PSPIKE

Parallel General Sparse Linear System Solver
Manual Version 1.0.1

Madan Sathe
Department of Mathematics and Computer Science
University of Basel
Switzerland

Olaf Schenk
Institute of Computational Science
University of Lugano
Switzerland

Murat Manguoğlu
Department of Computer Engineering
Middle East Technical University
Turkey

Ahmed H. Sameh
Department of Computer Science
Purdue University
USA

August 20, 2012

1 The Idea of PSPIKE

PSPIKE is a software package solver for parallel solution of large sparse linear systems. Solving large sparse linear systems is usually the most time consuming operation in many scientific computing applications. Existing parallel sparse linear systems rely on sparse Gaussian elimination while our software package is based on a totally different factorization combined with state-of-the art matrix reordering and partitioning routines. The objective of these reordering and partitioning routines is to both reduce the amount of communication operations and limit the communication to be between the neighboring nodes (or processes). These novel approaches lead naturally to a more favorable inter-process communication requirement, and hence improve the parallel scalability without sacrificing from robustness of the solver.

2 The algorithm

The PSPIKE algorithm, introduced in [11, 21], is based the Spike decomposition [17, 16, 8, 1, 4, 14, 15] to solve linear system $Ax = f$ iteratively using a preconditioner where the preconditioner is banded and sparse within the band.

The use of weighted matrix permutation to extract an effective banded preconditioner where the preconditioner is treated as dense within the band is introduced in [10]. In PSPIKE, we extend this idea by introducing other combinatorial techniques [19] which are applied after symmetric weighted permutations to enhance the effectiveness of the preconditioner.

For simplicity the effect of the a series of symmetric and nonsymmetric permutations to the linear system can be shown as a single nonsymmetric permutation,

$$PAQ^TQx = Pf \tag{1}$$

which yields a permuted linear system

$$\tilde{A}\tilde{x} = \tilde{f} \tag{2}$$

where $\tilde{A} = PAQ^T$, $\tilde{x} = Qx$, and $\tilde{f} = Pf$. From the permuted coefficient matrix a more effective banded preconditioner M can be extracted. For a given banded preconditioner M, its Spike decomposition $M = DS$ for a given number of partitions is computed by simply extracting the block diagonal matrix D and computing $S = D^{-1}M$. The last operation is performed by solving sparse linear systems with multiple right hand sides for which PSPIKE uses the Pardiso direct solver [20]. Solution is computed only partially that is just enough to form a *truncated* reduced system which is more suitable for parallelism and faster to compute but might be inaccurate if the reordered matrix is far from being diagonally dominant. The experiments, however, show that for most linear systems the combined effect of permutations and sometimes scalings will give a reasonably effective and scalable preconditioner. In summary, the preconditioner satisfies the following equality

$$\tilde{A} = M + R \tag{3}$$

in which the matrix R is the remainder matrix that combines the effect of the truncation of the reduced system and elements that are excluded from the preconditioner. $||R||_F$ is assumed to be small as a result of the reordering schemes that are applied on A. The solution process after computing DS factorization involves solution of independent linear systems where the coefficient matrices are the blocks diagonals of D matrix and solution of a small reduced system. Then, the algorithm can be formulated as follows

```
Solve  (DS + R)x̃ = f̃  via a preconditioned iterative method
  (with preconditioner M = DS);
  * solve systems in the form of Mz = y for various y;
  * multiply t = Au for various u;
  * other operations;
End
```

After computing $\tilde{x}$, x is obtained by $x = Q^T \tilde{x}$.

3 Software Implementation Aspects

PSPIKE is implemented for distributed memory architectures and follows the MPI–OpenMP programming paradigm. Thus, the source code must be compiled with an MPI compiler, linked with an OpenMP library, and has been tested with the PATHSCALE, GNU, and INTEL compiler suites using the optimization flag "-O3." Calling PSPIKE requires that MPI is initialized and finalized in the calling program. Several external libraries for computing reorderings via graph algorithms and solutions of sparse and dense linear equation systems are implemented in PSPIKE. In this chapter, the interplay between these libraries is explained in detail.

3.1 Input Data

The square matrix of the linear equation system can be either symmetrically or unsymmetrically stored and is expected to be available in a 1-based CSR storage format as the computational kernel of PSPIKE is written in Fortran. The right-hand side of the linear system must be provided as a dense vector. The control over PSPIKE is given to the user via an external option file which enables, for instance, different reordering strategies and the debug output level of the library. The bandwidth k and outer tolerance ϵ_{out} can be put either via the interface or via the option file into the solver. The values of the bandwidth and tolerance are read from the option file if their values are set to 0 in the interface.

3.2 The PSPIKE Phases

The implementation of PSPIKE consists of five phases: initialization, reordering, numerical hybrid factorization, solving, and destroy. The work flow of

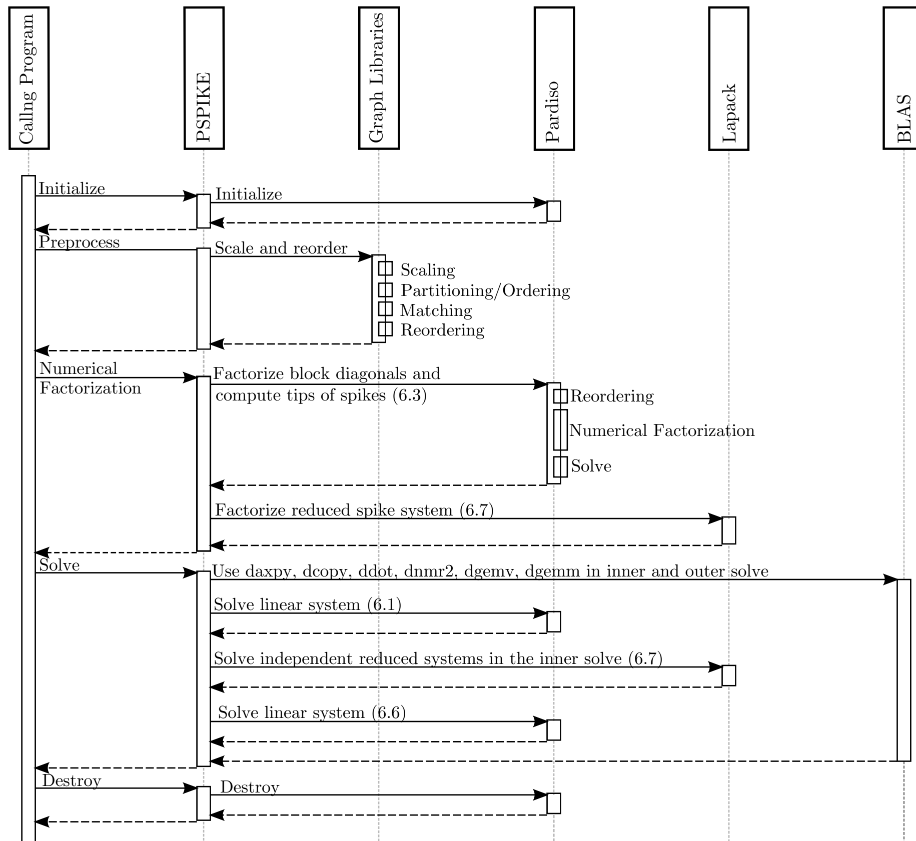

Figure 1: Sequence diagram showing the interplay between PSPIKE, external graph libraries, Pardiso, BLAS, and Lapack.

PSPIKE based on the five phases is illustrated in Fig. 1. The focal point of the figure is on the interplay between PSPIKE and the graph algorithm libraries Lapack (e.g., [12, 13]), BLAS (e.g., [3, 12]), and Pardiso [20].

Starting from an external program, the initialization phase of PSPIKE must first be called. In this phase, the PSPIKE option file is read and internal methods and data structures are activated to realize the work flow determined by the enabled options. Additionally, the direct solver Pardiso is initialized. Typically, this phase is called only once for the entire solving process.

In the preprocessing phase, the main task is to reorder the matrix in order to obtain the PSPIKE structure with block diagonal and coupling block matrices. Therefore, several interfaces to existing implementations of graph algorithms for solving the graph matching, graph partitioning and ordering, and weighted subgraph problems are embedded into PSPIKE. The software libraries provide permutation vectors which reorder nonzero elements into the PSPIKE structure. After each preprocessing step, auxiliary functions of Pardiso are used

to apply the permutations to the matrix. The available combinations to stick reordering routines together will be discussed in the next section.

The numerical hybrid factorization phase involves all MPI processes in the computation. It contains the factorization of the block diagonal matrices and solving of sparse linear systems via PARDISO in parallel. PARDISO internally reorders the matrix before factorize and solve the system. The solutions of the sparse linear system with the multiple right-hand sides are the dense spikes which are factorized using the function `dgetrf` of LAPACK. However, this factorization is only required if the preconditioned inner solve is enabled in the options.

In the outer and inner solve, several routines of BLAS are used to compute vector and matrix operations in the iterative solver BiCGSTAB. After each call of a BLAS routine the control flow is actually given to PSPIKE which is not explicitly shown in this figure. In the outer solve, PARDISO is called twice in the preconditioner call. In the inner solve, linear systems composed of the reduced spike system are solved using LAPACK.

In the destroy phase, the entire memory allocated by PSPIKE, LAPACK, and PARDISO is freed.

3.3 Combining Reordering Strategies

In the preprocessing phase, the reorderings applied to the matrix are crucial for obtaining a strong preconditioner. As the graph partitioning and spectral reordering heuristics provide solutions for $\mathcal{NP}$-hard problems, the quality of the solution considerably differs from application to application. Thus, it is recommended to find the best combination of reordering and scaling strategies in order to cover almost all entries of the matrix by the preconditioner. Note that PSPIKE automatically computes the quality of the preconditioner with regards to the coverage of the number of nonzeros and the number of heavy-weighted entries. In Fig. 2, the interfaces to follow scaling and reordering strategies are currently implemented which perform

- a scaling of the matrix entries (2×2 MAT [5], PAUL [18], MC64 [6])
- a spectral reordering (MC73 [6], TRACEMIN-FIEDLER [9], RCM [5]) or graph partitioning (MONDRIAAN [2], METIS [7]) of the corresponding graph to obtain heavy-weighted block diagonal matrices and nonempty candidate blocks
- a permuting of heavy-weighted entries on the diagonal via weighted graph matching algorithms (MC64 [6], PAUL [18])
- a reordering of the matrix to minimize its bandwidth and profile (SLOAN [6])
- a reordering of heavy-weighted entries into coupling blocks by solving the weighted dense subgraph problem (DEMIN, FIRSTFIT, (1+1)–EA [18]).

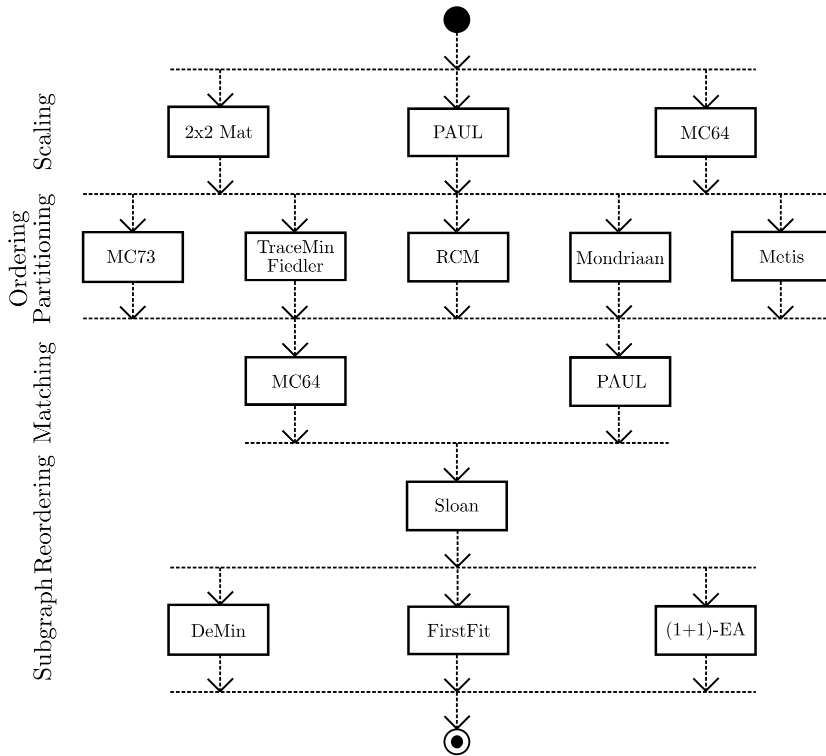

Figure 2: Component diagram of the preprocessing phase of PSPIKE.

At each step, at most one of the available libraries can be optionally included into the entire scaling and reordering schemes. After each call of the external software libraries the matrix is immediately scaled using the dual variables of the matching algorithm, and also directly reordered using the permutation returned by the software libraries. It is possible through the PSPIKE option file to enable and disable scaling and reordering strategies.

4 How to use PSPIKE

In Fortran, the hybrid solver PSPIKE can be invoked by the following routine:

```
subroutine pspike(job, n, m, ia, ja, a, rhs, sol,
                  bandwidth, tol, nrhs, info)

integer (in) :: job(1), bandwidth(1), nrhs(1), info(2)
integer (in) :: n(1), m(1), ia(n+1), ja(m)
double precision (in) :: a(m)
double precision (inout) :: rhs(n), sol(n)
```

```
double precision (in) :: tol(1)
```

In C, PSPIKE can be invoked by the following command (note, the attached "_" to the `pspike` call):

```
pspike_(int *job,
        int *n, int *m, int *ia, int *ja, double *a,
        double *rhs, double *sol,
        int *bandwidth, double *tol,
        int *nrhs, int *info)
```

4.1 Arguments of PSPIKE

Each of the arguments in the call of PSPIKE is explained in detail:

⋆ `job` — integer (1)

> The solving of a linear equation system with PSPIKE is split into five phases: initialization (`job` = 0), preprocessing (`job` = 1), numerical hybrid factorization (`job` = 2), solving the linear equation system (`job` = 3), and destroy (`job` = 4). Thus, the flag `job` controls the flow over the five phases. Typically, PSPIKE need to be initialized and destroyed once in a calling program. The preprocessing phase with `job` = 1 and the numerical factorization with `job` = 2 must be called at least one time for the entire process. The actual solving of the linear system and reading of the right-hand side(s) are performed in the solving phase with `job` = 3.

⋆ `n` — integer (1)

> The number of equations of the linear equation system is given by `n`, `n` > 0.

⋆ `m` — integer (1)

> The number of nonzeros in the coefficient matrix is given by `m`, `m` > 0.

⋆ `ia` — integer (`n+1`)

> The size of the integer array is `n+1`, where `ia`(i) points to the first column index of row i in the compressed sparse row storage format.

⋆ `ja` — integer (`m`)

> The size of the integer array is equal to the number of nonzeros `m`, where `ja` stores the column indices in the CSR format. The indices in each row must be sorted in an increasing order.

⋆ `a` — double precision (`m`)

The numerical values of the matrix are stored in `a` (in the same order as `ja`) following the CSR storage format.

⋆ `nrhs` — integer (1)

Number of right-hand sides of the linear equation system.

⋆ `rhs` — double precision (`n`, `nrhs`)

The right-hand side vector or matrix. The solution overwrites this array per default.

⋆ `sol` — double precision (`n`, `nrhs`)

A starting solution for the iterative solver can be provided here.

⋆ `bandwidth` — integer (1)

The bandwidth in relation to the upper triangular part of the matrix.

⋆ `tol` — double precision (1)

The stopping tolerance of the iterative solver.

⋆ `info` — integer (2)

The `info` array contains, on the one hand, the convergence status of PSPIKE and, on the other hand, extra options like the input of an existing preconditioner or starting solution for the iterative solver. In detail, the `info` array can be described by following Table 1.

Table 1: Description of the options given to the user by the `info` array.

Info Index	**Description**
`info` $(1) = 0$	On return, PSPIKE converges to the desired accuracy.
`info` $(1) = -1$	On return, PSPIKE was not able to converge to the desired residual. An error message is printed on the screen.
`info` $(1) = 9$	A preconditioner is given to PSPIKE in the phases before the solving phase; in the solving phase, the info flag is read and it is assumed that the linear system of interest is provided as input in this phase.
`info` $(1) = -2$	The partitioner/spectral heuristic is called once for the entire process (only of interest for the preprocessing phase)
`info` $(2) = -1$	The option file `pspike.opt2` should be read.
`info` $(2) = -2$	A starting solution is given to the iterative solver (only of interest for the solving phase). Per default the starting solution is filled with zeros in the iterative solver.

4.2 The Option File `pspike.opt`

The user has control over the crucial aspects in PSPIKE through the option file `pspike.opt` as, for instance, the applied reordering routines or iterative/direct solver parameters. This file will be read once at the initialization phase, and mainly influences the behavior of PSPIKE. If no option file is available, PSPIKE runs in its default mode. It is recommended that one figure out the best reordering strategy for the application. In the subsequent Table 2, the option itself, its possible values, and the description of the option are listed. The term in the column "Value" is always greater than or equal to 0. If the value is not explicitly given, then $\mathbb{N}$ indicates an integer, $\mathbb{R}$ a double precision number in scientific notation, $\mathbb{B}$ a boolean with the values $\{0, 1\}$, and $\mathbb{T}$ a string with at least one character. In the case of $\mathbb{B}$, a 1 enables the option and a 0 disables the option. The options then can be configured are given in Table 2.

Table 2: Description of the configuration file `pspike.opt`.

Name	Value	Description
`output_file`	$\mathbb{T}$	Name of the debug output file. The MPI rank is appended on the end of the string automatically.
`standalone_version`	$\mathbb{B}$	If enabled PSPIKE has the full control over the termination process (see also option `with_ipopt`).
`is_matrix_symmetric`	$\mathbb{B}$	If enabled on entry, only the upper triangular part of the matrix is stored.
`pspike_tol`	$\mathbb{R}$	Desired tolerance of the relative residual in PSPIKE; only enabled if the parameter `pspike_tol` in the call of the interface is zero on entry.
`bicgstab_max_iter`	$\mathbb{N}$	Number of the maximum BiCGStab iterations.
`bicgstab_inner_tol`	$\mathbb{R}$	Desired tolerance of the relative residual in the inner BiCGStab solver.
`bicgstab_inner_iter`	$\mathbb{N}$	Number of maximum iterations for the inner BiCGStab solver.
`with_prec_bicgstab_inner`	$\mathbb{B}$	If enabled, the preconditioner in the inner BiCGStab is used.
`bicgstab_max_stagnation`	$\mathbb{N}$	If the relative residual in the outer BiCGStab solver does not improve within the given number, then the current best solution is returned.
`pspike_bandwidth`	$\mathbb{N}$	The bandwidth > 0 of the PSPIKE solver can be controlled over the option file if the parameter `bandwidth` in the call of the interface is zero on entry.
`with_fast_matvec`	0	In the SpMV in BiCGStab, the entire vector is gathered across the processes.
	1	In the SpMV in BiCGStab, entries of the vector are only communicated to a process if required by the corresponding process.
`pspike_debug_level`	$0 - 6$	Debug print level of PSPIKE; messages are printed in the debug output file for each process.
`pardiso_iparm_31`	$0 - 2$	Control over Pardiso option "Partial solve for sparse right-hand sides and sparse solution."
`pardiso_32bit`	$\mathbb{B}$	If enabled, the 32-bit factorization (see iparm(29)) of Pardiso is used.
`pardiso_msglvl`	$\mathbb{N}$	If enabled, Pardiso prints debug messages on stdout.
`pardiso_iter_ref`	$\mathbb{N}$	Number of iterative refinement steps in Pardiso
`with_scaling`	$\mathbb{B}$	Enables the scaling of the entire matrix in general.

Table 2: Description of the configuration file `pspike.opt`.

Name	Value	Description
`scaling_symmetric`	$\mathbb{B}$	Enables the 2×2 matching including its scaling.
`with_mc64_scaling`	$\mathbb{B}$	Enables the MC64 scaling method.
`with_auction_scaling`	$\mathbb{B}$	Enables the auction scaling method.
`with_rcm_reordering`	$\mathbb{B}$	If enabled, RCM is used.
`with_mc64_matching`	$\mathbb{B}$	Enables the MC64 matching implementation.
`with_mc73_spectral`	$\mathbb{B}$	Enables the spectral heuristic implementation MC73.
`with_mc73_hager`	$\mathbb{B}$	If enabled, Hager method is enabled in the implementation MC73.
`with_tracemin_fiedler`	$\mathbb{B}$	Enables the spectral heuristic TraceMin–Fiedler.
`fiedler_tolerance`	$\mathbb{R}$	Desired residual for the Fiedler vector computed in spectral heuristics.
`with_fill_couplings`	$\mathbb{B}$	Enables the reordering of the weighted subgraph problem in general.
`with_sort_heuristic`	0	DeMin with objective to maximize the weight in the subgraph is used.
	1	FirstFit with MaxEntry is used.
	2	DeMin with objective to maximize the number of nonzeros in the subgraph is used.
	3	The $(1+1)-$EA is used.
`with_metis`	$\mathbb{B}$	If enabled, partitioner Metis is used.
`with_auction`	$\mathbb{B}$	Enables the auction algorithm.
`with_mondriaan`	$\mathbb{B}$	If enabled, partitioner Mondriaan is used.
`with_ml_sloan`	$\mathbb{B}$	Enables the multilevel Sloan algorithm (implemented in MC73) as a postprocessing routine.
`with_ipopt`	$\mathbb{B}$	If enabled, IPOPT controls the quality of the solution. Interaction with IPOPT is enabled to find an acceptable solution to do the next inexact optimization step.
`ipopt_inexact_iter`	$\mathbb{N}$	After this number of BiCGStab iterations the interaction starts with IPOPT.
`ipopt_inexact_tol`	$\mathbb{R}$	At this tolerance in PSPIKE the interaction starts with IPOPT.
`print_matrix`	$\mathbb{B}$	Enables printing of matrices in general.
`print_matrix_csr`	$\mathbb{B}$	Print all matrices in the CSR format.
`print_matrix_mtx`	$\mathbb{B}$	Print all matrices in the Matrix Market format.
`print_matrix_matlab`	$\mathbb{B}$	Print all matrices suitable for Matlab.

4.3 A Small Example

The linear equation system $\mathcal{A}\mathbf{x} = \mathbf{f}$ is solved where

$$\mathcal{A} = \begin{pmatrix} 9 & 6 & 0 & 3 & 0 & 2 \\ 0 & 2 & 7 & 0 & 1 & 0 \\ 5 & 4 & 0 & 0 & 0 & 3 \\ 0 & 6 & 8 & 3 & 4 & 0 \\ 8 & 0 & 4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 7 & 6 & 5 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} 20 \\ 10 \\ 12 \\ 21 \\ 13 \\ 18 \end{pmatrix} \text{ with the solution } \mathbf{x} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

The test program can be compiled using the GNU compiler suites with the following command
`mpif90 -O3 -o TestPspike TestPspike.f90 libpspike.so libfiedler.a libmc73.a libpatoh.a libpardiso.so libblas.so -fopenmp`

```
1   PROGRAM main
2
3       implicit none
4
5       include 'mpif.h'
6
7       ! Function parameter declaration
8       external PSPIKE
9
10      ! Linear equation system including an example
11      ! matrix stored in 1-based CSR format
12      integer :: neqns = 6
13      integer :: nnz = 20
14      integer :: ia(7)          = (/1,5,8,11,15,18,21/)
15      integer :: ja (20)        = (/1,2,4,6,2,3,5,1,2,6,
16                                   2,3,4,5,1,3,5,4,5,6/)
17      double precision :: a(20) = (/9,6,3,2,2,7,1,5,4,3,
18                                   6,8,3,4,8,4,1,7,6,5/)
19      ! right-hand side b and solution x
20      double precision :: b(6) = (/20,10,12,21,13,18/)
21      double precision :: x(6) = (/0,0,0,0,0,0/)
22
23      ! PSPIKE parameters
24      integer :: bandwidth
25      double precision :: bicgstab_tol
26      integer :: nrhs, info(2)
27
28      ! MPI Variables
29      integer error, nProcs, rank
30
31      ! MPI Settings
32      call MPI_INIT (error)
33      call MPI_COMM_SIZE (MPI_COMM_WORLD, nProcs, error)
34      call MPI_COMM_RANK (MPI_COMM_WORLD, rank, error)
35
36      !!! Setting the default parameter
37      bandwidth = 0
38      bicgstab_tol = 0
39      info = 0
```

```
40      nrhs = 1
41
42      !! Initialize PSPIKE (job = 0)
43      call PSPIKE(0, neqns, nnz, ia, ja, a, b, x,
44                  bandwidth, bicgstab_tol, nrhs, info)
45      !! Preprocessing PSPIKE (job = 1)
46      call PSPIKE(1, neqns, nnz, ia, ja, a, b, x,
47                  bandwidth, bicgstab_tol, nrhs, info)
48      !! Numerical Hybrid Factorization (job = 2)
49      call PSPIKE(2, neqns, nnz, ia, ja, a, b, x,
50                  bandwidth, bicgstab_tol, nrhs, info)
51      !! Solving System (job = 3)
52      call PSPIKE(3, neqns, nnz, ia, ja, a, b, x,
53                  bandwidth, bicgstab_tol, nrhs, info)
54      !! Destroy PSPIKE (job = 4)
55      call PSPIKE(4, neqns, nnz, ia, ja, a, b, x,
56                  bandwidth, bicgstab_tol, nrhs, info)
57
58      call MPI_FINALIZE(error)
59
60  END PROGRAM main
```

Listing 1: Example Fortran Program `TestPspike.f90`

The corresponding option file `pspike.opt` looks as follows:

```
1   output_file pspike.out
2   standalone_version 1
3   is_matrix_symmetric 0
4   matrix_stays_symmetric 0
5   bicgtab_tolerance 1e-10
6   bicgstab_max_iter 500
7   bicgstab_max_stagnation 20
8   bicgstab_inner_tol 1e-15
9   bicgstab_inner_iter 100
10  with_prec_bicgstab_inner 1
11  pspike_bandwidth 1
12  pspike_debug_level 2
13  pardiso_iparm_31 2
14  pardiso_32bit 0
15  pardiso_msglvl 0
16  pardiso_iter_ref 10
17  distribute_rowblockwise 1
18  with_fast_matvec 0
19  with_scaling 1
20  scaling_symmetric 0
21  with_matching_dualscaling 1
22  with_auction_dualscaling 0
23  with_rcm_reordering 0
24  with_mc64_matching 1
25  with_mc73_spectral 1
26  with_fill_couplings 1
27  with_sort_heuristic 3
28  with_metis_partitioner 0
29  with_auction_matching 0
30  with_tracemin_fiedler 0
```

```
31 with_mondriaan 0
32 with_ml_sloan 0
33 print_matrix_csr 0
34 print_matrix_mtx 0
35 print_matrix_matlab 0
36 print_matrix 0
37 with_ipopt 0
38 ipopt_inexact_iter 1
39 ipopt_inexact_tolerance 1e1
40 with_mc73_hager 0
41 fiedler_tolerance 1e-5
```

Listing 2: Options File `pspike.opt` for the `TestPspike.f90`

References

[1] Michael W. Berry and Ahmed Sameh. Multiprocessor schemes for solving block tridiagonal linear systems. *The International Journal of Supercomputer Applications*, 1(3):37–57, 1988.

[2] R. Bisseling. Mondriaan for sparse matrix partitioning. `http://www.staff.science.uu.nl/~bisse101/Mondriaan/`. Accessed January 2012.

[3] BLAS. Basic linear algebra subprograms. `http://netlib.org/blas/`. Accessed November 2011.

[4] Jack J. Dongarra and Ahmed H. Sameh. On some parallel banded system solvers. *Parallel Computing*, 1(3):223–235, 1984.

[5] M. Hagemann and O. Schenk. Weighted matchings for preconditioning symmetric indefinite linear systems. *SIAM Journal of Scientific Computing*, 28:403–420, 2006.

[6] HSL. The HSL mathematical software library. `http://www.hsl.rl.ac.uk/`. Accessed January 2012.

[7] G. Karypis. Family of graph and hypergraph partitioning software. `http://glaros.dtc.umn.edu/gkhome/views/metis`. Accessed January 2012.

[8] D H. Lawrie and A H. Sameh. The computation and communication complexity of a parallel banded system solver. *ACM Trans. Math. Softw.*, 10(2):185–195, 1984.

[9] M. Manguoglu. TraceMin-Fiedler. `http://www.cs.purdue.edu/homes/mmanguog/fiedler.html`. Accessed January 2012.

[10] Murat Manguoglu, Mehmet Koyutürk, Ahmed H. Sameh, and Ananth Grama. Weighted matrix ordering and parallel banded preconditioners for iterative linear system solvers. *SIAM J. Scientific Computing*, 32(3):1201–1216, 2010.

[11] Murat Manguoglu, Ahmed Sameh, and Olaf Schenk. A parallel hybrid sparse linear system solver. *LNCS - Proceedings of EURO-PAR09*, 5704:797–808, 2009.

[12] Intel Software Network. Intel math kernel library (mkl). `http://software.intel.com/en-us/articles/intel-mkl/`. Accessed January 2012.

[13] LAPACK Linear Algebra PACKage. `http://www.netlib.org/lapack/`. Accessed November 2011.

[14] Eric Polizzi and Ahmed H. Sameh. A parallel hybrid banded system solver: the spike algorithm. *Parallel Computing*, 32(2):177–194, 2006.

[15] Eric Polizzi and Ahmed H. Sameh. Spike: A parallel environment for solving banded linear systems. *Computers & Fluids*, 36(1):113–120, 2007.

[16] D. J. Kuck S. C. Chen and A. H. Sameh. Practical parallel band triangular system solvers. *ACM Transactions on Mathematical Software*, 4(3):270–277, 1978.

[17] A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *J. ACM*, 25(1):81–91, 1978.

[18] M. Sathe, O. Schenk, and H. Burkhart. An auction-based weighted matching implementation on massively parallel architectures. *Parallel Computing*, 2012. accepted for publication.

[19] M. Sathe, O. Schenk, B. Ucar, and A. Sameh. Towards a scalable hybrid linear solver based on combinatorial algorithms. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, pages 96–127. Chapman-Hall CRC Computational Science, 2012.

[20] O. Schenk and K. Gärtner. PARDISO solver project. `http://www.pardiso-project.org/`. Accessed January 2012.

[21] O. Schenk, M. Manguoglu, A. Sameh, M. Christian, and M. Sathe. Parallel scalable PDE-constrained optimization: antenna identification in hyperthermia cancer treatment planning. *Computer Science Research and Development*, 23(3-4), 2009.