposition methods: algorithms and theory. Vol. **34**. Springer, 2005.
40 AJ Cleary, RD Falgout, VE Henson, JE Jones, TA Manteuffel, SF McCormick, GN Miranda, JW Ruge, Robustness and Scalability of Algebraic Multigrid, *SIAM J. Sci. Comput.,* **21,** 2000, 1886–1908.
41 H. Liu, S. Yu, Z. Chen, B. Hsieh and L. Shao, Parallel Preconditioners for Reservoir Simulation on GPU, SPE-152811, SPE Latin America and Caribbean Petroleum Engineering Conference, 16-18 April, 2012, Mexico City, Mexico.
42 X. Cai, and M. Sarkis, A restricted additive Schwarz preconditioner for general sparse linear systems, *SIAM Journal on Scientific Computing* **21.2** (1999): 792–797.
43 H. Klie, H. Sudan, R. Li, and Y. Saad, Exploiting capabilities of many core platforms in reservoir simulation, SPE RSS Reservoir Simulation Symposium, 21-23 February 2011.
44 H. Liu, Z. Chen and B. Yang, Accelerating preconditioned iterative linear solver on GPU, *International Journal of Numerical Analysis and Modelling: Series B,* **5**(1-2), 2014, 136–146.

45 R. Li and Y. Saad, GPU-accelerated preconditioned iterative linear solvers, Technical Report umsi-2010-112, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2010.

46 AJ Cleary, RD Falgout, VE Henson, JE Jones, Coarse grid selection for parallel algebraic multigrid, in Proceedings of the fifth international symposium on solving irregularly structured problems in parallel, Springer-Verlag, New York, 1998.

47 A Sedighi, Y Deng, P Zhang, Fairness of Task Scheduling in High Performance Computing Environments, *Scalable Computing: Practice and Experience,* vol. **15,** no. 3, pp. 273–285, 2014.

48 Y Deng, P Zhang, Perspectives of Exascale Computing, *Journal of New Computing Architectures and Applications,* vol. **1,** Sep. 2010.

49 MT Jones, PE Plassman, A parallel graph coloring heuristic, *SIAM Journal on Scientific Computing,* **14**(1993): 654–669.

50 M Luby, A simple parallel algorithm for the maximal independent set problem, *SIAM Journal on Computing,* **15**(1986), 1036–1053.

51 S. Yu, H. Liu, Z. Chen, B. Hsieh, and L. Shao, GPU-based Parallel Reservoir Simulation for Large-scale Simulation Problems, SPE Europec/EAGE Annual Conference Copenhagen, Denmark, 2012.

52 JW Ruge and K Stuben, Algebraic multigrid (AMG), in: S.F. McCormick (Ed.), *Multigrid Methods, Frontiers in Applied Mathematics,* Vol. **5,** SIAM, Philadelphia, 1986.

53 A Brandt, SF McCormick, J Ruge, Algebraic multigrid (AMG) for sparse matrix equations D.J. Evans (Ed.), *Sparsity and its Applications,* Cambridge University Press, Cambridge, 1984, 257–284.

54 Z. Chen, H. Liu, S. Yu, B. Hsieh, L. Shao, Reservoir Simulation on NVIDIA Tesla GPUs, The Eighth International Conference on Scientific Computing and Applications, University of Nevada, Las Vegas, April, 2012.

55 RD Falgout, An Introduction to Algebraic Multigrid, Computing in Science and Engineering, *Special Issue on Multigrid Computing,* **8,** 2006, 24–33.

56 Z. Chen, H. Liu, and B. Yang, Accelerating iterative linear solvers using multiple graphical processing units, *International Journal of Computer Mathematics,* **92**(7): 1422–1438, 2015.

57 K. Stuben, T. Clees, H. Klie, B. Lou, M.F. Wheeler, Algebraic multigrid methods (AMG) for the efficient solution of fully implicit formulations in reservoir simulation, *SPE Reservoir Simulation Symposium*. 2007.

58 K. Stuben, A review of algebraic multigrid, *Journal of Computational and Applied Mathematics* **128.1** (2001): 281–309.

59 Z. Chen, G. Huan, and Y. Ma. Computational methods for multiphase flows in porous media, Vol. **2**. Siam, 2006.

60 F. Kwok, and H. Tchelepi, Potential-based reduced newton algorithm for nonlinear multiphase flow in porous media, *Journal of Computational Physics* **227.1** (2007): 706–727.

61 HYPRE: high performance preconditioner. http://acts.nersc.gov/hypre/

62 R. D. Falgout, and U.M. Yang, HYPRE: A library of high performance preconditioners, *Lecture Notes in Computer Science,* Springer Berlin Heidelberg, 2002. 632–641.

63 E. Boman, K. Devine, R. Heaphy, B. Hendrickson, V. Leung, L.A. Riesen, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell and J. Teresco, Zoltan v3: Parallel Partitioning, Load Balancing and Data-Management Services, User's Guide, Sandia National Laboratories Tech. Rep. SAND2007-4748W, 2007.

64 H. Liu, Dynamic Load Balancing on Adaptive Unstructured Meshes, 10th IEEE International Conference on High Performance Computing and Communications, 2008.

65 G. Li, J. Wallis, G. Shaw, A Parallel Linear Solver Algorithm for Solving Difficult Large Scale Thermal Models, SPE-173207-MS, SPE Reservoir Simulation Symposium, Houston, Texas, USA, Feb 2015.
66 M.A. Christie, and M. J. Blunt, Tenth SPE comparative solution project: A comparison of upscaling techniques. *SPE Reservoir Evaluation & Engineering* **4.4** (2001): 308–317.
67 G. Karypis, K. Schloegel, and V. Kumar, Parmetis: Parallel graph partitioning and sparse matrix ordering library. Version 1.0, *Dept. of Computer Science,* University of Minnesota (1997).
68 S Lacroix, YV Vassilevski, MF Wheeler, Decoupling preconditioners in the implicit parallel accurate reservoir simulator (IPARS), *Numerical linear algebra with applications,* **8**(8), 2001: 537–549.
69 B. Wang, S. Wu, Q. Li, X. Li, H. Li, C. Zhang, J. Xu, A Multilevel Preconditioner and Its Shared Memory Implementation for New Generation Reservoir Simulator, SPE-172988-MS, SPE Large Scale Computing and Big Data Challenges in Reservoir Simulation Conference and Exhibition, 15-17 September, Istanbul, Turkey, 2014.
70 Q. Li, J. Xu, X. Li, H. Li, S. Wu, B. Wang, C. Zhang, Multilevel Preconditioners for a New Generation Reservoir Simulator, SPE-166011-MS, SPE Reservoir Characterization and Simulation Conference and Exhibition, 16-18 September, Abu Dhabi, UAE, 2013.
71 C. Li, Y. Feng, Algorithm for analyzing n-dimensional Hilbert curve, vol. **3739,** Springer Berlin/Heidelberg, 2005; 657–662.