Efficient massively parallel implementation of the ReLPM exponential integrator for advection-diffusion models[*][†]

A. Martínez[¶]   L. Bergamaschi[‡]   M. Caliari[¶]   M. Vianello[¶]

February 5, 2007

Abstract

This work considers the Real Leja Points Method (ReLPM), [*J. Comput. Appl. Math.*, 172 (2004), pp. 79–99], for the exponential integration of large-scale sparse systems of ODEs, generated by Finite Element or Finite Difference discretizations of 3-D advection-diffusion models. We present an efficient parallel implementation of ReLPM for polynomial interpolation of the matrix exponential propagators $\exp(\Delta t A)\,\mathbf{v}$ and $\varphi(\Delta t A)\,\mathbf{v}$, $\varphi(z) = (\exp(z) - 1)/z$. A scalability analysis of the most important computational kernel inside the code, the parallel sparse matrix-vector product has been performed as well as an experimental study of the communication overhead. As a result of this study an optimized parallel sparse matrix-vector product routine has been implemented. The resulting code shows good scaling behavior even when using more than one thousand processors.

Keywords: Advection-diffusion equation: Sparse matrix; Exponential integrator; Parallel computation; Terascaling

1 Introduction

Three-dimensional advection-diffusion equations describe the transport of solutes in groundwater and surface water, the displacement of oil by fluid injection in oil recovery, the movement of aerosols and trace gases in the atmosphere, and miscible fluid flow processes in many other applications. In industrial applications, these equations are commonly discretized via the Finite Difference (FD) or the Finite Element (FE) methods.

We consider the classical evolutionary 3-D advection-diffusion problem on a domain Ω with mixed Dirichlet and Neumann boundary conditions:

$$
\begin{cases}
\dfrac{\partial c}{\partial t} = \operatorname{div}(D\nabla c) - \operatorname{div}(c\,\vec{v}) + \phi & x \in \Omega, \quad t > 0 \\[4pt]
c(x,0) = c_0(x), \ x \in \Omega; \\[2pt]
c(x,t) = g_{\mathrm{D}}(x,t), \ x \in \Gamma_{\mathrm{D}}; \\[2pt]
D\nabla c(x,t) \cdot \vec{\nu} = g_{\mathrm{N}}(x,t), \ x \in \Gamma_{\mathrm{N}}; & t > 0
\end{cases}
\tag{1}
$$

where $\Gamma_D \cap \Gamma_N = \partial\Omega$. Equation (1) represents, e.g., a simplified model for solute transport in groundwater flow (advection-dispersion), where c is the solute concentration, D the hydrodynamic dispersion tensor, $\vec{v} = (v_1, v_2, v_3)^T$ the average linear velocity of groundwater flow, $\vec{\nu}$ the outward normal and ϕ the source.

FE Discretization. The standard Galerkin FE discretization of (1) with nodes $\{x_i\}_{i=1}^N$ and linear basis functions gives a large scale linear system of ODEs like

$$\begin{cases} P\dot{\mathbf{c}} = H\mathbf{c} + \mathbf{b}, & t > 0 \\ \mathbf{c}(0) = \mathbf{c}_0 \end{cases} \tag{2}$$

where $\mathbf{c} = [c(x_1, t), \dots, c(x_N, t)]^T$, $\mathbf{c}_0 = [c(x_1, 0), \dots, c(x_N, 0)]^T$, P is the symmetric positive-definite mass matrix and H the (possibly nonsymmetric) stiffness matrix. Boundary conditions are incorporated in the matrix formulation (2) in the standard ways.

In the sequel we consider stationary velocity, source and boundary conditions in (1), which give constant H and $\mathbf{b}$ in system (2), which is the discrete approximation of the PDE (1). As known [7], the solution can be written explicitly in the exponential form

$$\mathbf{c}(t) = \mathbf{c}_0 + t\varphi(tP^{-1}H)\left[P^{-1}H\mathbf{c}_0 + P^{-1}\mathbf{b}\right], \tag{3}$$

where $\varphi(z)$ is the entire function

$$\varphi(z) = \frac{e^z - 1}{z} \quad \text{if } z \neq 0, \quad \varphi(0) = 1. \tag{4}$$

Clearly, availability of matrix P^{-1} is a computationally expensive task. Applying the well known mass-lumping technique (sum on the diagonal of all the row elements) to P, we obtain a diagonal mass matrix P_L. Now system (3) (with P_L replacing P) can be solved by the exact and explicit exponential time-marching scheme.

$$\mathbf{c}_{j+1} = \mathbf{c}_j + \Delta t_j \varphi(\Delta t_j A)\mathbf{v}_j, \quad \mathbf{v}_j = A\mathbf{c}_j + P_L^{-1}\mathbf{b}, \qquad j = 0, 1, \dots, \tag{5}$$

where $A = P_L^{-1}H$.

FD Discretization. Discretization of standard 7-point FD of eq. (1) gives raise to a system of ODEs like (2) which in its turn can be solved by the scheme (5) with A the classical eptadiagonal FD matrix and $\mathbf{v}_j = A\mathbf{c}_j + \mathbf{b}$.

Exactness of the exponential integrator (5) entails that the time-steps Δt_j can be chosen, at least in principle, arbitrarily large with no loss of accuracy, making it an appealing alternative to classical time-differencing integrators (cf. [7, 9, 13]). The practical application of (5) rests on the possibility of approximating efficiently the exponential propagator $\varphi(\Delta t A)\mathbf{v}$, where $\mathbf{v} \in \mathbb{R}^N$. To this aim, two classes of polynomial methods are currently used. We have Krylov-like methods, which are based on the idea of projecting the propagator on a "small" Krylov subspace of the matrix via the Arnoldi process, and typically involve long-term recurrences in the nonsymmetric case; see, e.g., [12, 23, 24]. These methods have been successfully used in solving e.g. the time-dependent Schrödinger equation [26]. The second class consists of methods based on polynomial interpolation or series expansion of the entire function φ on a suitable compact subset containing the spectrum (or in general the field of values) of the matrix (e.g. Faber and Chebyshev series, interpolation at special points like Faber, Fejér and Leja points) [4, 7, 19, 20]. They are conceived for approximating directly the matrix function, and typically require some preliminary estimate of the underlying spectral structure, but,

despite of this, these methods turned out to be competitive with Krylov-based approaches, especially on very large nonsymmetric matrices, cf. [4, 3].

Among these methods we consider the Real Leja Points Method (shortly ReLPM), recently proposed in the framework of spatial discretization of advection-diffusion equations [7]. In [3] the ReLPM code has been compared, in a sequential environment, with the PHIPRO Fortran code by Y. Saad, which is based on Krylov subspace approximations (cf. [22]), on large scale sparse matrices arising from the spatial discretization of 2-D and 3-D advection-diffusion equations. In all the experiments reported in [3] the ReLPM outperformed PHIPRO even with an optimal choice of the Krylov subspace dimension m. Comparisons with the classical Crank Nicolson method [10] with variable stepsize in solving realistic advection-diffusion problems has been performed both in sequential [5] and parallel [2] environments. In [2] a parallel implementation of ReLPM is described and results on two parallel machines are presented which show that ReLPM turns out to be from 4 to 7 times faster than Crank Nicolson, for a given accuracy.

We note also that ReLPM can be regarded as an exponential propagator and it can be successfully used in combination with other well-known integrators in the solution of large-scale nonlinear problems [8, 11, 14, 15, 27].

In this work we present a massively parallel implementation of the ReLPM algorithm obtained after a performance study and optimization of the parallel code presented in [2]. With the aim of attaining terascale performance, we have accomplished both performance evaluation and scalability analysis of the the most important computational kernel inside the code, the sparse-matrix times dense-vector product routine.

The paper is organized as follows. In section 2 we recall the ReLPM Algorithm and report comparisons between ReLPM and PHIPRO. Section 3 overviews the parallel implementation, addressing primarily the data distribution scheme and the implementation of the sparse matrix times dense vector product routine. Section 4 contains the parallel experiments for a variety of test problems. Speedup and parallel efficiency data are provided and analyzed. Finally, some concluding remarks are included in section 5.

2   The ReLPM Algorithm

The ReLPM (Real Leja Points Method), proposed in [7] and applied to advection-diffusion models in [4, 2], has shown very attractive computational features. It rests on Newton interpolation of the exponential functions at a sequence of Leja points on the real focal interval of a family of confocal ellipses in the complex plane.

Sequences of Leja points $\{z_j\}_{j=0}^{\infty}$ for the compact K are defined recursively as follows [16]: if z_0 is an arbitrary fixed point in K (usually such as $|z_0| = \max_{z \in K} |z|$, cf. [21]), the z_j are chosen in such a way that

$$\prod_{k=0}^{j-1} |z_j - z_k| = \max_{z \in K} \prod_{k=0}^{j-1} |z - z_k|, \quad j = 1, 2, \ldots \tag{6}$$

The use of Leja points is suggested by the fact that they guarantee maximal (and thus superlinear) convergence of the interpolant on every ellipse of the confocal family, and thus superlinear convergence of the corresponding matrix polynomials to the matrix exponential functions. This feature is shared also by other set of interpolation points, like e.g. standard Chebyshev points, but differently from the latter, at the same time Leja points allow to increase the interpolation degree just by adding new nodes of the same sequence; see [1,

7] for the scalar and matrix features of interpolation at Leja points. A key step in the approximation procedure is given by estimating cheaply a real focal interval, say $[a, b]$, such that the "minimal" ellipse of the confocal family which contains the spectrum (or the field of values) of the matrix is not too "large" (the underlying theoretical notion is that of "capacity" of a compact complex set). The numerical experience with matrices arising from stable spatial discretizations of parabolic equations (which are the main target of the ReLPM code) has shown that good results can be obtained at a very low cost, simply by intersecting the Gershgorin's circles of the matrix with the real axis. Indeed, it is worth stressing that the ReLPM method works well with "stiff" matrices, whose spectrum (or whose field of values) has a projection on the real axis which is non-positive and much larger than the projection on the imaginary axis; cf. [7].

The kernel of the ReLPM code is given by interpolation of $\varphi(h\lambda)$, for suitable $h \leq \Delta t$, at Leja points of the real focal interval $[a, b] = [c - 2\gamma, c + 2\gamma]$. Observe that once $\varphi(hA)\mathbf{v}$ is computed, then $\exp(hA)\mathbf{v} = h\varphi(hA)\mathbf{v} + \mathbf{v}$. In practice, it is numerically convenient to interpolate the function $\varphi(h(c+\gamma\xi))$ at Leja points $\{\xi_s\}$ of the reference interval $[-2, 2]$ (since it has capacity equal to 1, cf. [25]). Then, given the corresponding divided differences $\{d_i\}$ for such a function, the matrix Newton polynomial of degree m is

$$p_m(A) = \sum_{i=0}^{m} d_i \Omega_i \approx \varphi(hA) , \quad \Omega_i = \prod_{s=0}^{i-1} ((A - cI)/\gamma - \xi_s I) . \tag{7}$$

In general, it is not feasible to interpolate with the original time step Δt, which has to be fractionized. This happens, for example, when the expected degree for convergence is too large. The ReLPM code subdivides dynamically Δt into smaller substeps h_k, and recovers the required vector $\varphi(\Delta tA)\mathbf{v}$ according to a time marching scheme like (5), i.e.

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \varphi(h_k A)(A\mathbf{y}_k + \mathbf{v}) , \quad k = 0, 1, \ldots, k^* - 1 ; \quad \mathbf{y}_0 = \mathbf{0} , \tag{8}$$

where $\sum h_k = \Delta t$. Here we use the fact that $\Delta t\varphi(\Delta tA)\mathbf{v}$ is the solution at $t = \Delta t$ of the differential system $\dot{\mathbf{y}}(t) = A\mathbf{y}(t) + \mathbf{v}$, $\mathbf{y}(0) = \mathbf{0}$ so that $\varphi(\Delta tA)\mathbf{v} = \dfrac{\mathbf{y}_{k^*}}{\Delta t}$.

2.1 The ReLPM code

In this section we present the pseudo-code of the two main subroutines which compose the ReLPM code. They are displayed in Tables 1–2. We refer the reader to [7, 3] for other specific implementation details.

Comments to Table 1. This is the main subroutine. It accepts a matrix $A \in \mathbb{R}^{N \times N}$, a vector $\mathbf{v} \in \mathbb{R}^N$, and a time step $\Delta t > 0$. The output is a vector $\mathbf{u} \approx \varphi(\Delta tA)\mathbf{v}$. The underlying method is the time-marching scheme (5), with a dynamical managing of the variable substeps $h = h_k$, and Newton interpolation as in (7) of $\varphi(hA)$ at real Leja points related to spectral estimates for A. In steps 4. and 5. the algorithm tries an approximation of the real focal interval of a "minimal" ellipse which contains the numerical range of the underlying matrix, and the associated parameters: c is the center and γ the capacity (length/4) of the interval.

Hereafter, h and *oldh* are the current and the previous (sub)steps. For any given substep h, theoretical estimates show that superlinear convergence of the matrix interpolation polynomial should start at a degree between $h\gamma$ and $2h\gamma$, cf. [19, 7], provided that the capacity of the minimal ellipse above is relatively close to γ. Hence, convergence at reasonable tolerances of the matrix interpolation polynomial is expected for a degree lower than $h\nu = 3h\gamma$ (the factor 3 is an "empirical" choice, based on numerical experience). Hence, the input step Δt

Table 1: SUBROUTINE ReLPM

1. INPUT: A, $\mathbf{v}$, Δt, *tol*

2. CONSTANTS: $M = 124$, $(\xi_0, \ldots, \xi_M)$ array of $M + 1$ Leja points in $[-2, 2]$

3. $k := 0$, $\rho := \Delta t$, $\mathbf{u} := \mathbf{0}$, $\mathbf{w} := \mathbf{v}$

4. $a, b :=$ "extrema of the real points in the Gersghorin's circles of A"

5. $c := (a + b)/2$, $\gamma := (b - a)/4$, $\nu := 3\gamma$, $h := \min\{\Delta t, M/\nu\}$, $oldh := 0$

6. REPEAT UNTIL $\rho = 0$

    7. IF $h \neq oldh$ THEN

        8. computes $(d_0, \ldots, d_M)$, the divided differences of $\varphi(h(c+\gamma\xi))$, $\xi \in [-2, 2]$, at the Leja points $(\xi_0, \ldots, \xi_M)$; $oldh := h$

    9. CALL INTERP$(A, \mathbf{w}, h, tol, c, \gamma, M, (\xi_0, \ldots, \xi_M), (d_0, \ldots, d_M), \mathbf{q}, err, m)$

    10. IF $m > M$ THEN $h := h/2$

    11. $\rho := \rho - h$, $\mathbf{u} := \mathbf{u} + h\mathbf{q}$

    12. IF $\rho > 0$ THEN

        13. $\mathbf{w} := A\mathbf{u}$, $\mathbf{w} := \mathbf{w} + \mathbf{v}$, $k := k + 1$, $h := \min\{h, \rho\}$

14. $k^* := k$, $\mathbf{u} := \mathbf{u}/\Delta t$

15. OUTPUT: the vector $\mathbf{u} \approx \varphi(\Delta t A)\mathbf{v}$; the total number of substeps k^*

Table 2: SUBROUTINE INTERP

1. INPUT: A, $\mathbf{w}$, h, tol, c, γ, M, $(\xi_0, \ldots, \xi_M)$, $(d_0, \ldots, d_M)$

2. $\mathbf{u} := \mathbf{w}$, $\mathbf{q} := d_0\mathbf{w}$, $e_0 := \|\mathbf{q}\|_2$, $\beta := \|\mathbf{w}\|_2$, $m := 0$

3. REPEAT UNTIL $err \leq \beta\,tol$ or $m \geq M$

    4. $\mathbf{z} := (A\mathbf{u})/\gamma$, $\mathbf{u} := \mathbf{z} - (c/\gamma + \xi_m)\mathbf{u}$

    5. $m := m + 1$, $e_m := |d_m|\,\|\mathbf{u}\|_2$

    6. $\mathbf{q} := \mathbf{q} + d_m\mathbf{u}$

    7. IF $m \geq 4$ THEN $err := (e_m + \cdots + e_{m-4})/5$

8. IF $err > \beta\,tol$ THEN $m := M + 1$ ENDIF

9. OUTPUT: the vector $\mathbf{q} = p_m(A)\mathbf{w}$, the estimated error err, and the interpolation degree m, such that $\|\mathbf{q} - \varphi(hA)\mathbf{w}\|_2 \approx err$, with $err \leq \|\mathbf{w}\|_2\,tol$ when $m \leq M$, whereas a convergence failure occurred when $m > M$.

---

is possibly reduced in such a way that $[h\nu] \leq M$. In step 8., the $M + 1$ divided differences are accurately computed as the first column of $\varphi(h(c+\gamma\,\Xi_M))$, where Ξ_M is the $(M+1) \times (M+1)$ bidiagonal matrix with the Leja points $(\xi_0, \ldots, \xi_M)$ on the main diagonal and $(1, \ldots, 1)$ on the diagonal immediately below. The matrix $\varphi(h(c + \gamma\,\Xi_M))$ is approximated via 16-term Taylor expansions by the scheme proposed in [6], on the basis of [17].

Comments to Table 2. This subroutine tries to compute an approximation $\mathbf{q} = p_m(A)\mathbf{w} \approx \varphi(hA)\mathbf{w}$, cf. (7), up to an error (relative to $\|\mathbf{w}\|_2$) less than a given tolerance, where $A \in \mathbb{R}^{N \times N}$, $\mathbf{w} \in \mathbb{R}^N$ and $\Delta t \geq h > 0$.

2.2 Comparison of ReLPM with Krylov subspace methods

We now report a comparison between our Fortran ReLPM code and the Phipro Fortran code by Y. Saad, which is based on Krylov subspace approximations (cf. [23]). The comparison is made on two tests (the first one is problem # 1 in table 4, while the second one is a FD discretization of equation (1) with $\Delta x = 0.005$). For more details of the comparison, see [3]. We have computed $\varphi(\Delta t A)\mathbf{v}$ with $\mathbf{v} = (1, \ldots, 1)^t$ (corresponding to $u_0 \equiv 1$), for two values $(\Delta t)_1$ and $(\Delta t)_2$ of the time step Δt, depending on the problem. The tests have been performed on an IBM Power5 processor with 1.8Gb of RAM. The results are collected in Table 3. We report, for both methods, the number of substeps (steps), the number of total iterations (i.e., of matrix-vector products), and the CPU time. In addition, for Phipro we show also the chosen dimension m for the Krylov subspace. The tolerances for both methods have been tuned in order to have an error, relative to the "exact" result of the exponential operator, of about 10^{-6} (in the 2-norm).

Table 3: Comparing ReLPM and Phipro on the advection-diffusion discretization matrices in Examples 1–4 (the CPU times are in seconds).

Δt	Code		FE-3D			FD-3D		
	Phipro		steps	iter	CPU	steps	iter	CPU
	$m = 10$		81	891	59.8	35	385	349.3
$(\Delta t)_1$	$m = 20$		28	588	50.2	†	†	†
	$m = 25$		21	546	45.2	†	†	†
	$m = 30$		17	527	48.1	†	†	†
	$m = 50$		10	510	58.9	†	†	†
	ReLPM		12	585	32.0	3	234	133.0
	Phipro		steps	iter	CPU	steps	iter	CPU
	$m = 10$		428	4708	302.1	158	1738	1593.2
$(\Delta t)_2$	$m = 20$		152	3192	242.1	†	†	†
	$m = 25$		117	3042	250.4	†	†	†
	$m = 30$		94	2914	268.0	†	†	†
	$m = 50$		53	2703	301.0	†	†	†
	ReLPM		74	4335	231.7	16	1094	633.4

ReLPM performs better than Phipro even with an optimal choice of the Krylov subspace dimension (boxed CPU times), in spite of a smaller number of total Krylov iterations. Indeed, it is worth stressing that ReLPM computes only 1 matrix-vector product, 2 daxpys, 1 vector scaling and 1 scalar product per iteration inside INTERP. Moreover, it allocates only the matrix and 6 vectors, whereas Phipro has to allocate, besides the matrix, all the m Krylov subspace generators and 4 vectors (neglecting a matrix and some vectors of dimension m). In addition, when m increases, Phipro decreases the total number of iterations, but is penalized by the long-term recurrence in the orthogonalization process. For small m, it is penalized by a larger number of substeps. The difference in storage requirements produces remarkable consequences in the FD-3D example. There, the matrix is extremely large, and Phipro can work only with $m \leq 10$ due to the memory limitations (1.8Gb), becoming about 2.5 times slower than ReLPM.

6

3 Parallelization

A standard data-parallel implementation of the ReLPM algorithm has been performed. The parallel program is written in Fortran 90, and uses the MPI standard [18] for interprocessor communication. To perform an efficient parallel implementation of the ReLPM we choose to use compressed sparse row (CSR) storage of the nonzero matrix elements. Matrix A is is uniformly partitioned by rows among the p processors, so that $n \approx N/p$ rows are assigned to each processor. This partitioning is a consequence of assigning to each processor a subset of contiguous nodes of the computational mesh. All the vectors involved in the algorithm are accordingly split. In this way the `daxpy` operations in steps 3, 11, 12 and 14 of Table 1 and in steps 2, 4 and 6 of Table 2 are performed without any communication among processors. The estimation of the focal interval (step 4 of subroutine ReLPM) and the computation of the 2-norm of a vector (to check the exit test) needs that the processors exchange only a scalar. In particular, estimating the norm of the approximation vector in parallel requires a collective communication (implemented via an `MPI_allreduce` call) which becomes costly when largely increasing the number of processors. Nevertheless the main scalability bottleneck is the sparse matrix vector product ($A\mathbf{u}$) of steps 13 (Table 1) and 4 (Table 2) which requires the processors to communicate a number of elements of vector $\mathbf{u}$.

3.1 Efficient matrix-vector product.

Our implementation of the matrix-vector product is tailored for application to sparse matrices and minimizes data communication between processors. Within the ReLPM algorithm, the vector $\mathbf{z} := A\mathbf{u}$ has to be calculated. Every processor do exchange entries of its local components of vector $\mathbf{u}$ with a very small number of other processors (compared to the total number p). Besides, it is always a reciprocal exchange, since matrix A has a symmetric nonzero pattern due to the Galerkin FE and central FD discretization schemes we adopted. Thus, we implemented data exchanges by calling the `MPI_SendRecv` routine.

Let us consider a given processor with processor identifier (pid) say $r, 0 \leq r \leq p-1$. Assume for simplicity that N is exactly np, and denote by S the indices of the nonzero entries of matrix A

$$S = \{(i,j) : a_{ij} \neq 0\}.$$

The set S is normally referred as the nonzero pattern of A. After distributing the matrix, the subset P^r containing the indices corresponding to nonzero elements of A belonging to processor r can be defined as

$$P^r = \{(i,j) : rn+1 \leq i \leq (r+1)n \} \cap S.$$

This set can be partitioned into two subsets

$$P^r_{\text{loc}} = \{(i,j) \in P^r, \ rn+1 \leq j \leq (r+1)n\} \qquad P^r_{\text{nonloc}} = P^r \backslash P^r_{\text{loc}}.$$

For every processor r we also define the subsets $C^r_k, R^r_k, k \neq r$ of indices as:

$$R^r_k = \{i : (i,j) \in P^r_{\text{nonloc}}, kn+1 \leq j \leq (k+1)n\}$$

and

$$C^r_k = \{j : (i,j) \in P^r_{\text{nonloc}}, kn+1 \leq j \leq (k+1)n\}$$

Processor r has in its local memory the elements of the vector $\mathbf{u}$ whose indices lie in the interval $[rn+1, (r+1)n]$. Before computing the matrix-vector product processor r does the

following: for every k such that $R_k^r \neq \emptyset$ sends to processor k the components of vector $\mathbf{u}$ whose indices belongs to R_k^r; gets from every processor k such that $C_k^r \neq \emptyset$, the elements of $\mathbf{u}$ whose indices are in C_k^r. We subdivided the overall communication on *communication phases*. On each *phase* two exchanges are completed, one with a processor on the left (e.g. a processor with pid less than r) and another with a processor on the right (e.g. pid greater than r). We scheduled the matching of couples of processors exchanging data in order to minimize waiting times, that is, to avoid that processors wait for their partners to complete another communication first. When all the communication phases have completed processors are able to compute locally their part of the matrix-vector product.

4   Numerical experiments

To understand how the parallel ReLPM algorithm performs with problems of varying degrees of difficulty we considered a set of problems concerning FE and FD discretizations of 3-D advection-dispersion models like (1). The test problems are summarized in Table 4.

Table 4: Summary of test problems.

#	Discretization	Type	θ	Δx	N	nnz	Δt
1	FE	Adv-Diff			534681	7837641	10.0
2	FE	Adv-Diff			2099601	31079601	10.0
3	FD	Diff	0	1/128	2097152	14680064	0.52
4	FD	Diff	0	1/256	16777216	117440512	0.52
5	FD	Adv-Diff	25.0	1/256	16777216	117440512	0.04
6	FD	Adv-Diff	100.0	1/512	134217728	939524096	0.01
7	FD	Adv-Diff	1000.0	1/1024	1073741824	7516192768	0.0003

We report the timing results of the parallel ReLPM code running on HPCx with a number of processors varying from $p = 2, \cdots, 1024$, depending on the problem size. HPCx is a large cluster of IBM SMP nodes actually housed in Daresbury, England. HPCx is available for HPC-Europa visitors at EPCC (Edinburgh Parallel Computing Center). HPCx is presently 46th in the TOP 500 list of fastest machines in the world.

4.1   HPCx architecture overview

At the time of writing the HPCx system is composed of IBM eServer 575 nodes for the compute and IBM eServer 575 nodes for login and disk I/O. Each eServer node (frame) contains 16 processors. At present there are two service nodes. The main HPCx service provides 96 frames for compute jobs for users, giving a total of 1536 processors (however the largest CPU count for a single job is 1024). Each eServer system frame consists of 16 1.5 GHz POWER5 processors. In the POWER5 architecture, a chip contains two processors, together with the Level 1 (L1) and Level 2 (L2) cache. Each processors has its own L1 instruction cache of 32 Kb and L1 data cache of 64 Kb integrated onto one chip. Also on board the chip is the L2 cache (instructions and data) of 1.9 Mbyte, which is shared between the two processors. Four chips (8 processors) are integrated into a multi-chip module (MCM). Two MCMs (16 processors) comprise one frame. Each MCM is configured with 128 Mb of L3 cache and 16 GB of main memory. The total main memory of 32 GB per frame is shared between the 16 processors of the frame.

Inter node communication is provided by an IBM's High Performance Switch (HPS), also known as "Federation." Each frame has two network adapters and there are two links per

adapter, making a total of four links between each of the frames and the switch network. Intra node communication is accomplished via shared memory.

4.2 Results on FE matrices

In our first study we consider a 3-D advection-dispersion problem given by equation (1), where D is the hydrodynamic dispersion tensor, $D_{ij} = \alpha_T |\vec{v}| \delta_{ij} + (\alpha_L - \alpha_T) v_i v_j / |\vec{v}|$, $1 \leq i, j \leq 3$ (α_L and α_T being the longitudinal and transverse dispersivity, respectively). The domain is $\Omega = [0, 1] \times [0.0.5] \times [0, 1]$, discretized by a regular mesh of $N = 161 \times 81 \times 41 = 534\,681$ nodes and $3\,072\,000$ tetrahedral elements. The source term is $\phi = 0$; the boundary conditions are homogeneous Dirichlet on $\Gamma_{\mathcal{D}} = \{0\} \times [0.2, 0.3] \times [0, 1]$, whereas homogeneous Neumann conditions are imposed on $\Gamma_{\mathcal{N}} = \partial\Omega \setminus \Gamma_{\mathcal{D}}$. The velocity is $\vec{v} = (1, 0, 0)$, the transmissivity coefficients are piecewise constant and vary by an order of magnitude depending on the elevation of the domain, $\alpha_L(x_3) = \alpha_T(x_3) \in \{0.0025, 0.025\}$. The resulting FE matrix has $7\,837\,641$ nonzeros (average per row ≈ 14).

The second example considered regards the same problem as of Example 1, but in this case the domain is discretized with a regular grid of $N = 161 \times 81 \times 161 \approx 2.1 \times 10^6$ nodes and about 12 millions of tetrahedral elements. Matrix of discretization A has roughly 2.1×10^6 rows and 3.1×10^7 nonzero elements.

We distribute the 41(161) horizontal strata of the mesh among the processors by associating to any processor more than one stratum, that is more than 161×81 grid points, in the case $p \leq 41(161)$. This implies that each processor communicates with at most two other processors. If, on the contrary, $p > 41(161)$ a single stratum is shared by more than one processor, thus increasing the number of communicating processors.

Throughout the whole section we will denote with T_p the CPU elapsed times expressed in seconds (unless otherwise stated) when running the code on p processors. We include a relative measure of the parallel efficiency achieved by the code. To this aim we will denote as $S_p^{(\bar{p})}$, the pseudo speedup computed with respect to the smallest number of processors ($\bar{p}$) used to solve the given problem:

$$S_p^{(\bar{p})} = \frac{T_{\bar{p}} \bar{p}}{T_p}.$$

We will denote $E_p^{(\bar{p})}$ the corresponding relative efficiency, obtained according to

$$E_p^{(\bar{p})} = \frac{S_p^{(\bar{p})}}{p} = \frac{T_{\bar{p}} \bar{p}}{T_p p}.$$

Table 5: Timing results and statistics for scalability study for Example 1.

p	steps	iter	T_p	$S_p^{(2)}$	$E_p^{(2)}$
2	74	4319	122.6		
4	74	4319	53.5	**4.6**	**>100%**
8	74	4319	26.1	**9.4**	**>100%**
16	74	4319	15.7	15.6	98%
32	74	4319	10.2	24.0	75%
64	74	4319	7.1	34.6	54%
128	74	4319	4.0	61.4	48%

Table 6: Timing results and statistics for scalability study for Example 2.

p	steps	iter	T_p	$S_p^{(2)}$	$E_p^{(2)}$
2	91	5001	629.6		
4	91	5001	323.9	3.9	98%
8	91	5001	167.8	7.5	94%
16	91	5001	86.0	14.6	91%
32	91	5001	36.0	**35.0**	**>100%**
64	91	5001	21.3	59.1	92%
128	91	5001	13.6	92.6	72%

Tables 5–6 contain the results obtained when running the parallel ReLPM code on HPCx to compute the solution of the two FE problems described above for a value of $\Delta t = 10$ seconds. The initial approximation vector is $\mathbf{c}_0 = (1, \cdots, 1)^T$, and the relative tolerance *tol* in subroutine `interp` was set to 10^{-7} for Example 1 and to 10^{-8} for Example 2. With such choices, the 2-norm of the error is expected to be at least one order of magnitude below the discretization error, (see [5] for a discussion on proper selection of parameter *tol*).

Results showing a speedup larger than the number of processors employed or an efficiency larger than 100% are printed in boldface. For these small/medium size problems we scaled up the number of processors from 2 to 128. For the smallest problem the relative parallel efficiency decreases till 48% when 128 processors are used. As known, when a small problem is solved across a large number of processors, parallel efficiency is expected to decrease. In particular, this is due to fact that each processor needs communicating with a number of other processors, which is problematic because of communication latencies. Due to the increased problem size of Example 2 the parallel efficiency rises up to 72% when using 128 processors. The better scaling of the second problem can be seen also in Figure 1. Here we show the scaling behavior with respect to the number of processors of the ReLPM code for the two FE matrices. Also, for comparison an ideal perfect scaling line is included in the figure.

We obtained superspeedup values with $p = 4, 8$ for Example 1 and with $p = 32$ for Example 2. This results are likely to be ascribed to cache effects. Problem 1 can be kept in Level 3 cache starting from $p = 8$ processors while problem 2, being 4 times larger, stays in cache from $p = 32$ processors.

4.3 Results on FD matrices

Regarding 3D FD discretizations of equation (1), we have considered the domain $\Omega = (0, 1)^3$, $D = I$, $\vec{v} = \theta \cdot (1, 1, 1)^T$, and homogeneous Dirichlet boundary conditions. The source term is $\phi = 0$. The matrix A has been generated by adopting a second order central FD discretization on a uniform grid with stepsize $\Delta x = 1/nx$, with $nx = 128, 256, 512, 1024$ being nx^3 the total number of grid points (and hence the number of rows of matrix A). Table 4 summarizes all the tests considered: Examples 3 and 4 are purely diffusion problems, while Examples 5 to 7 arise from advection diffusion problems with different Peclet numbers. The initial approximation vector is $\mathbf{c}_0 = (1, \cdots, 1)^T$, and the relative tolerance *tol* in subroutine `interp` was set to 10^{-8} for all the FD problems.

The scaling results from running parallel ReLPM for large scale FD discretizations on HPCx are given in Tables 7–11. For this study we considered a number of processors ranging from 16 to 1024 (the largest CPU count for a single job on HPCx). We give the measured
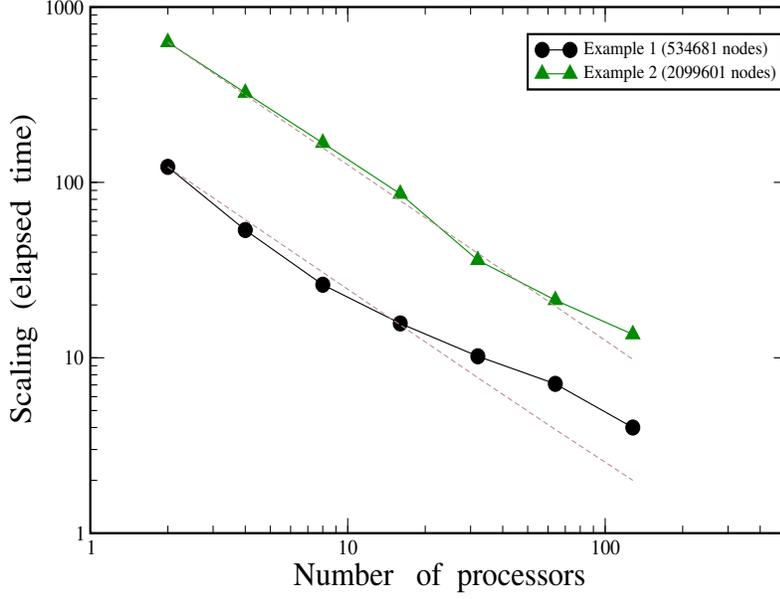
Figure 1: Scaling results – total elapsed time of the ReLPM code on HPCx for the two FE matrices (number of processors ranging from 2 to 128). Dashed lines represent perfect scalability.

elapsed times when running the parallel ReLPM code on HPCx to compute the solution of the five FD problems for a value of Δt which again depends on the specific problem. In particular for the largest problems (with more than one thousand millions nodes) the value of Δt was selected particularly small in order to keep bounded the CPU cost of the run on the HPCx service.

Table 7: Timings and speedups for Example 3 ($128 \times 128 \times 128$ grid nodes).

p	steps	iter	T_p	$S_p^{(2)}$	$E_p^{(2)}$
2	609	29310	2532.7		
4	609	29310	1256.5	4.0	100%
8	609	29310	566.3	**8.9**	**>100%**
16	609	29310	327.9	15.4	96%
32	609	29310	167.1	30.3	95%
64	609	29310	97.6	51.9	81%
128	609	29310	69.5	72.9	57%

The pseudo efficiency values provided in Tables 7–11 are always larger than 79% with the only exception of Example 3, where an efficiency of 57% is reported with 128 processors. In

Table 8: Timings and speedups for Example 4 ($256 \times 256 \times 256$ grid nodes).

p	steps	iter	T_p	$S_p^{(16)}$	$E_p^{(16)}$
16	878	62148	6617.9		
32	878	62148	3191.7	**33.1**	**>100%**
64	878	62148	1675.8	63.2	99%
128	878	62148	875.6	120.6	94%
256	878	62148	509.6	210.2	82%

Table 9: Timings and speedups for Example 5 ($256 \times 256 \times 256$ grid nodes).

p	steps	iter	T_p	$S_p^{(16)}$	$E_p^{(16)}$
16	189	9451	998.8		
32	189	9451	502.8	31.8	99%
64	189	9451	254.1	62.4	98%
128	189	9451	128.7	124.2	97%
256	189	9451	78.5	203.7	80%

Table 10: Timings and speedups for Example 6 ($512 \times 512 \times 512$ grid nodes).

p	steps	iter	T_p	$S_p^{(32)}$	$E_p^{(32)}$
32	187	8432	4100.4		
64	187	8432	2085.6	63.0	98%
128	187	8432	1106.1	118.4	92%
256	187	8432	553.6	236.8	92%
512	187	8432	322.1	406.4	79%

Table 11: Timings and speedups for Example 7 ($1024 \times 1024 \times 1024$ grid nodes).

p	steps	iter	T_p	$S_p^{(256)}$	$E_p^{(256)}$
256	23	1706	1029.7		
512	23	1706	564.0	468.5	91%
1024	23	1706	326.0	809.0	79%

this case the loss of efficiency is due to the relatively small size of the problem with respect to the number of processors employed. On Examples 3 and 4, pseudo efficiency larger than one are probably due to cache effects. On the largest problems these effects are not so evident because they are partially counterbalanced by the increasing cost of communication.

We provide two types of scalability analysis. The first one consists in considering a fixed size problem and trying to approximate $\varphi(\Delta t A)\mathbf{v}$, for a suitable Δt in a shorter time by increasing the number of processors used. This type of scalability is more difficult to achieve than the other type in which the problem size and the number of processors grow accordingly.
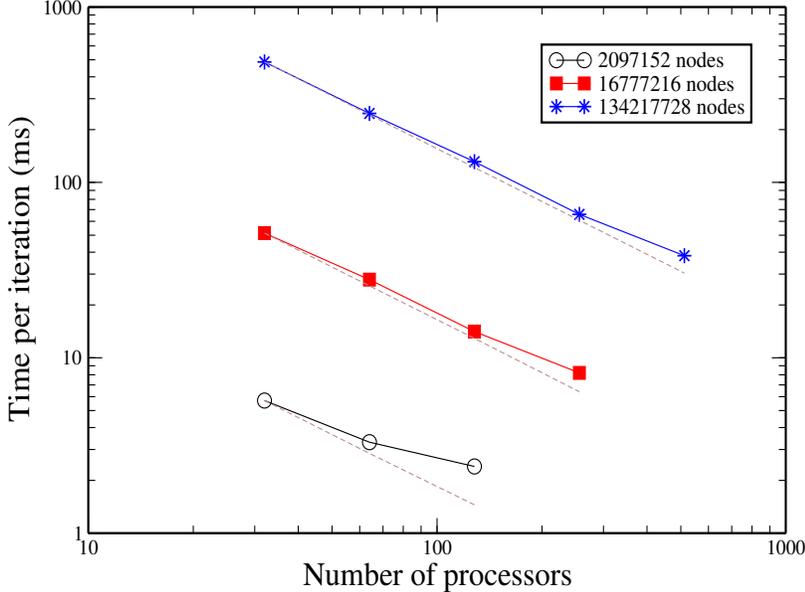
Figure 2: Scaling results – time to perform one iteration of the ReLPM code as the number of processors increases with fixed problem size (FD matrices) on HPCx (number of processors ranging from 16 to 512). Dashed lines represent perfect scalability.

This is so because increasing the number of processors normally causes an increment of the communication overhead. We present the summary of results for the first type of scalability analysis on Figure 2, where we plot the average time to perform one iteration of the ReLPM code as the number of processors increases with three different (fixed) problem sizes. The number of processors varies from 16 to 512. These plots are very close to the dashed lines describing optimal scalability. For instance, in the largest problem presented in this figure (512^3 mesh), only for the largest number of processors employed (512) there is a slight difference between observed and optimal efficiency.

In the second study we measure scalability by increasing the size of the problem (by a factor of 2^3) and the number of processors accordingly. In this case, scalability is achieved when the time per iteration is bounded or slightly increasing. We summarize the results of this analysis in Figure 3 where we plot the average time to perform one iteration of the ReLPM code as the problem size increases with fixed number of processors on HPCx. The horizontal lines join together the same average time per iteration corresponding to a given problem running on p processors and a problem eight times larger running on $8p$ processors. A slight loss of efficiency is clearly shown by the position of the points in the figure which are above this horizontal (dotted) line.

5   Concluding remarks

In this paper we have assessed the parallel performance of a massively parallel MPI code for the solution of advection-diffusion equations on 3-D domains. The code is a parallel
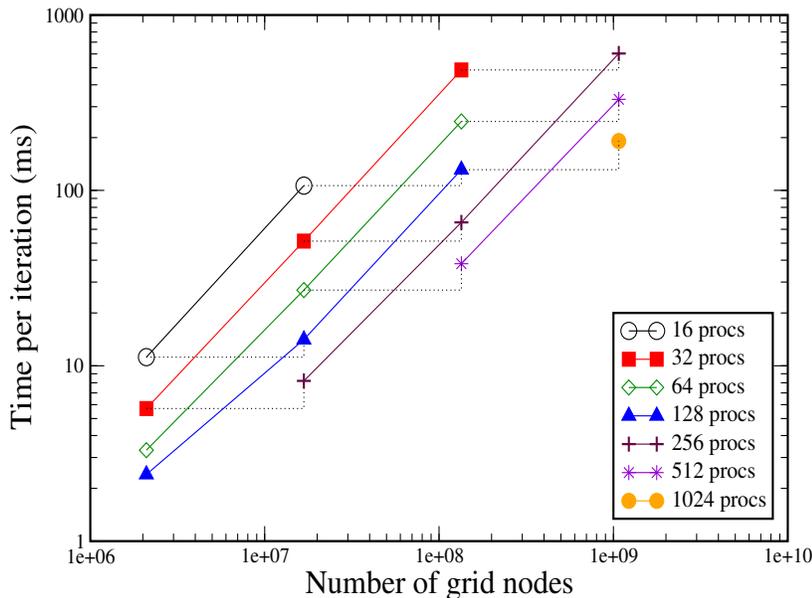
Figure 3: Scaling results – time to perform one iteration of the ReLPM code as the problem size increases with fixed number of processors on HPCx (FD matrices). Problem size increases by a factor of 8.

implementation of ReLPM (Real Leja Points Method) for the exponential integration of large-scale sparse systems of ODEs, generated by FD and FE discretizations of advection-diffusion models, which has been previously shown [5, 2] to outperform the Crank Nicolson method. An optimized parallel sparse matrix-vector product routine has been implemented. The numerical results presented demonstrate that the ReLPM propagator displays excellent parallelization properties. Our implementation has achieved almost linear scaling parallelization with both the number of processors and the problem size. In view of the very satisfactory parallel efficiency of ReLPM, research is undergoing in order to use this code as a building block for the parallel solution of large nonlinear advection-diffusion problems.

References

[1] J. Baglama, D. Calvetti, and L. Reichel, *Fast Leja points*, Electron. Trans. Numer. Anal., 7 (1998), pp. 124–140.

[2] L. Bergamaschi, M. Caliari, A. Martínez, and M. Vianello, *A parallel exponential integrator for large-scale discretizations of advection-diffusion models*, in Recent Advances in PVM and MPI, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, vol. 3666 of Lecture Notes in Computer Sciences, Springer-Verlag Heidelberg, 2005, pp. 483–492.

[3] ——, *Comparing Leja and Krylov approximations of large scale matrix exponentials.*, in Computational Sciences, ICCS 2005, Reading (UK), vol. 3994 of Lecture Notes in Computer Sciences, Springer-Verlag Heidelberg, 2006, pp. 685–692.

[4] L. BERGAMASCHI, M. CALIARI, AND M. VIANELLO, *Efficient approximation of the exponential operator for discrete 2d advection-diffusion problems*, Numer. Lin. Alg. Appl., 10 (2003), pp. 271–289.

[5] ——, *The ReLPM exponential integrator for FE discretizations of advection-diffusion equations*, in Computational Science - ICCS 2004: 4th International Conference Krakow, Poland, June 6-9, 2004, Proceedings, Part IV, M. Bubak, G. D. van Albada, and P. Sloot, eds., vol. 3036 of Lecture Notes in Computer Sciences, Springer-Verlag Heidelberg, 2004, pp. 434–442.

[6] M. CALIARI, *Accurate evaluation of divided differences for polynomial interpolation of exponential propagators*, (2006). submitted.

[7] M. CALIARI, M. VIANELLO, AND L. BERGAMASCHI, *Interpolating discrete advection-diffusion propagators at spectral Leja sequences*, J. Comput. Appl. Math., 172 (2004), pp. 79–99.

[8] ——, *The LEM exponential integrator for advection-diffusion-reaction equations*, J. Comput. Appl. Math., (2006). published online on December 5, 2006. doi:10.1016/j.cam.2006.10.055.

[9] S. M. COX AND P. C. MATTHEWS, *Exponential time differencing for stiff systems*, J. Comput. Phys., 176 (2002), pp. 430–455.

[10] J. CRANK AND P. NICOLSON, *A practical method for numerical evaluation of solutions of partial differential equations of the heat conduction type.*, in Proc. Cambridge Philos. Soc., vol. 40, 1947, pp. 50–67.

[11] C. GONZÁLEZ, A. OSTERMANN, AND M. THALHAMMER, *A second-order Magnus-type integrator for nonautonomous parabolic problems*, J. Comput. Appl. Math., 189 (2006), pp. 142–156.

[12] M. HOCHBRUCK AND C. LUBICH, *On Krylov subspace approximations to the matrix exponential*, SIAM J. Numer. Anal., 34 (1997), pp. 1911–1925.

[13] M. HOCHBRUCK, C. LUBICH, AND H. SELHOFER, *Exponential integrators for large systems of differential equations*, SIAM J. Sci. Comput., 19 (1998), pp. 1552–1574.

[14] M. HOCHBRUCK AND A. OSTERMANN, *Explicit exponential Runge-Kutta methods for semilinear parabolic problems*, SIAM J. Numer. Anal., 43 (2005), pp. 1069–1090 (electronic).

[15] S. KROGSTAD, *Generalized integrating factor methods for stiff PDEs*, J. Comput. Phys., 203 (2005), pp. 72–88.

[16] F. LEJA, *Sur certaines suites liées aux ensembles plans et leur application à la représentation conforme*, Ann. Polon. Math., 4 (1957), pp. 8–13.

[17] A. MCCURDY, K. C. NG, AND B. N. PARLETT, *Accurate computation of divided differences of the exponential function*, Math. Comp., 43 (1984), pp. 501–528.

[18] MESSAGE PASSING INTERFACE FORUM, *MPI: A message passing interface standard*, June 1995. also available online at `http://www.mpi-forum.org/`.

[19] I. MORET AND P. NOVATI, *The computation of functions of matrices by truncated Faber series*, Numer. Funct. Anal. Optim., 22 (2001), pp. 697–719.

[20] ——, *An interpolatory approximation of the matrix exponential based on Faber polynomials*, J. Comput. Appl. Math., 131 (2001), pp. 361–380.

[21] L. REICHEL, *Newton interpolation at Leja points*, BIT, 30 (1990), pp. 332–346.

[22] Y. SAAD, *SPARSKIT: A basic tool kit for sparse matrix computation*, Technical Report 1029, University of Illinois, CSRD, Urbana, 1990.

[23] Y. SAAD, *Analysis of some Krylov subspace approximations to the matrix exponential operator*, SIAM J. Numer. Anal., 29 (1992), pp. 209–228.

[24] R. B. SIDJE, *Expokit. A Software Package for Computing Matrix Exponentials*, ACM Trans. Math. Software, 24 (1998), pp. 130–156.

[25] H. TAL-EZER AND R. KOSLOFF, *An accurate and efficient scheme for propagating the time dependent Schrödinger equation*, J. Chem. Phys., 81 (1984), pp. 3965–3971.

[26] H. TAL-EZER, R. KOSLOFF, AND C. CERIEN, *Low-order polynomial approximation of propagators for the time-dependent Schrödinger equation*, J. Comput. Phys., 100 (1992), pp. 179–187.

[27] M. TOKMAN, *Efficient integration of large stiff systems of ODEs with exponential propagation iterative (EPI) methods*, J. Comput. Phys., 213 (2006), pp. 748–776.