RAL-TR-98-039

PARASOL
An Integrated Programming Environment for Parallel Sparse Matrix Solvers[1]

Patrick Amestoy[2], Iain Duff[3], Jean Yves L'Excellent[4] and Petr Plecháč[5]

ABSTRACT

PARASOL is an ESPRIT IV Long Term Research Project whose main goal is to build and test a portable library for solving large sparse systems of equations on distributed memory systems. There are twelve partners in five countries, five of whom are code developers and five end users. The software is written in Fortran 90 and uses MPI for message passing. There are routines for both direct and iterative solution of symmetric and unsymmetric systems. The final library will be in the public domain.

We will discuss the PARASOL Project with particular emphasis on the algorithms and software for direct solution that are being developed by RAL and CERFACS in collaboration with ENSEEIHT-IRIT in Toulouse. The underlying algorithm is a multifrontal one with a switch to ScaLAPACK processing towards the end of the factorization (and solution). We will discuss the algorithms, the interface, and their current status and illustrate the performance of the direct solver on a range of problems from the PARASOL end users.

Keywords: MPI, distributed memory architecture, sparse matrices, multifrontal direct methods.

AMS(MOS) subject classifications: 65F05, 65F50.

---

[1]Support from the EU ESPRIT IV LTR Project 20160 is gratefully acknowledged. This report is a preprint of an article that will appear in the Proceedings of the HPCI Conference 1998 that was held in Manchester, England in January 1998.

[2] amestoy@enseeiht.fr, [3] I.S.Duff@rl.ac.uk, [4] excelle@cerfacs.fr, [5] P.Plechac@rl.ac.uk.

Current reports available by anonymous ftp from matisa.cc.rl.ac.uk in the directory "pub/reports". This report is in file hpci98RAL98039.ps.gz. Also published as Technical Report TR/PA/98/13 from CERFACS.

Department for Computation and Information
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

May 6, 1998.

Contents

1 Introduction

PARASOL is a Long Term Research (LTR) ESPRIT IV Project (No 20160) for "An Integrated Environment for Parallel Sparse Matrix Solvers". This Project started on January 1st, 1996, and its aim is to develop a parallel scalable library of sparse matrix solvers using Fortran 90 and MPI. At the end of the Project, the codes will be made available in the public domain.

The PARASOL Consortium is managed by PALLAS in Germany and consists of

- leading European research organizations with internationally recognized experience and an established track record in the development of parallel solvers (CERFACS, GMD-SCAI, ONERA, Rutherford Appleton Laboratory (RAL), University of Bergen);
- industrial code developers who define the requirements for PARASOL, are providing test cases generated by their finite-element packages, and will use the developed software in production mode (Apex Technologies, Det Norske Veritas (DNV), INPRO, MacNeal-Schwendler (MSC), Polyflow);
- two leading European HPC software companies who will exploit the project results and are providing state-of-the-art programming development tools (GENIAS, PALLAS).

For more information, see the project web site at `http://www.genias.de/parasol`.

The codes in the Library include direct methods, domain decomposition techniques, and multigrid algorithms. Within this project, RAL and CERFACS are involved in the development of direct solvers and are working in this context in close collaboration with ENSEEIHT-IRIT, Toulouse, France. The codes are implemented as portable prototypes integrated into the PARASOL Library. The four different solvers and their implementation within the PARASOL Library are depicted in Figure 1.1.

The Library provides its own communication and data exchange routines (the PARASOL Interface) as a higher level message passing protocol based on MPI and designed for specific data structures that arise in the finite-element approximation of partial differential equations and operations with sparse matrices. In this short paper, we focus on the sparse direct solver (MUMPS) developed by RAL and CERFACS in collaboration with ENSEEIHT-IRIT (see Amestoy, Duff and L'Excellent (1998) for more details). We first give a brief overview of the PARASOL Interface Library (see Supalov (1998)) emphasizing aspects relevant to the implementation of MUMPS in Section 2. We then briefly describe the algorithm used by the MUMPS package in Section 3.

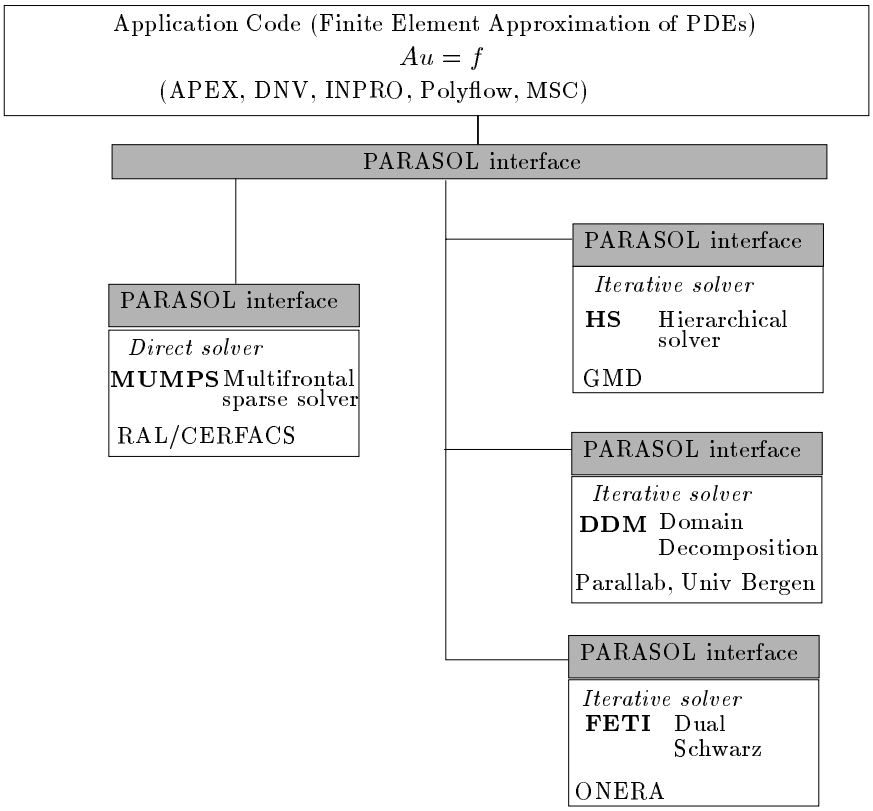

Figure 1.1: PARASOL Library

The PARASOL end users are providing several examples for the solver developers to use as a test set. The test cases range from quite small problems (around 10000 equations) to large problems with over 2.5 million degrees of freedom. We consider the performance of the MUMPS package on some of these test cases in Section 4, before commenting on planned future work in Section 5.

2 The PARASOL Interface

Several very different solvers and algorithms for the solution of large sparse linear systems are integrated within the PARASOL Library. The Project has designed and uses a higher level message passing protocol in order to overcome difficulties arising from the different data structures typically used by the different algorithms. These difficulties are exacerbated by a distributed memory environment. The package uses the MPI 1.1 message passing interface (Message Passing Interface Forum 1995) as the basis for its own interface standard. The PARASOL Interface allows users (and solver developers) to exchange data between different modules of the package in

a precisely defined way that is independent of the specific data structures needed internally in the user's codes or solvers.

The PARASOL Interface accommodates three modes of parallel execution:

- Host-node - one process inputs the problem definition and the others exchange data with it and compute the solution,
- Hybrid-host - one process inputs the problem definition and exchanges data with the others, but then helps the others to compute the solution, and
- Hybrid-node - all processes input the problem definition and compute the solution.

Currently only the Host-node mode is efficiently implemented by the MUMPS solver.

It is the responsibility of the user to define, configure and terminate the parallel environment in addition to controlling I/O resources. However, the PARASOL Library offers tools for controlling synchronization of data exchanges. The basic process within the PARASOL environment is called *an instance* and is described by its descriptor, which also encodes an MPI communicator used for data exchanges related to a given instance. Every PARASOL routine is passed an instance descriptor as an argument and then performs operations only on data associated with this instance. The PARASOL Library allows the user to run several PARASOL instances simultaneously. The instance operates on its own set of private data structures, although it can initialize and use another instance. This feature provides an easy mechanism for integrating different solvers from the Library into a new solver. For example, the domain decomposition solver can call the sparse direct solver. This new solver will use all parallel features of both solvers, without any changes of data structures.

The PARASOL Interface guide (Supalov, 1998) gives a detailed description of the data exchange protocol and data structures defined for PARASOL routines. Here we restrict ourselves to data structures relevant for the current version of MUMPS, which only handles assembled sparse matrices. We will not discuss the implementation of data protocols related to structures such as finite-element meshes, element matrix format etc.

Sparse matrices are represented within the PARASOL Library in the compressed column storage format, that is, for an $m \times n$ sparse matrix $A = (a_{ij})$ we have a descriptor which contains the data shown in Table 2.1.

The matrix is stored in associated arrays:

`col` - stores pointers to the start of the column data (entries in a column) in `row` and `val`. The last entry of `col` points to the first free entry in `row`, `val`. The length of `col` is equal to `PSL_NCOL + 1`.

`row` - stores row indices of entries for each column. The length of `row` is equal to `PSL_NVAL + 1`.

PSL_TYPE	MPI_DOUBLEPRECISION ...	MPI data type
PSL_NAME	PSL_MATRIX	data name
PSL_ATTR	PSL_SYMMETRIC,PSL_HERMITIAN,PSL_UNSYMMETRIC	matrix structure
PSL_FORM	PSL_SPARSEMAT	sparse matrix format
PSL_NROW	INTEGER $m > 0$	number of rows
PSL_NCOL	INTEGER $n > 0$	number of columns
PSL_NVAL	INTEGER $ne > 0$	number of entries

Table 2.1: Sparse matrix descriptor

`val` - stores values of entries for each column. The length of `val` is equal to `PSL_NVAL + 1`.

The right-hand side(s) are entered either in the sparse vector format, which has the same descriptor as a sparse matrix with the following fields modified: `PSL_NAME = PSL_RHSIDE or PSL_SOLUTION`, `PSL_ATTR = PSL_LEFT or PSL_RIGHT` (vector position), `PSL_FORM = PSL_SPARSEVEC`, or in the dense vector format (see Table 2.2). In the case of dense vectors, the arrays `col`, `row` are not used and the vectors are stored in `val(1:PSL_LDIM,1:PSL_NVEC)`.

PSL_TYPE	MPI_DOUBLEPRECISION	MPI data type
PSL_NAME	PSL_RHSIDE,PSL_SOLUTION,...	data name
PSL_ATTR	PSL_LEFT,PSL_RIGHT	vector position
PSL_FORM	PSL_DENSEVEC	dense vector format
PSL_NROW	INTEGER > 0	number of rows
PSL_NVEC	INTEGER > 0	number of vectors
PSL_LDIM	INTEGER > 0	leading dimension of `val`

Table 2.2: Dense vector descriptor

The PARASOL Library provides the user with easy access to the solvers via control routines for each solver and data exchange routines related to the transfer of data structures relevant to the given solver.

We illustrate the use of the PARASOL Library for the MUMPS solver implemented within a simple code which reads a sparse matrix from an external file, maps the solver onto a given number of processors, and performs the solution. The skeleton for this is shown in Figure 2.1.

The generic calls to `psl_init` and the other `psl_` calls are interpreted by the PARASOL Interface, through the defined configuration name, to be calls to the PARASOL control routines for the MUMPS solver.

Unlike the exchange routines, the control routines make heavy use of the internal structure of the MUMPS code. Their implementation is based on appropriate calls

to the subroutine psl_mumps. We use the name "MUMPS code" to refer to the whole set of routines for which the routine psl_mumps serves as a driver.

- psl_mumps_init - this routine initializes the MUMPS instance:
  1. creating a description of nodes, the current version efficiently supports PSL_MODE = PSL_HOSTNODE and the user has reserved one process (rank = 0),
  2. allocating private data structures for the instance, and
  3. calling the initialization phase of the MUMPS code,
- psl_mumps_end - this routine terminates the instance:
  1. deallocating the resources,
- psl_mumps_map - the mapping routine performs the analysis and factorization phases of the MUMPS code:
  1. setting output/diagnostics parameters,
  2. receiving the sparse structure of the matrix (n,ne,col,row) on the master,
  3. performing the analysis phase of the MUMPS code,
  4. receiving numerical values for the matrix entries (val) on the master, and
  5. performing the numerical factorization,
- psl_mumps_solve - the solution routine calls the solution phase of the MUMPS code:
  1. receiving the right-hand side(s), and
  2. solving,
- psl_mumps_endsolve - the termination routine sends the solution to the user and deallocates the memory.

The data exchange routines of the form host_ in Figure 2.1 must be coded by the user and should include calls to the following data exchange routines that support data communication needed between the user's code and the MUMPS solver:

- psl_mumps_contract - establishing a data exchange session,
- psl_mumps_what2send - send an inquiry,
- psl_mumps_sendIdata - send INTEGER data (col, row),
- psl_mumps_sendDdata - send DOUBLE PRECISION data (val),

- `psl_mumps_need2recv` - request data,
- `psl_mumps_recvIdata` - receive `INTEGER` data, and
- `psl_mumps_recvDdata` - receive `DOUBLE PRECISION` data.

These routines are used only for data exchange between a user's application (program) and the MUMPS solver (package). The basic data communication for MUMPS requires: sending a matrix (`PSL_NAME = PSL_MATRIX`), and a right-hand side vector (`PSL_NAME = PSL_RHSIDE`), and receiving a solution vector (`PSL_NAME = PSL_SOLUTION`).

The current version (Version 2.0) of the MUMPS integration into the PARASOL package supports the `PSL_HOSTNODE` and `PSL_HYBRIDHOST` execution models, although only the former fully benefits from the parallel features of the MUMPS Version 2.0 code. The execution mode `PSL_HOSTNODE` is schematically described in Figure 2.2. In this mode, one processor is dedicated to serve as a master process for PARASOL and the remaining processors are used for the parallel implementation of the solver.

3 MUMPS - MUltifrontal Massively Parallel Solver

MUMPS is a parallel sparse direct solver for distributed memory architectures using a multifrontal method, which is a direct method based on the LU factorization of the matrix. We refer the reader to our earlier papers (Amestoy and Duff 1989, Duff and Reid 1983, Duff and Reid 1984) for full details of this technique. The current version of MUMPS (Version 2.0) solves the system

$$\mathbf{A}\mathbf{x} = \mathbf{b},$$

where $\mathbf{A}$ is assembled and unsymmetric.

The structure of the matrix is first *analysed* to determine an ordering that, in the absence of any numerical pivoting, will preserve sparsity in the factors. In Version 2.0 of MUMPS, an approximate minimum degree ordering strategy is used on the symmetrized pattern $\mathbf{A}+\mathbf{A}^T$, and this analysis phase produces both an ordering and an assembly tree. The assembly tree is then used to drive the subsequent numerical factorization and solution phases. At each node of the tree, a dense submatrix (called a *frontal matrix*) is assembled using data from the original matrix and from the sons of the node. Pivots can be chosen from within a submatrix of the frontal matrix (called the *pivot block*) and eliminations performed. The resulting factors are stored for use in the solution phase and the Schur complement (the *contribution block*) is passed to the father node for assembly at that node. In the numerical factorization phase, the tree is processed from the leaf nodes to the root

```
      PROGRAM example
! Include files
      INCLUDE 'mpif.h'       ! MPI definitions
      INCLUDE 'pslf.h'       ! PARASOL definitions
! Local data
      INTEGER self           ! MPI rank of the current process
      INTEGER id(PSL_IDSIZE) ! PARASOL instance descriptor
      INTEGER rc             ! return code

      CALL MPI_INIT(rc)      ! initialize MPI
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,self,rc)    ! get MPI rank

      id = 0                         ! clear the instance desc.
      id(PSL_COMM) = MPI_COMM_WORLD ! use MPI communicator
      id(PSL_CONF) = PSL_MUMPS      ! name the configuration
      id(PSL_MODE) = PSL_HOSTNODE   ! select execution mode

      CALL psl_init(id,rc)    ! initialize PARASOL instance

      CALL psl_map(id,rc)     ! nodes compute mapping/reordering
      IF( self.EQ.0) THEN
        CALL host_serves_mapping_data_ex(....id,rc) ! reading data from a file
      END IF
      CALL psl_solve(id,rc)   ! nodes solve the system
      IF( self.EQ.0 ) THEN
        CALL host_serves_solution_data_ex(....id,rc) ! collecting the solution
        CALL host_outputs_data(....id,rc)  ! printing the solution
      END IF

      CALL psl_end(id,rc)     ! terminate PARASOL instance
      CALL MPI_FINALIZE(rc)   ! terminate MPI
      END
```

Figure 2.1: Skeleton of PARASOL Test Driver for MUMPS code

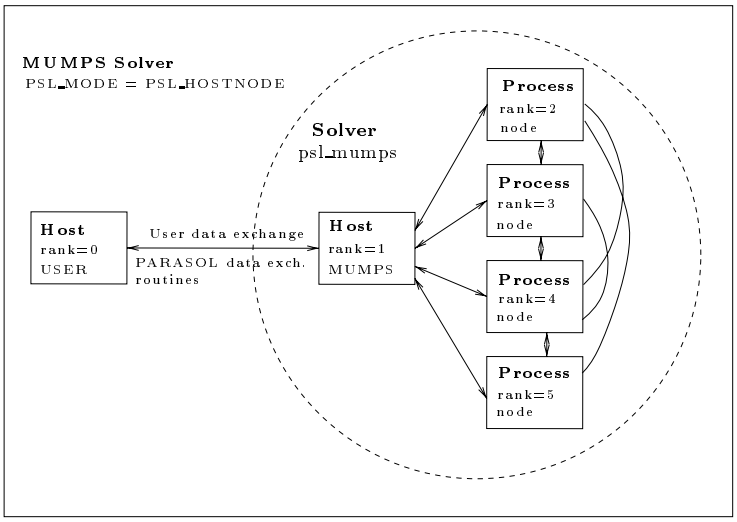

Figure 2.2: `HOST_NODE` execution model for MUMPS

(if the matrix is reducible, we have a forest, and each component tree of the forest will be treated similarly and independently). The subsequent forward and backward substitutions during the solution phase process the tree from the leaves to the root and from the root to the leaves, respectively. A crucial aspect of the assembly tree is that it defines only a partial order for the factorization since the only requirement is that a son must complete its elimination operations before the father can be fully processed. It is this freedom that enables us to exploit parallelism in the tree (*tree parallelism*).

In the unsymmetric case, threshold pivoting is used to maintain numerical stability so that it is possible that the pivots selected at the analysis phase are unsuitable. In the numerical factorization phase, we are at liberty to choose pivots from anywhere within the pivot block (including off-diagonal pivots) but it still may be impossible to eliminate all variables from this block. The result is that the Schur complement that is passed to the father node may be larger than anticipated by the analysis phase and so our data structures may be different from those forecast by the analysis. This implies that we need to allow dynamic scheduling during numerical factorization, in contrast to the symmetric positive definite case where only static scheduling is required.

A version of the multifrontal code for shared memory computers was developed by Amestoy and Duff (1989) and was included in Release 12 of the Harwell Subroutine Library (HSL, 1996) as code `MA41`. This was the basis for Version 1.0 of MUMPS that was released in May 1997.

In the current version of MUMPS (Version 2.0), both tree and node parallelism are exploited, and we distribute the pool of work among the processors, but our model still requires an identified host node to perform the analysis phase, distribute

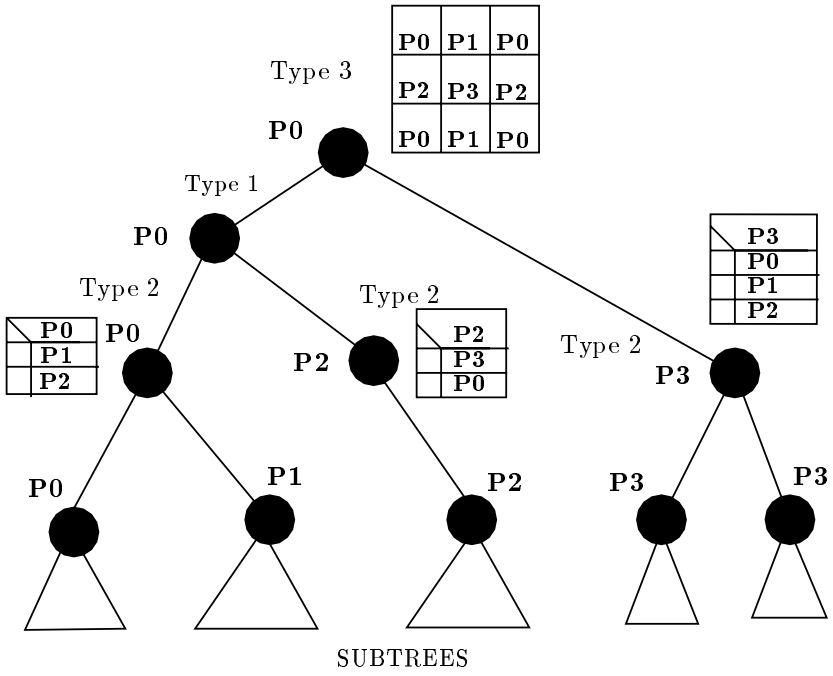

Figure 3.1: Distribution of the computations of a multifrontal tree

the incoming matrix, collect the solution, and generally oversee the computation. All routines called by the user for the different steps are SPMD, and the distinction between the host and the other processors is made by the MUMPS code. The code is organized with a designated host node and other processors as follows (notice that the following steps are easily implemented within the controlling strategy of the PARASOL Library):

1. Analysis. The host performs an approximate minimum degree algorithm based on the symmetrized pattern $\mathbf{A}+\mathbf{A}^T$, and carries out symbolic factorization. A mapping of the multifrontal tree is then computed, and symbolic information is transferred from the host to the other processors. Using this information, the processors estimate the memory necessary for factorization/solution.

2. Factorization. The host sends appropriate entries of the original matrix to the other processors that are responsible for the numerical factorization. The numerical factorization on each frontal matrix is conducted by a *master* processor (determined by the analysis phase) and one or more *slave* processors (determined dynamically) as discussed later in this section. Each processor allocates an array for contribution blocks and factors; the latter should be kept for the solution.

3. Solution. The right-hand side is broadcast from the host to the other

processors. These processors compute the solution using the (distributed) factors computed during Step 2, and the solution is assembled on the host.

For an efficient and more scalable parallelism on general matrices, the elimination of frontal matrices near the root of the tree has to be parallelized. Version 2.0 of MUMPS exploits both tree parallelism and node parallelism; this is done by introducing Type 2 and Type 3 nodes, as defined below.

We consider the assembly tree of Figure 3.1 where, instead of single nodes as the leaves, there are subtrees whose constituent nodes have frontal matrices of small order. Each subtree is processed by a single processor, to avoid communication at that stage. This mapping of subtrees to processors is performed by the analysis phase. For large problems, there will be more subtrees than processors which will aid in the overall load balancing of the computation.

Above the subtrees, there can still be some nodes processed by only one processor. These nodes (as well as nodes inside the subtrees) are called nodes of Type 1.

Consider a typical frontal matrix in the tree in which there are `NPIV` pivots to eliminate (that is, the pivot block has order `NPIV`) and `NCB` rows to update (that is, the order of the frontal matrix is `NPIV+NCB`). A node is of Type 2 if `NCB` is large enough (the default in the code is that `NCB` should be larger than 800). The partial factorization process is then parallelized with the first `NPIV` rows on one processor, called the *master of the node* and the `NCB` rows distributed among other processors (called the *slaves of the node*). For instance, in the Type 2 node on the right of Figure 3.1, `P3` is the master, and `P0`, `P1`, and `P2` are the slaves.

A pipelined factorization is used, and updates to the contribution blocks are performed in parallel. In our implementation, the assembly process is also fully parallel.

At the root node, a full LU factorization is performed. If the size of the root node is deemed large enough, the root node is said to be of Type 3, and is factorized using ScaLAPACK (Blackford, Choi, Cleary, D'Azevedo, Demmel, Dhillon, Dongarra, Hammarling, Henry, Petitet, Stanley, Walker and Whaley 1997). The assembly of the root node is directly distributed in a 2D cyclic grid and is completely parallel.

4 Results

In this section, we present results to demonstrate the performance of the MUMPS code on a few of the set of PARASOL test examples. We show the performance of MUMPS with different levels of parallelism as well as the speedup obtained on an IBM SP2.

The test cases are summarized in Table 4.1 where N, NE denote the size of the matrix and the number of entries, respectively. The size of the LU factors is reported in the column "LU-Fac", and the number of floating-point operations to factorize the matrix is given in the last column.

The results in Table 4.2 show that there is often good speedup on the test examples, indeed sometimes the performance is superlinear. This is caused by the reduction in memory requirements on individual processors with a consequent reduction in paging overheads.

Problem	Origin	Type	N	NE $\times 10^6$	LU-Fac $\times 10^6$	Ops $\times 10^9$
INV_EXTRUSION-1	POLYFLOW	U	30412	1.79	49.7	36
OILPAN	INPRO	S	73752	1.84	20.6	8
MIXING-TANK	POLYFLOW	U	29957	1.99	62.7	142
CRANKSEG1	MSC	S	52804	10.6	80.1	101
BMW7ST_1	MSC	S	141347	3.74	54.0	31
WANG3	RBSMC	U	26064	0.2	11.5	11

Table 4.1: The test matrices used in numerical experiments.

	Analysis	LU Factorization				
Number processors	1	2	4	8	16	32
Test case						
INV_EXTRUSION-1	5.2		536.7	179.1	72.7	69.1
OILPAN	4.4	133.7	25.9	22.0	21.0	
MIXING-TANK	4.4			607.6	83.8	80.1
CRANKSEG1	36.0				626.0	
CRANKSEG1 [a]	9.1			1251.4	1022.8	899.6
Number processors	1	8	12	16	20	24
BMW7ST_1	9.7	118.9	60.8	43.8	44.1	38.7

Table 4.2: Times for Analysis and Factorization in seconds. Results obtained on the IBM SP2 at GMD, Bonn. [a]Computed on SGI Origin at Parallab, Bergen.

Table 4.3 compares MUMPS with the symmetric code WSSMP (Gupta, Joshi and Kumar 1997). As MUMPS is an unsymmetric solver, the symmetric matrix is expanded into full memory storage format and subsequent operations are performed on this expanded form. Nonetheless, MUMPS exhibits a comparable performance with the well-tuned symmetric solver.

The importance of different levels of parallelism is demonstrated on an example of an unsymmetric matrix, WANG3, from the collection of sparse matrices by Davis (1997). CPU times for the numerical factorization are shown in Table 4.4. The times for only using tree parallelism are in the column headed by L1, while L2 denotes that nodes of Type 2 are treated in parallel. With L3, the root node is assembled and factorized in parallel using ScaLAPACK.

	WSSMP	MUMPS 2.0
Analysis	500	10.2
Matrix redistribution		11.8
Factorization	22.1	43.8
Triangular solution	1.14	1.6

Table 4.3: Comparison with WSSMP for the test example BMW7ST_1 on 16 processors of the IBM SP2 at GMD (thin nodes with 128 MB of physical memory and 512MB of virtual memory). Times are in seconds.

No. procs	L1	L1 + L2	L1 + L2 + L3	
			Time	Speed-up
Seq CPU Time	71.0	71.0	71.0	1
2	61.3	89.3		
3	96.6	79.5		
4	46.9	65.7	77.9	
5	49.1	33.3	23.7	3.0
6	46.4	31.9	22.4	3.2
7	45.4	30.1	20.5	3.5
8	44.1	27.9	20.7	3.4

Table 4.4: Comparison of different levels of parallelism

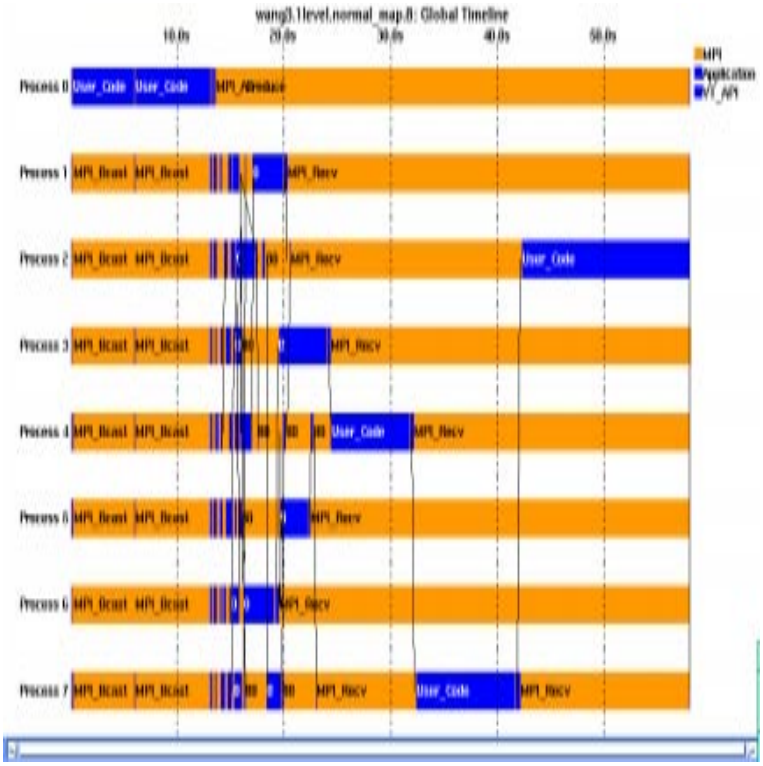

Figure 4.1: Output from the VAMPIR log of the MUMPS run with only tree parallelism (L1).

The tuning of the code and also the influence of different levels of parallelism can be traced and visualized using VAMPIR, which is a tool developed by one of the PARASOL project partners (PALLAS Gmbh). The two figures in Figure 4.1 and Figure 4.2 show communication and load balancing, with the vertical lines indicating messages being passed and the dark shaded regions work being performed on processors. Although it may be difficult to see too much from these figures without prior experience with the tool, the poor parallelism from using only tree parallelism is seen in the left-hand figure while the right-hand figure shows more communication due to the L2 and L3 levels, the L3 level being the denser part at the right of the figure. The elapsed time is of course much reduced, as can been seen from the times along the top edges of the figures.

5 Conclusions and future work

We have briefly described the current interface to the PARASOL Library and have discussed the direct solver available in that Library. We have illustrated that the current version performs well, exhibits a good degree of parallelism, and is competitive with a vendor supplied direct solver.

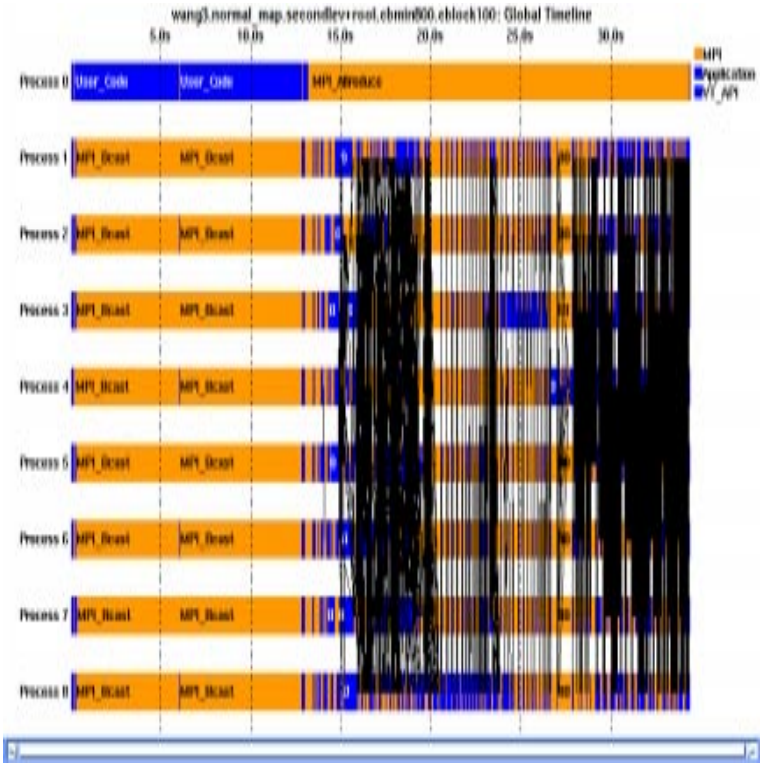

Figure 4.2: Output from the VAMPIR log of the MUMPS run with all levels of parallelism (L1+L2+L3).

At a recent Project Review meeting, there was support for an extension of the PARASOL Project to June 30, 1999. Further enhancements to the MUMPS code that will be undertaken during this period include: a version for symmetric systems, an element entry, more sophisticated ordering and mapping strategies, integration with the other codes in the PARASOL package so that the direct solver can be called from within the iterative solvers, and more rigorous testing and tuning.

Acknowledgment

We would like to thank Jennifer Scott of the Rutherford Appleton Laboratory for helpful comments on an earlier draft of the paper.

References

Supalov, A. (Editor) (1998), PARASOL Interface Specification. Version 2.1, January 9th, 1998.

Amestoy, P. R. and Duff, I. S. (1989), 'Vectorization of a multiprocessor multifrontal code', *Int. J. of Supercomputer Applics.* **3**, 41–59.

Amestoy, P. R., Duff, I. S. and L'Excellent, J.-Y. (1998), MUMPS MUltifrontal Massively Parallel Solver, Version 2.0, Technical Report TR/PA/98/02, CERFACS.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C. (1997), *ScaLAPACK Users' Guide*, SIAM Press.

Davis, T. A. (1997), 'University of Florida sparse matrix collection', Available at http://www.cise.ufl.edu/~davis and ftp://ftp.cise.ufl.edu/pub/faculty/davis.

Duff, I. S. and Reid, J. K. (1983), 'The multifrontal solution of indefinite sparse symmetric linear systems', *ACM Trans. Math. Softw.* **9**, 302–325.

Duff, I. S. and Reid, J. K. (1984), 'The multifrontal solution of unsymmetric sets of linear systems', *SIAM J. Scientific and Statistical Computing* **5**, 633–641.

Gupta, A., Joshi, M. and Kumar, V. (1997), WSSMP: Watson Symmetric Sparse Matrix Package. Users Manual: Version 2.0β, Technical Report RC 20923 (92669), IBM T. J. Watson Research Centre, P. O. Box 218, Yorktown Heights, NY 10598.

HSL (1996), *Harwell Subroutine Library. A Catalogue of Subroutines (Release 12)*, AEA Technology, Harwell Laboratory, Oxfordshire, England. For information concerning HSL contact: Dr Scott Roberts, AEA Technology, 552 Harwell, Didcot, Oxon OX11 0RA, England (tel: +44-1235-434988, fax: +44-1235-434136, email: Scott.Roberts@aeat.co.uk).

Message Passing Interface Forum (1995), MPI: A Message Passing Interface Standard Version 1.1, Technical report.