Parallel Methods for ODEs

Levels of parallelism

There are a number of levels of parallelism that are possible within a program to numerically solve ODEs.

An obvious place to start is with manual code restructuring and/or a parallelizing compiler.

This can be augmented with replacing serial routines with corresponding parallel ones, e.g., linear algebra.

However, these levels of parallelization might not be expected to yield much in terms of improvement because a lot of code may still have to run in serial.

A more fruitful approach would likely be to redesign the fundamental sequential nature of the algorithms used for solving ODEs to target parallelism, e.g., using block predictor-corrector algorithms that permit many values to be computed simultaneously within a step.

This fine-grained approach to parallelization is called **parallelism across the method**.

Levels of parallelism

An important coarse-grained approach can be classified as **parallelism across the system**.

In this case, the ODEs themselves are partitioned in such a way that sub-problems can be solved in parallel.

For example, so-called *multi-rate methods* partition the ODEs into sub-problems that are integrated with different step sizes.

$$\begin{aligned}\dot{\mathbf{y}}_1 &= \mathbf{f}_1(t, \mathbf{y}_1, \mathbf{y}_2),\\ \dot{\mathbf{y}}_2 &= \mathbf{f}_2(t, \mathbf{y}_1, \mathbf{y}_2).\end{aligned}$$

The typical scenario is that one system varies rapidly and hence requires a small step size, whereas the other system varies slowly and hence can be integrated with a larger step size.

The key to the success of such methods is the amount of inter-processor communication that is required.

Levels of parallelism

A third approach to the parallel solution of ODEs is called **parallelism across the steps**.

In this approach, equations are solved in parallel over a large number of steps.

It is highly likely that an effective strategy for parallelizing the solution of ODEs will involve aspects of all three of these levels of parallelism.

It is important to appreciate which types of parallelism are small scale (and hence can really only take advantage of a relatively small number of processors) and those that are large scale (and can take advantage of massive parallelism, i.e., thousands of processors).

In particular, parallelism across the system has the potential for massive parallelism, especially for systems arising from the method of lines.

Parallelism across the method is generally suitable for small-scale parallelization only.

When to parallelize

Not every problem needs to be parallelized in order to find its solution.

There are only two scenarios in which parallelization makes sense as a way to help solve a problem:

1. The problem is too big to fit into the memory of one computer.

2. The problem takes too long to run.

The goal in both of these scenarios can be described as reducing the amount of (real) time it takes to get the solution[1].

Note that this is **not** the same as reducing the overall amount of computation.

[1] In the first scenario, the original time required can be viewed as infinite (no such single computer exists) or indefinite (until you buy a new computer).

When to parallelize

Problems that are amenable to parallel solution typically have some or all of the following attributes:

- The right-hand side function of the ODE is expensive to evaluate.
- The interval of integration is long.
- Multiple integrations must be performed.
- The size of the system is large.

Of these, the last one is probably the most important.

Parallel Runge–Kutta methods

Runge–Kutta methods can be analysed for parallelizability via the sparsity of the Butcher tableau.

As an example, consider a 5-stage explicit Runge–Kutta method with the following sparsity pattern:

$$\mathbf{A} = \begin{bmatrix} 0 & & & & \\ \times & 0 & & & \\ \times & 0 & 0 & & \\ \times & \times & \times & 0 & \\ \times & \times & \times & 0 & 0 \end{bmatrix}$$

Stages (2 and 3) and (4 and 5) can be computed concurrently.

With this construction, the achievable order is usually strictly less than what is theoretically possible.

This is particularly noticeable for explicit methods because of more severe order restrictions; implicit methods offer more potential in this case.

Diagonal Runge–Kutta methods

The simplest parallel implicit Runge–Kutta methods are those with a strictly diagonal $\mathbf{A}$ matrix.

Recall that an s-stage implicit Runge–Kutta method applied to an ODE of size m generally requires the simultaneous solution of sm nonlinear algebraic equations at each step.

A strictly diagonal $\mathbf{A}$ decouples these equations into s independent systems of size m that can be solved in parallel.

It can be shown that the maximum order of Runge–Kutta methods with strictly diagonal $\mathbf{A}$ is 2.

This may be sufficient for some applications, but generally Runge–Kutta methods with a strictly diagonal $\mathbf{A}$ are of limited use.

Block-Diagonal RK methods

A natural way to overcome the order barrier associated with strictly diagonal $\mathbf{A}$ and yet maintain parallelizability is to allow $\mathbf{A}$ to be *block diagonal*.

The blocks can be used to build in other desirable properties for the method (such as A-stability or high order) and still be processed in parallel.

An efficient construction is to have the diagonal elements on a given block be the same.

The following is a 4-stage, 2-parallel, 2-processor, A-stable method of order 4 (Iserles and Nørsett, 1990):

$$\begin{array}{c|cccc}
\frac{1}{3} & \frac{1}{3} & & & \\
\frac{2}{3} & \frac{1}{3} & \frac{1}{3} & & \\
\frac{21+\sqrt{57}}{48} & & & \frac{21+\sqrt{57}}{48} & \\
\frac{27-\sqrt{57}}{48} & & & \frac{3-\sqrt{57}}{24} & \frac{21+\sqrt{57}}{48} \\
\hline
 & \frac{9+3\sqrt{57}}{16} & \frac{9+3\sqrt{57}}{16} & -\frac{1+3\sqrt{57}}{16} & -\frac{1+3\sqrt{57}}{16}
\end{array}$$

Block-Diagonal RK methods

If we further assume that the diagonal blocks may be full, we can construct a 4-stage, 2-parallel, 2-processor, L-stable method of order 4 (Iserles and Nørsett, 1990):

$$
\begin{array}{c|cccc}
\frac{3-\sqrt{3}}{6} & \frac{5}{12} & \frac{1-2\sqrt{3}}{12} & & \\
\frac{3+\sqrt{3}}{6} & \frac{1+2\sqrt{3}}{12} & \frac{5}{12} & & \\
\frac{3-\sqrt{3}}{6} & & & \frac{1}{2} & -\frac{\sqrt{3}}{6} \\
\frac{3+\sqrt{3}}{6} & & & \frac{\sqrt{3}}{6} & \frac{1}{2} \\
\hline
 & \frac{3}{2} & \frac{3}{2} & -1 & -1
\end{array}
$$

If implemented with 2 processors, the cost of this method is the same as the two-stage Gauss method, which is also of order 4, but it is only A-stable, not L-stable (if that is an advantage for a particular problem).

In general, we will categorize parallel RK methods as s-stage, k-parallel, and ℓ-processor methods, where k is the number of blocks, ℓ is the maximum block size, and $s = k\ell$ is the (usual) number of stages.

Block-Diagonal RK methods

So far we have only considered Runge–Kutta methods with completely decoupled diagonal blocks.

If we allow $\mathbf{A}$ to be block lower triangular with diagonal diagonal blocks, we can construct an L-stable method of order 4 with an embedded method of order 3 for local error control (Iserles and Nørsett, 1990):

$$\begin{array}{c|cccc}
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{2}{3} & & \frac{2}{3} & & \\
\frac{1}{2} & -\frac{5}{2} & \frac{5}{2} & \frac{1}{2} & \\
\frac{1}{3} & -\frac{5}{3} & \frac{4}{3} & & \frac{2}{3} \\
\hline
 & -1 & \frac{3}{2} & -1 & \frac{3}{2}
\end{array}$$

The first pair of stages can be computed concurrently; then the second pair of stages can be computed concurrently, in this case using the same $\mathbf{LU}$ factorization as the first pair.

Block-Diagonal RK methods

There is an order barrier on block-diagonal RK methods that can be proved:

Theorem 1. *Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the distinct diagonal coefficients of $\mathbf{A}$ with respective multiplicities $\mu_1, \mu_2, \ldots, \mu_n$. Then the order p of any k-parallel, ℓ-processor parallel DIRK satisfies*

$$p \leq 1 + \sum_{i=1}^{n} \min(\mu_i, k).$$

Thus the maximum order of any k-parallel, ℓ-processor SDIRK method (i.e., $\lambda_1 = \lambda_2 = \ldots = \lambda_n$) is $k+1$.

This is the same order bound for SDIRK methods when implemented serially.

So improved order can only be attained if diagonal elements are allowed to vary within a block.

Multiply implicit RK methods

The structure of the Runge–Kutta matrix $\mathbf{A}$ greatly affects the cost of implementing the method.

We have seen this to be true not only in differentiating between explicit and implicit RK methods, but also within the classes of explicit and implicit RK methods themselves and within serial and parallel architectures.

In 1976, Butcher proposed an ingenious technique for improving the computational efficiency of an implicit Runge–Kutta based on transforming $\mathbf{A}$ to its Jordan canonical form.

Suppose there exists a non-singular matrix $\mathbf{T}$ such that

$$\mathbf{TAT}^{-1} = \mathbf{\Lambda}.$$

Consider the RK method defined in tensor notation.

Let

$$\mathbf{Y} = (\mathbf{Y}_1^T, \mathbf{Y}_2^T, \ldots, \mathbf{Y}_s^T)^T \in \mathbb{R}^{sm},$$
$$\mathbf{F}(\mathbf{Y}) = \mathbf{f}(\mathbf{Y}_1)^T, \mathbf{f}(\mathbf{Y}_2)^T, \ldots, \mathbf{f}(\mathbf{Y}_s)^T)^T.$$

Then any RK method can be written as

$$\mathbf{Y} = \mathbf{e} \otimes \mathbf{y}_n + \Delta t(\mathbf{A} \otimes \mathbf{I}_m)\mathbf{F}(\mathbf{Y}), \tag{1a}$$
$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t(\mathbf{b}^T \otimes \mathbf{I}_m)\mathbf{F}(\mathbf{Y}), \tag{1b}$$

where $\otimes$ denotes the tensor (or Kronecker) product between two matrices; i.e., for matrices $\mathbf{A} \in \mathbb{R}^{m_A \times n_A}$ and $\mathbf{B} \in \mathbb{R}^{m_B \times n_B}$,

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1,n_A}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m_A,1}\mathbf{B} & \cdots & a_{m_A,n_A}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{m_A m_B \times n_A n_B}$$

Equations (1) represent a system of nonlinear algebraic equations of size sm.

Each Newton iteration involves solution of the linear system for the correction $\boldsymbol{\delta}$

$$(\mathbf{I}_s \otimes \mathbf{I}_m - \Delta t \tilde{\mathbf{J}})\boldsymbol{\delta} = \boldsymbol{\Psi},$$

where $\tilde{\mathbf{J}}$ is a block matrix of size sm with blocks

$$\tilde{\mathbf{J}}(i,j) = a_{ij}\mathbf{J}(\mathbf{Y}_j), \qquad i,j = 1,2,\ldots,s,$$

and

$$\boldsymbol{\delta} = (\boldsymbol{\delta}_1^T, \ldots, \boldsymbol{\delta}_s^T)^T,\ \boldsymbol{\Psi} = (\boldsymbol{\Psi}_1^T, \ldots, \boldsymbol{\Psi}_s^T)^T,$$

$$\boldsymbol{\Psi}_i = -\mathbf{Y}_i + \mathbf{y}_n + \Delta t \sum_{j=1}^{s} a_{ij}\mathbf{f}(\mathbf{Y}_j), \quad i = 1,2,\ldots,s,$$

so that $\mathbf{Y} \leftarrow \mathbf{Y} + \boldsymbol{\delta}$.

Freezing $\mathbf{J}$ at $\mathbf{y}_n$ leads to the linear system

$$(\mathbf{I}_s \otimes \mathbf{I}_m - \Delta t \mathbf{A} \otimes \mathbf{J}_n)\boldsymbol{\delta} = \boldsymbol{\Psi}. \qquad (2)$$

Now define

$$\bar{\boldsymbol{\delta}} = (\mathbf{T} \otimes \mathbf{I}_m)\boldsymbol{\delta},\ \ \bar{\mathbf{Y}} = (\mathbf{T} \otimes \mathbf{I}_m)\mathbf{Y},$$

$\mathbf{e} \otimes \bar{\mathbf{y}}_n = (\mathbf{T}\mathbf{e}) \otimes \mathbf{y}_n,\ \bar{\mathbf{f}} = (\mathbf{T} \otimes \mathbf{I}_m)\mathbf{f}((\mathbf{T}^{-1} \otimes \mathbf{I}_m)\bar{\mathbf{Y}}).$
Then (2) becomes

$$(\mathbf{I}_s \otimes \mathbf{I}_m - \Delta t \boldsymbol{\Lambda} \otimes \mathbf{J}_n)\bar{\boldsymbol{\delta}} = \bar{\boldsymbol{\Psi}}, \tag{3}$$

where

$$\bar{\boldsymbol{\delta}} = (\bar{\boldsymbol{\delta}_1}^T, \ldots, \bar{\boldsymbol{\delta}_s}^T)^T,\ \ \bar{\boldsymbol{\Psi}} = (\bar{\boldsymbol{\Psi}}_1^T, \ldots, \bar{\boldsymbol{\Psi}}_s^T)^T,$$

$$\bar{\boldsymbol{\Psi}}_i = -\bar{\mathbf{Y}}_i + \bar{\mathbf{y}}_n + \Delta t \sum_{j=1}^{s} a_{ij}\bar{\mathbf{f}}(\bar{\mathbf{Y}}_j), \quad i = 1, 2, \ldots, s.$$

Generally, the $\mathbf{LU}$ factorization involved in (2) requires

$$\mathcal{O}(s^3m^3/3) \text{ and } \mathcal{O}(s^2m^2)$$

floating-point operations for the forward and backward substitutions, respectively.

This is about s^3 times the work for a BDF method because the systems there are size m.

So if this complexity cannot be improved, implicit Runge–Kutta methods will generally be less competitive than BDF methods.

Perhaps the only situations in which implicit Runge–Kutta would be favourable would be those where high-order, L-stable methods are desired, i.e., for high-accuracy solutions to large, stiff problems.

In those situations, the facts that (1) BDF methods are (a) limited in order to 5, (b) not L-stable past first order, (c) not A-stable past second order and (2) the poor scaling of s matters less when m is large may tip the balance in favour of implicit RK methods.

Now if $\mathbf{A}$ has only real eigenvalues and is similar to a diagonal matrix, the $\mathbf{LU}$ factorization in (3) requires

$$\mathcal{O}(sm^3/3) \text{ and } \mathcal{O}(sm^2) \text{ operations.}$$

Furthermore, if $\mathbf{A}$ can be constructed to have a *one-point spectrum*, i.e., it is similar to the matrix with diagonal elements λ and subdiagonal elements $-\lambda$, then the $\mathbf{LU}$ factorization in (3) requires

$$\mathcal{O}(m^3/3) \text{ and } \mathcal{O}(sm^2) \text{ operations.}$$

These methods are called **singly implicit Runge–Kutta** (SIRK) methods.

The nonlinear equations are effectively decoupled and can be solved sequentially with one $\mathbf{LU}$ decomposition.

For parallel implementations, the decoupling is more important than having only one $\mathbf{LU}$ decomposition, but there are also certain stability advantages of having a single eigenvalue.

As we have seen, they compare well against BDF methods in terms of complexity for solving the linear systems arising in the Newton iteration.

However, at each iteration of Butcher's procedure, $\mathbf{Y}$ and $\mathbf{F}(\mathbf{Y})$ must be transformed and untransformed by the similarity matrix $\mathbf{T}$.

If ℓ iterations are required per step, the cost of a SIRK per step is

$$\mathcal{O}(m^3/3 + \ell s m^2 + 2\ell s^2 m).$$

So again only if $m \gg s, \ell$ will a SIRK method be comparable in overall cost with a BDF method, assuming of course that both methods perform satisfactorily in terms of stability, order, etc.

As well as transforming the method to Jordan canonical form, it is also possible to transform to upper Hessenberg form.

This procedure is especially beneficial if $\mathbf{J}$ can be kept constant over many steps.

Waveform Relaxation Techniques

So far we have focused on techniques for parallelism across the method.

We have seen that although some parallelism is possible, the factors of improvement are generally small and bounded even in the limit of infinite processors.

We now turn to a potentially powerful technique for parallelism across the system.

Because there are many large ODE systems (whose individual sizes far exceed the number of processors available to work on them), there is the potential for massive parallelism.

Standard application of numerical methods for ODEs become inefficient for large systems where different variables vary on different time scales.

The main problem is that the same method and step size are applied to every component.

The major challenge of allowing methods and step sizes to change according to the solution component is how to do this automatically.

In waveform relaxation (WR) methods, the full system is partitioned into a number of subsystems (potentially as small as one component each) that are integrated independently over a number of *iterative step sweeps*[2] with information exchanged between subsystems only at the end of each sweep.

WR methods were originally introduced by Lelarasmee in 1982 for solving problems in very large scale integrated (VLSI) circuit simulation.

[2]Steps typically have to be re-done until some convergence in the solution to the full system is achieved.

Picard iteration

The basic idea of WR is to solve a sequence of ODEs for a sequence of solutions $\{\mathbf{y}^{(1)}(t), \mathbf{y}^{(2)}(t), \ldots\}$ starting from an initial starting solution $\mathbf{y}^{(0)}(t)$ with the hope that the *waveforms* $\mathbf{y}^{(\nu)} \to \mathbf{y}(t)$ as $\nu \to \infty$.

The simplest and most well-known approach to this is the *Picard iteration*, in which the sequence of ODEs solved is

$$\dot{\mathbf{y}}^{(\nu+1)} = \mathbf{f}(t, \mathbf{y}^{(\nu)}(t)),\ \mathbf{y}^{(\nu+1)} = \mathbf{y}_0, t \in [t_o, t_f]. \quad (4)$$

Because the solution to (4) is

$$\mathbf{y}^{(\nu+1)}(t) = \mathbf{y}_0 + \int_{t_0}^{t} \mathbf{f}(\tau, \mathbf{y}^{(\nu)}(\tau))\, d\tau,$$

we have naturally decoupled the problem into m (embarrassingly) parallel quadrature problems.

Different quadrature methods can be used on different components, and if extra idle processors are available, each individual quadrature can itself be done in parallel.

The only communication is the updating of the waveforms between processors.

Sadly, Picard iteration usually converges *very* slowly:

Theorem 2. *The global error bound for Picard iteration applied to the linear test problem* $\dot{y} = \lambda y$ *on* $t \in [0, T]$ *satisfies*

$$|y(t) - y^{(\nu)}(t)| \leq \frac{(|\lambda| t)^{\nu+1}}{(\nu+1)!},\ t \in [0, T], \lambda < 0.$$

Proof: It is easy to see that Picard iteration generates the waveforms

$$y^{(\nu)}(t) = 1 + \lambda t + \ldots + \frac{(\lambda t)^{\nu}}{\nu!}.$$

The result follows from noting that $y(t) = e^{\lambda t}$.

So we see that the order of convergence is increased by 1 at each iteration.

But we also see that the approximation will not be very good until

$$\nu \geq |\lambda| T.$$

So if the interval of integration is large or the problem is stiff ($|\lambda|$ is large) then many iterations will be required for an accurate answer.

For nonlinear problems, we can derive a similar result using the Lipschitz constant L in place of λ.

In practice, the rate of convergence will likely be unacceptably slow.

A simple idea to improve convergence is to split the interval of integration into a series of subintervals (or *windows*) and perform Picard iteration on each window.

The idea is that convergence can be achieved more quickly on small windows, and the more accurate starting values obtained can improve convergence on subsequent windows.

Jacobi WR

It seems Picard iteration is too slow for stiff problems to allow for an efficient parallel implementation.

We now look at more general WR methods.

As an example, consider the following system:

$$\begin{aligned} \dot{y}_1 &= f_1(y_1, y_2), \quad y_1(t_0) = y_{10}, \\ \dot{y}_2 &= f_2(y_1, y_2), \quad y_2(t_0) = y_{20}, \quad t \in [t_0, T]. \end{aligned}$$

One possible iteration takes the form

$$\begin{aligned} \dot{y}_1^{(\nu+1)} &= f_1(y_1^{(\nu+1)}, y_2^{(\nu)}), \quad y_1^{(\nu+1)}(t_0) = y_{10}, \\ \dot{y}_2^{(\nu+1)} &= f_2(y_1^{(\nu)}, y_2^{(\nu+1)}), \quad y_2^{(\nu+1)}(t_0) = y_{20}; \end{aligned}$$

i.e., for each ν, two decoupled ODEs can be solved in parallel on $[t_0, T]$.

Communication between processors occurs only at the end of the iterate for (possibly interpolated) values of $y_1^{(\nu+1)}(t)$ and $y_2^{(\nu+1)}(t)$.

$y_{1,2}^{(0)}(t)$ are arbitrary but satisfy $y_{1,2}^{(0)}(t_0) = y_{10,20}$.

Because of its obvious similarity with the Jacobi method for solving linear systems of equations, this method is called the **Jacobi WR method**.

The generalization of Picard iteration takes the form

$$\dot{\mathbf{y}}^{(\nu+1)} = \mathbf{F}(t, \mathbf{y}^{(\nu+1)}, \mathbf{y}^{(\nu)}), \quad \mathbf{y}^{(\nu+1)}(t_0) = \mathbf{y}_0,$$

where $\mathbf{F} : [t_0, T] \times \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m$ is called a *splitting function* and satisfies

$$\mathbf{F}(t, \mathbf{y}, \mathbf{y}) = \mathbf{f}(t, \mathbf{y}).$$

The Jacobi iteration satisfies

$$\dot{y}_i^{(\nu+1)} = f_i(t, y_1^{(\nu)}, \ldots, y_{i-1}^{(\nu)}, y_i^{(\nu+1)}, y_{i+1}^{(\nu)}, \ldots, y_m^{(\nu)}),$$
$$i = 1, 2, \ldots, m.$$

The formal definition of a Jacobi WR method is as follows:

Definition 1. *A WR scheme is said to be of* Jacobi type *if the splitting function* $\mathbf{F}(t, \mathbf{v}, \mathbf{w})$ *satisfies*

$$\left.\frac{\partial \mathbf{F}}{\partial \mathbf{v}}\right|_{\mathbf{v}=\mathbf{u},\mathbf{w}=\mathbf{u}} = \operatorname{diag}\left(\frac{\partial \mathbf{f}}{\partial \mathbf{u}}\right).$$

Of course, the definition assumes $\partial\mathbf{f}/\partial\mathbf{u}$ exists.

The generalization to block Jacobi WR methods is now simply to allow for block diagonal elements.

The block-structured approach can improve convergence if e.g., it maintains strong coupling between parts that are strongly coupled physically.

Again, the principal advantage of Jacobi WR is that each component of the system can be solved in parallel.

Disadvantages include that convergence can be slow and a substantial amount of information may need to be passed between processors after each iteration if m and/or T are large.

Being able to identify and maintain the strong coupling of strongly coupled physical parts can make WR quite effective in practice.

The main difficulty is in identifying strong coupling automatically and/or adapting to changes in coupling.

One way to deal with this is to allow for components to belong to more than one subsystem; i.e., we allow *overlapping* of components between different subsystems.

This is also known as *multi-splitting*.

Multi-splitting WR methods

Multi-splitting is the idea of splitting a given problem in more than one way, thus allowing for components to overlap, i.e., to belong to more than one subsystem.

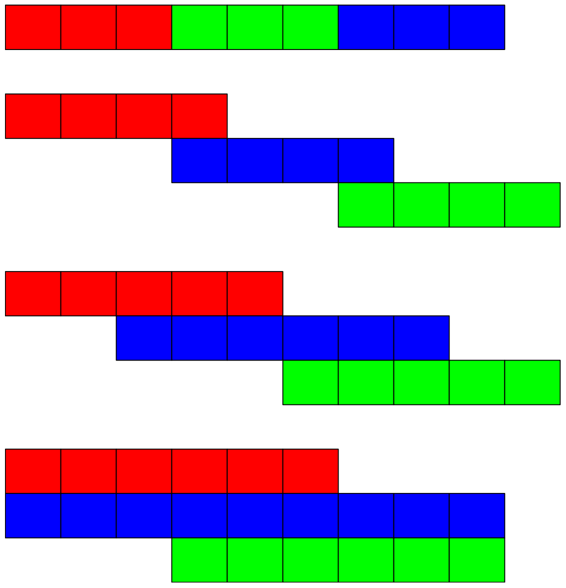
This introduces additional computational overhead because subsystems are now larger than they strictly have to be.

The hope is that overlapping will capture and preserve more of the important physical coupling and hence result in better convergence.

There remains the difficulty of determining in general what the overlap should be for a given problem, but numerical evidence suggests that some overlap is often better than none.

Here the concept of overlap represents a symmetric or two-way overlap; i.e., a given component cannot belong to only one subsystem.

In order to illustrate this concept, consider a system of 9 ODEs that has been split into 3 subsystems of dimension 3 with an overlap of 0, 1, 2, and 3.



We note that an overlap of 3 in this example means that the second subsystem coincides with the original system; so it would not make sense in practice have such a large overlap for such a small system.

More sophisticated multi-splitting methods allow for non-symmetric overlaps as well as variable overlaps.

A final issue that must be addressed in multi-splitting WR methods is the weighting to assign to the overlapping components when updating the next waveform; i.e., when a component $y_i^{(\nu+1)}(t)$ is computed in more than one subsystem, how should the different values produced be combined to produce the final $y_i^{(\nu+1)}(t)$ for the next iteration?

Suppose $y_i^{(\nu+1)}(t)$ is computed by

$$\mathbf{y}^{(\nu+1)}(t) = \sum_{\ell=1}^{N} \mathbf{E}_\ell \tilde{\mathbf{y}}_\ell^{(\nu+1)}(t),$$

where $\tilde{\mathbf{y}}_\ell^{(\nu+1)}(t)$ is from subsystem ℓ and N is the number of subsystems in which $\tilde{\mathbf{y}}_\ell^{(\nu+1)}(t)$ appears.

The matrices $\mathbf{E}_\ell$ are non-negative diagonal matrices that satisfy

$$\sum_{\ell=1}^{N} \mathbf{E}_\ell = \mathbf{I};$$

i.e., we are taking convex combinations of the $\tilde{\mathbf{y}}_\ell^{(\nu+1)}(t)$ to form $\mathbf{y}_\ell^{(\nu+1)}(t)$.

There is some evidence that suggests an *all-or-nothing* weighting is reasonable; e.g., simply assign $y_i^{(\nu+1)}(t)$ to the first subsystem found containing $y_i^{(\nu)}(t)$.

So in our example, we would set

$$\begin{aligned}
\mathbf{E}_1 &= \operatorname{diag}(1,1,1,0,0,0,0,0,0)\\
\mathbf{E}_2 &= \operatorname{diag}(0,0,0,1,1,1,0,0,0)\\
\mathbf{E}_3 &= \operatorname{diag}(0,0,0,0,0,0,1,1,1)
\end{aligned}$$

regardless of the overlap.