Concurrent DASSL Applied to Dynamic Distillation Column Simulation

Anthony Skjellum Manfred Morari

California Institute of Technology
Chemical Engineering; mail code 210-41
Pasadena, California 91125
e-mail: *tony@perseus.ccsf.caltech.edu*

Abstract

The accurate, high-speed solution of systems of ordinary differential-algebraic equations (DAE's) of low index is of great importance in chemical, electrical and other engineering disciplines. Petzold's Fortran-based *DASSL* is the most widely used sequential code for solving DAE's. We have devised and implemented a completely new C code, *Concurrent DASSL*, specifically for multicomputers and patterned on *DASSL*. In this work, we address the issues of data distribution and the performance of the overall algorithm, rather than just that of individual steps. *Concurrent DASSL* is designed as an open, application-independent environment below which linear algebra algorithms may be added in addition to standard support for dense and sparse algorithms. The user may furthermore attach explicit data interconversions between the main computational steps, or choose compromise distributions. A "problem formulator" (simulation layer) must be constructed above *Concurrent DASSL*, for any specific problem domain. We indicate performance for a particular chemical engineering application, a sequence of coupled distillation columns. Future efforts are cited in conclusion.

Introduction

In this paper, we discuss the design of a general-purpose integration system for ordinary differential-algebraic equations of low index, following up on our more preliminary discussion in [16]. The new solver, *Concurrent DASSL*, is a parallel, C-language implementation of the algorithm codified in Petzold's *DASSL*, a widely used Fortran-based solver for DAE's [11,4], and based on a loosely synchronous model of communicating sequential processes [9]. *Concurrent DASSL* retains the same numerical properties as the sequential algorithm, but introduces important new degrees of freedom compared to it. We identify the main computational steps in the integration process; for each of these steps, we specify algorithms that have correctness independent of data distribution.

We cover the computational aspects of the major computational steps, and their data distribution preferences for highest performance. We indicate the properties of the concurrent sparse linear algebra as it relates to the rest of the calculation. We describe the *proto-Cdyn* simulation layer, a distillation-simulation-oriented *Concurrent DASSL* driver which, despite specificity, exposes important requirements for concurrent solution of ordinary DAE's; the ideas behind a template formulation for simulation are, for example, expressed.

We indicate formulation issues and specific features of the chemical engineering problem – dynamic distillation simulation. We indicate results for an example in this area, which demonstrates the feasibility of this method, but the need for additional future work, both on the sparse linear algebra, and on modifying the *DASSL* algorithm to reveal more concurrency, thereby amortizing the cost of linear algebra over more time steps in the algorithm.

Mathematical Formulation

We address the following initial-value problem consisting of combinations of N linear and nonlinear coupled, ordinary differential-algebraic equations over the interval $t \in [T_0, T_1]$:

0-8186-2113-3/90/0000/0595$01.00 © 1990 IEEE

$\mathbf{IVP}(\mathbf{F}, \mathbf{u}, \mathbf{Z}_0, [T_0, T_1]; N, P)$:

$$\mathbf{F}(\mathbf{Z}, \dot{\mathbf{Z}}, \mathbf{u}; t) = \mathbf{0}, \quad t \in [T_0, T_1], \tag{1}$$

$$\mathbf{Z}(t = T_0) \equiv \mathbf{Z}_0, \quad \dot{\mathbf{Z}}(t = T_0) \equiv \dot{\mathbf{Z}}_0,$$

with unknown state vector $\mathbf{Z}(t) \in \Re^N$, known external inputs $\mathbf{u}(t) \in \Re^P$, where $\mathbf{F}(\bullet; t) \mapsto \Re^N$ and $\mathbf{Z}_0, \dot{\mathbf{Z}}_0 \in \Re^N$ are the given initial-value, derivative vectors, respectively. We will refer to Equation 1's deviation from $\mathbf{0}$ as the residuals or residual vector. Evaluating the residuals means computing $\mathbf{F}(\mathbf{Z}, \dot{\mathbf{Z}}, \mathbf{u}; t)$ ("model evaluation") for specified arguments $\mathbf{Z}$, $\dot{\mathbf{Z}}$, $\mathbf{u}$ and t.

DASSL's integration algorithm can be used to solve systems fully implicit in $\mathbf{Z}$ and $\dot{\mathbf{Z}}$ and of index zero or one, and specially structured forms of index two (and higher) [4, Chapter 5], where the index is the minimum number of times that part or all of Equation 1 must be differentiated with respect to t in order to express $\dot{\mathbf{Z}}$ as a continuous function of $\mathbf{Z}$ and t [4, page 17].

By substituting a finite-difference approximation $\mathcal{D}_i\mathbf{Z}$ for $\dot{\mathbf{Z}}$, we obtain:

$$\mathbf{F}_{\mathcal{D}}(\mathbf{Z}_i; \tau_i) \equiv \mathbf{F}(\mathbf{Z}_i, \mathcal{D}_i\mathbf{Z}_i, \mathbf{u}_i; t = \tau_i) = \mathbf{0}, \tag{2}$$

a set of (in general) nonlinear *staticized* equations. A sequence of Equation 2's will have to be solved, one at each discrete time $t = \tau_i$, $i = 1, 2, \ldots, M$[1], in the numerical approximation scheme; neither M nor the τ_i's need be pre-determined. In *DASSL*, the variable step-size integration algorithm picks the τ_i's as the integration progresses, based on its assessment of the local error. The discretization operator for $\dot{\mathbf{Z}}$, $\mathcal{D}$, varies during the numerical integration process and hence is subscripted as $\mathcal{D}_i$.

The usual way to solve an instance of the staticized equations, Equation 2, is via the familiar Newton-Raphson iterative method (yielding $\mathbf{Z}_i \equiv \mathbf{Z}_i^{\infty}$):

$$\mathbf{Z}_i^{k+1} = \mathbf{Z}_i^k - c\left\{\nabla_{\mathbf{Z}}\mathbf{F}_{\mathcal{D}}(\mathbf{Z}_i^{m_k}; \tau_i)\right\}^{-1}\mathbf{F}_{\mathcal{D}}(\mathbf{Z}_i^k; \tau_i), \quad k = 0, 1, \ldots \tag{3}$$

given an initial, sufficiently good approximation $\mathbf{Z}_i^0$. The classical method is recovered for $m_k = k$ and $c = 1$, whereas a modified (damped) Newton-Raphson method results for $m_k < k$ (respectively, $c < 1$). In the original *DASSL* algorithm and in *Concurrent DASSL*, the Jacobian $\nabla_{\mathbf{Z}}\mathbf{F}_{\mathcal{D}}(\mathbf{Z})$ is computed by finite differences rather than analytically; this departure leads in another sense to a modified Newton-Raphson method even though $m_k = k$ and $c = 1$ might always be satisfied. For termination, a limit $k \leq k^*$ is imposed; a further stopping criterion of the form $\|\mathbf{Z}_i^{k+1} - \mathbf{Z}_i^k\| < \epsilon$ is also incorporated (see Brenan *et al.* [4, pages 121-124]).

Following Brenan *et al.*, the approximation $\mathcal{D}_i\mathbf{Z}$ is replaced by a BDF-generated linear approximation, $\alpha\mathbf{Z} + \beta$, and the Jacobian

$$\nabla_{\mathbf{Z}}\mathbf{F}(\mathbf{Z}, \alpha\mathbf{Z} + \beta, \mathbf{u}; t) = \frac{\partial \mathbf{F}}{\partial \mathbf{Z}} + \alpha\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{Z}}}. \tag{4}$$

From this approximation, we define $\mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i)$ in the intuitive way. We then consider Taylor's Theorem with remainder, from which we can easily express a forward finite-difference approximation for each Jacobian column (assuming sufficient smoothness of $\mathbf{F}_{\alpha,\beta}$) with a scaled difference of two residual vectors:

$$\mathbf{F}_{\alpha,\beta}(\mathbf{Z} + \delta_j; \tau_i) - \mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i) = \left\{\nabla_{\mathbf{Z}}\mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i)\right\}\delta_j + O\left(\|\delta_j\|_2^2\right) \tag{5}$$

By picking δ_j proportional to $\mathbf{e}_j$, the jth unit vector in the natural basis for $\Re^N$, namely $\delta_j = d_j\mathbf{e}_j$, Equation 5 yields a first-order-accurate approximation in d_j of the jth column of the Jacobian matrix:

$$\frac{\mathbf{F}_{\alpha,\beta}(\mathbf{Z} + \delta_j; \tau_i) - \mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i)}{d_j} = \left\{\nabla_{\mathbf{Z}}\mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i)\right\}\mathbf{e}_j + O(d_j), \quad j = 1, \ldots, N \tag{6}$$

Each of these N Jacobian-column computations is independent and trivially parallelizable. It's well known, however, that for special structures such as banded and block n-diagonal matrices, and even for general sparse matrices, a single residual can be used to generate multiple Jacobian columns [4,8]. We discuss these issues as part of the concurrent formulation section below.

The solution of the Jacobian linear system of equations is required for each k-iteration, either through a direct (*e.g.*, LU-factorization) or iterative (*e.g.*, preconditioned-conjugate-gradient) method. The most advantageous solution approach depends on N as well as special mathematical properties and/or structure of the Jacobian matrix $\nabla_{\mathbf{Z}}\mathbf{F}_{\mathcal{D}}$. Together, the inner (linear equation solution) and outer (Newton-Raphson iteration) loops solve a single time point; the overall algorithm generates a sequence of solution points $\mathbf{Z}_i$, $i = 0, 1, \ldots, M$.

In the present work, we restrict our attention to direct, sparse linear algebra as described in [13], although future versions of *Concurrent DASSL* will support the iterative linear algebra approaches by Ashby, Lee, Brown, Hindmarsh *et al.* [3,5]. For the sparse

[1] and more at trial timepoints which are discarded by the integration algorithm.

LU factorization, the factors are stored and reused in the modified Newton scenario. Then, repeated use of the old Jacobian implies just a forward and back-solve step using the triangular factors L and U. Practically, we can use the Jacobian for up to about five steps [4]. The useful lifetime of a single Jacobian evidently depends somewhat strongly on details of the integration procedure [4].

proto-Cdyn – Simulation Layer

To use the *Concurrent DASSL* system on other than toy problems, a simulation layer must be constructed above it. The purpose of this layer is to accept a problem specification from within a specific problem domain, and formulate that specification for concurrent solution as a set of differential-algebraic equations, including any needed data. On one hand, such a layer could explicitly construct the subset of equations needed for each processor, generate the appropriate code representing the residual functions, and create a set of node programs for effecting the simulation. This is the most flexible approach, allowing the user to specify arbitrary nonlinear DAE's. It has the disadvantage of requiring a lot of compiling and linking for each run in which the problem is changed in any significant respect (including but not limited to data distribution), although with sophisticated tactics, parametric variations within equations could be permitted without re-compiling from scratch, and incremental linking could be supported.

We utilize a template-based approach here, as we do in the Waveform-Relaxation paradigm for concurrent dynamic simulation [15]. This is akin to the *ASCEND II* methodology utilized by Kuru and many others [10]. It is a compromise approach from the perspective of flexibility; interesting physical prototype subsystems are encapsulated into compiled code as templates. A template is a conceptual building block with states, non-states, parameters, inputs and outputs (see below). A general network made from instantiations of templates can be constructed at runtime without changing any executable code. User input specifies the number and type of each template, their interconnection pattern, and the initial value of systemic states and extraneous (non-state) variables, plus the value of adjustable parameters and more elaborate data, such as physical properties. The addition of templates requires new subroutines for the evaluation of the residuals of their associated DAE's, and also for interfacing to the remainder of the system (*e.g.*, parsing of user input, interconnectivity issues). With suitable automated tools, this addition process can be made straightforward to the user.

Importantly, the use of a template-based methodology does not imply a degradation in the numerical quality of the model equations or solution method used. We are not obliged to tear equations based on templates or groups of templates as is done in sequential-modular simulators [19,6], where "sequential" refers in this sense to the stepwise updating of equation subsets, without connection to the number of computers assigned to the problem solution.

Ideally, the simulation layer could be made universal. That is, a generic layer of high flexibility and structural elegance would be created once and for all (and without predilection for a specific computational engine). Thereafter, appropriate templates would be added to articulate the simulator for a given problem domain. This is certainly possible with high-quality simulators such as *ASCEND II* and *Chemsim* (a recent Fortran-based simulator driving *DASSL* and *MA28* [2,11,7]). Even so, we have chosen to restrict our efforts to a more modest simulation layer, called *proto-Cdyn*, which can create arbitrary networks of coupled distillation columns. This restricted effort has required significant effort, and already allows us to explore many of the important issues of concurrent dynamic simulation. General-purpose simulators are for future consideration. They must address significant questions of user-interface in addition to concurrency-formulation issues.

In the next paragraphs, we describe the important features of *proto-Cdyn*. In doing so, we indicate important issues for any *Concurrent DASSL* driver.

Template Structure

A template is a prototype for a sequence of DAE's which can be used repeatedly in different instantiations. Normally, but not always, the template corresponds to some subsystem of a physical-model description of a system, like a tank or distillation tray. The key characteristics of a template are: the number of integration states it incorporates (typically fixed), the number of non-state variables it incorporates (typically fixed), its input and output connections to other templates, and external sources (forcing functions) and sinks. State variables participate in the overall *DASSL* integration process. Non-states are defined as variables which, given the states of a template alone, may be computed uniquely. They are essentially local tear variables. It is up to the template designer whether or not to use such local tear variables: They impact the numerical quality of the solution, in principle. Alternative formulations, where all variables of a template are treated as states, can be posed, and comparisons

made. Because of the superlinear growth of linear algebra complexity, the introduction of extra integration states must be justified on the basis of numerical accuracy. Otherwise, they artificially slow down the problem solution, perhaps significantly. Non-states are extremely convenient, and practically useful; they appear in all the dynamic simulators we have come across.

The template state and non-state structure implies a two-phase residual computation. First, given a state **Z**, the non-states of each template are updated on a template-by-template basis. Then, given its states and non-states, inputs from other templates and external inputs, each template's residuals may be computed. In the sequential implementation, this poses no particular nuisances, other than two evaluation loops over all templates. However, in concurrent evaluation, a communication phase intervenes between non-state updates and residual updates. This communication phase transmits all states and non-states appearing as outputs of templates to their corresponding inputs at other templates. This transmission mechanism is considered further below under concurrent formulation.

Problem Preformulation

In general, the "optimal" ordering for the equations of a dynamic simulation will in general be too difficult to establish[2], because of the NP-hard issues involved in structure selection. However, many important heuristics can be applied, such as those that precedence order the nonlinear equations, and those that permute the Jacobian structure to a more nearly triangular or banded form [8]. For the *proto-Cdyn* simulator, we skirt these issues entirely, because it proves easy to arrange a network of columns to produce a "good structure" – a main block tri-diagonal Jacobian structure with off-block-diagonal structure for the intercolumn connections, simply by taking the distillation columns with their states in tray-by-tray, top-down (or bottom-up) order.

Given a set of DAE's, and an ordering for the equations and states (*i.e.*, rows and columns of the Jacobian, respectively), we need to partition these equations between the multicomputer nodes, according to a two-dimensional process grid of shape $PxQ = R$. The partitioning of the equations forms, in main part, the so-called "concurrent database." This grid structure is illustrated in [13, Figure 2.]. In *proto-Cdyn*, we utilize a single process grid for the entire *Concurrent DASSL* calculation. That is, we don't currently exploit the *Concurrent DASSL* feature which allows explicit transformations between the main calculational phases (see below). In each process column, the entire set of equations is to be reproduced, so that any process column can compute not only the entire residual vector for a prediction calculation, but also, any column of the Jacobian matrix.

A mapping between the global equations and local equations must be created. In the general case, it will be difficult to generate a closed-form expression for either the global-to-local mapping or its inverse (that also require $< O(N)$ storage). At most, we will have on a hand a partial (or weak) inverse in each process, so that the corresponding global index of each local index will be available. Furthermore, in each node, a partial global-to-local list of indices associated with the given node will be stored in global sort order. Then, by binary search, a weak global-to-local mapping will be possible in each process. That is, each process will be able to identify if a global index resides within it, and the corresponding local index. A strong mapping for row (column) indices will require communication between all the processes in a process row (respectively, column). In the foregoing, we make the tacit assumption that is is an unreasonable practice to use storage proportional to the entire problem size N in each node, except if this unscalability can be removed cheaply when necessary for large problems.

The *proto-Cdyn* simulator works with templates of specific structure – each template is a form of a distillation tray and generates the same number of integration states. It therefore skirts the need for weak distributions. Consequently, the entire row mapping procedure can be accomplished using the closed-form general two-parameter distribution function family ξ described in [13], where the block size B is chosen as the number of integration states per template. The column mapping procedure is accomplished with the one-parameter distribution function family ζ also described in [13]. The effects of row and column degree-of-scattering are described in [13] with attention to linear algebra performance.

Concurrent Formulation

Overview

Next, we turn to Equation 1's (that is, **IVP**'s) concurrent numerical solution via the *DASSL* algorithm. We cover the major computational steps in abstract, and we also describe the generic aspects of *proto-Cdyn* in

[2] Optimality *per se* hinges on what our objective is. If, for instance, we want minimum time for LU factorization, still the objective of minimum fill-in does not guarantee minimum time in a concurrent setting.

this connection. In the subsequent section, we discuss issues peculiar to the distillation simulation.

Broadly, the concurrent solution of **IVP** consists of three block operations: startup, dynamic simulation, and a cleanup phase. Significant concurrency is apparent only in the dynamic simulation phase. We will assume that the simulation interval requested generates enough work so that the startup and cleanup phases prove insignificant by comparison and consequently pose no serious Amdahl's-law bottleneck. Given this assumption, we can restrict our attention to a single step of **IVP** as illustrated schematically in Figure 0.

In the startup phase, a sequential host program interprets the user specification for the simulation. From this it generates the concurrent database: the templates and their mutual interconnections, data needed by particular templates, and a distribution of this information among the processes that are to participate. The processes are themselves spawned and fed their respective databases. Once they receive their input information, the processes re-build the data structures for interfacing with *Concurrent DASSL*, and for generating the residuals. Tolerances, and initial derivatives must be computed and/or estimated. Furthermore, in each process column, the processes must rendezvous to finalize their communication labeling for the transmission of states and non-states to be performed during the residual calculation. This provides the basis for a reactive, deadlock-free update procedure described below.

The cleanup phase basically retrieves appropriate state values and returns them to the host for propagation to the user. Cleanup may actually be interspersed intermittently with the actual dynamic simulation. It provides simple bookkeeping of the results of simulation and terminates the concurrent processes at the simulation's conclusion.

The dynamic simulation phase consists of repetitive prediction and correction steps, and marches in time. Each successful time step requires the solution of one or more instances of Equation 2 – additional timesteps that converge but fail to satisfy error tolerances, or fail to converge quickly enough, are necessarily discarded. In the next section, we cover the aspects of these operations in more detail, for a single step.

Single Integration Step

The Integration Computations of *DASSL* are a fixed leading-coefficient, variable-stepsize and order, backward-differentiation-formula (BDF) implicit integration scheme, described clearly in [4, Chapter 5] and outlined in [11]. *Concurrent DASSL* faithfully implements this numerical method, with no significant differences. Test problems run with the *DASSL* Fortran code and the new C code (on one and multiple computers) certify this degree of compatibility.

The sequential time complexity of the integration computations is $O(N)$, if considered separately from the residual calculation called in turn, which is also normally $O(N)$ (see below). We pose these operations on a $PxQ = R$ grid, where we assume that each process column can compute complete residual vectors. Each process column repeats the entire prediction operations: there is no speedup associated with $Q > 1$, and we replicate all *DASSL* BDF and predictor vectors in each process column. Taller, narrower grids are likely to provide the overall greatest speedup, though the residual calculation may saturate (and slow down again) because of excessive vertical communication requirements — It's definitely not true that the $Rx1$ shape is optimal in all cases.

The distribution of coefficients in the rows has no impact on the integration operations, and is dictated largely by the requirements of the residual calculation itself. In practical problems, the concurrent database cannot be reproduced in each process (*cf.*, [18]), so a given process will only be able to compute some of the residuals. Furthermore, we may not have complete freedom in scattering these equations, because there will often be a tradeoff between the degree of scattering and the amount of communication needed to form the entire residual vector.

The amount of $O(N)$ integration-computation work is not terribly large — there is consequently a non-trivial but not tremendous effort involved in the integration computations. (Residual computations dominate in many if not most circumstances.) Integration operations consist mainly of vector-vector operations not requiring any interprocess communication and, in addition, fixed startup costs. Operations include prediction of the solution at the time point, initiation and control of the Newton iteration that "corrects" the solution, convergence and error-tolerance checking, and so forth. For example, the approximation $\mathcal{D}_i$ is chosen within this block using the BDF formulas. For these operations, each process column currently operates independently, and repetitively forms the results. Alternatively, each process column could stride with step Q, and row-*combines* could be used to propagate information across the columns [14]. This alternative would increase speed for sufficiently large problems, and can easily be implemented. However, because of load-imbalance in other stages of the calculation, we are convinced that including this type of synchroniza-

tion could be an overall negative rather than positive to performance. This alternative will nevertheless be a future user-selectable option.

Included in these operations are a handful of norm operations, which constitute the main interprocess communication required by the integration computations step; norms are implemented concurrently via recursive doubling (*combine*) [17,14]. Actually, the weighted norm used by *DASSL* requires two recursive doubling operations, each *combines* a scalar: first to obtain the vector coefficient of maximum absolute value, then to sum the weighted norm itself. Each can be implemented as Q independent column *combines*, each producing the same repetitive result, or a single Q-striding norm, that takes advantage of the repetition of information, but utilizes two *combines* over the entire process grid. Both are supported in *Concurrent DASSL*, although the former is the default norm. As with the original *DASSL*, the norm function can be replaced, if desired.

Single Residuals are computed in prediction, and as needed during correction. Multiple residuals are computed when forming the finite-difference Jacobian. Single residuals are computed repetitively in each process column, whereas the multiple residuals of a Jacobian computation are computed uniquely in the process columns.

Here, we consider the single residual computation required by the integration computations just described. Given a state vector $\mathbf{Z}$, and approximation for $\dot{\mathbf{Z}}$, we need to evaluate $\mathbf{F}(\mathbf{Z}, \dot{\mathbf{Z}}, \tau_i) \equiv \mathbf{F}_{\mathcal{D}}(\mathbf{Z}, \tau_i)$. The exploitable concurrency available in this step is strictly a function of the model equations. As defined, there are N equations in this system, so we expect to use at best N computers for this step. Practically, there will be interprocess communication between the process rows, corresponding to the connectivity among the equations. This will place an upper limit on $P \leq K$ (the number of row processes) that can be used before the speed will again decrease: we can expect efficient speedup for this step provided that the cost of the interprocess communication is insignificant compared to the single-equation grain size. As estimated in [14], the granularity T_{comm}/T_{calc} for the Symult s2010 multicomputer is about fifty, so this implies about four hundred and fifty floating point operations per communication in order to achieve 90% concurrent efficiency in this phase.

Jacobian Computation There is evidently much more available concurrency in this computational step than for the single residual and integration operations, since, for finite differencing, N independent residual computations are apparently required, each of which is a single-state perturbation of $\mathbf{Z}$. Based on our overview of the residual computation, we might naively expect to use $K \times N$ processes effectively; however, the simple perturbations can actually require much less model evaluation effort because of latency [8,10], which is directly a function of the sparsity structure of the model equations, Equation 1. In short, we can attain the same performance with much less than $K \times N$ processors.

In general, we'd like to consider the Jacobian computation on a rectangular grid. For this, we can consider using $P \times Q = R$ to accomplish the calculation. With a general grid shape, we exploit some concurrency in *both* the column evaluations and in the residual computations, with $T_{Jac,PxQ=R}$ the time for this step, $S_{Jac,PxQ=R}$ the corresponding speedup, $T_{res,P}$ the residual evaluation time with P row processes, and $S_{res,P}$ the apparent speedup compared to one row process:

$$T_{Jac,PxQ=R} \approx \lceil N/Q \rceil \times T_{res,P}$$
$$S_{Jac,PxQ=R} \approx \frac{N}{\lceil N/Q \rceil} \times S_{res,P}$$

assuming no shortcuts are available as a result of latency. This timing is exemplified in the example below, which does not take advantage of latency.

There is additional work whenever the Jacobian structure is rebuilt for better numerical stability in the subsequent LU factorization (A-mode). Then, $O(N^2/PQ)$ work is involved in each process in the filling of the initial Jacobian. In the normal case, work proportional to the number of local non-zeroes plus fill elements is incurred in each process for re-filling the sparse Jacobian structure.

Exploitation of Latency has been considered in the *Concurrent DASSL* framework. We currently have experimental versions of two mechanisms, both of which are designed to work with the sparse-matrix structures associated with direct, sparse LU factorization (see [13]). The first is called "bandlike" Jacobian evaluation. For a banded Jacobian matrix of bandwidth b, only b residuals are needed to evaluate the Jacobian. This feature is incorporated into the original *DASSL*, along with a LINPACK banded solver. In *Concurrent DASSL*, collections of Jacobian columns are placed in each process column, according to the column data distribution, which thus far is picked solely to balance LU factorization and triangular-solve performance [13]. In each process column, there will be

"compatible" columns that can be evaluated using a single, composite perturbation. Identification of these compatible columns is accomplished by checks on the bandwidth overlap condition. Columns that possess off-band structure are stricken from the list and evaluated separately. Presumably, a heuristic algorithm could be employed further to increase the size of the compatible sets, but this is yet to be implemented. The same algorithm "greedy" algorithm of Curtis *et al.* used for the sequential reduction of Jacobian computation effort would be applied independently to each process column (see comments by [8, Section 12.3]). Then, clearly, the column distribution effects the performance of the Jacobian computation, and the linear-algebra performance can no longer be viewed so readily in isolation.

We have also devised a "blocklike" format, which will be applied to block n-diagonal matrices that include some off-block entries as well. Optimally, fewer residual computations will be needed than for the banded case. The same column-by-column compatible sets will be created, and the Curtis algorithm can also be applied. Hopefully, because of the less restrictive compatibility requirement, the "blocklike" case will produce higher concurrent speedups than that attained using the conservative bandlike assumption for Jacobians possessing blocklike structure. Comparative results will be presented in a future paper.

The LU Factorization Following the philosophy of Harwell's *MA28*, we have interfaced a new concurrent sparse solver to *Concurrent DASSL*, the details of which are quoted elsewhere in this proceedings [13]. In short, there is a two-step factorization procedure: A-mode, which chooses stable pivots according to a user-specified function, and builds the sparse data structures dynamically; and B-mode, which re-uses the data structures and pivot sequence on a similar matrix, but monitors stability with a growth-factor test. A-mode is repeated whenever necessary to avoid instability. We expect sub-cubic time complexity and sub-quadratic space complexity in N for the sparse solver. We attain acceptable factorization speedups for systems that are not narrow banded, and of sufficient size. We intend to incorporate multiple pivoting heuristic stategies, following [1], further to improve performance of future versions of the solver. This may also contribute to better performance of the triangular solves.

Forward- and Back-solving Steps take the factored form

$$\mathsf{P}_R A \mathsf{P}_C^T = \hat{L}\hat{U},$$

with $\hat{L}$ unit lower-triangular, $\hat{U}$ upper-triangular, and permutation matrices P_R, P_C, and solve $Ax = b$, using the implicit pivoting approach described in [13]. Sequentially, the triangular solves each require work proportional to the number of entries in the respective triangular factor, including fill-in. We have yet to find an example of sufficient size for which we actually attain speedup for these operations, at least for the sparse case. At most, we try to prevent these operations from becoming competitive in cost to the B-mode factorization; we detail these efforts in [13]. In brief, the optimum grid shape for the triangular solves has $Q = 1$, and P somewhat reduced than what we can use in all the other steps. As stated, P small seems better thus far, though for many examples, the increasing overhead as a function of increasing P is not unacceptable (see [13] and the example below).

Residual Communication is an important aspect of the *proto-Cdyn* layer. As indicated in the startup-phase discussion, the members of a process column initially share information about the groups of states and non-states they will exchange during a residual computation. For residual communication, a reactive transmission mechanism is employed, to avoid deadlocks. Each process transmits its next group of states to the appropriate process and then looks for any receipt of state information. Along with the state values are indices that directly drive the destinations for these values. This index information is shared during the startup phase and allows the messages to drive the operation. Through non-blocking receives, this procedure avoids problems of transmission ordering. Regardless of the template structure, at most one send and receive is needed between any pair of column processes.

Chemical Engineering Example

The algorithms and formalism needed to run this example amount to about 70,000 lines of C code including the simulation layer, *Concurrent DASSL*, the linear algebra packages, and support functions [14,13,12].

In this simulation, we consider seven distillation columns arranged in a tree-sequence [12], working on the distillation of eight alcohols: methanol, ethanol, propan-1-ol, propan-2-ol, butan-1-ol, 2-methyl propan-1-ol, butan-2-ol, and 2-methyl propan-2-ol. Each column has 143 trays. Each tray is initialized to a non-steady condition, and the system is relaxed to the steady state governed by a single feed stream to the first column in the sequence. This setup

generates suitable dynamic activity for illustrating the cost of a single "transient" integration step.

We note the performance in Table 0. Because we have not exploited latency in the Jacobian computation, this calculation is quite expensive, as seen for the sequential times on a Sun 3/260 depicted there. (The timing for the Sun 3/260 is quite comparable to a single Symult s2010 node and was lightly loaded during this test run.) As expected, Jacobian calculations speedup efficiently, and we are able to get approximately a speedup of 100 for this step using 128 nodes. The A-mode linear algebra also speeds up significantly. The B-mode factorization speeds up negligibly and quickly slows down again for more than 16 nodes. Likewise, the triangular solves are significantly slower than the sequential time. It should be noted that B-mode reflects two orders of magnitude speed improvement over A-mode. This reflects the fact that we are seeing almost linear time complexity in B-mode, since this example has a narrow block tri-diagonal Jacobian with too little off-diagonal coupling to generate much fill-in. It seems hard to imagine speeding up B-mode for such an example, unless we can exploit multiple pivots. We expect multiple-pivot heuristics to do reasonably well for this case, because of its narrow structure, and nearly block tri-diagonal structure. We have used Wilson Equation Vapor-Liquid Equilibrium with the Antoine Vapor equation. We have found that the thermodynamic calculations were much less demanding than we expected, with bubble-point computations requiring "1+ϵ" iterations to converge. Consequently, there was not the greater weight of Jacobian calculations we expected beforehand. Our model assumes constant pressure, and no enthalpy balances. We include no flow dynamics and include liquid and vapor flows as states, because of the possibility of feedbacks.

Were we to utilize latency in the Jacobian calculation, we could reduce the sequential time by a factor of about 100. This improvement would also carry through to the concurrent times for Jacobian solution. At that ratio, Jacobian computation to B-mode factorization has a sequential ratio of about 10:1. As is, we achieve legitimate speedups of about five. We expect to improve these results using the ideas quoted elsewhere here and in [13].

From a modeling point-of-view, two things are important to note. First, the introduction of more non-ideal thermodynamics would improve speedup, because these calculations fall within the Jacobian computation phase and Single-Residual Computation. Furthermore, the introduction of a more realistic model will likewise bear on concurrency, and likely improve it. For example, introducing flow dynamics, enthalpy balances and vapor holdups makes the model more difficult to solve numerically (higher index). It also increases the chance for a wide range of step-sizes, and the possible need for additional A-mode factorizations to maintain stability in the integration process. Such operations are more costly, but also have a higher speedup. Furthermore, the more complex models will be less likely to have near diagonal dominance; consequently more pivoting is to be expected, again increasing the chance for overall speedup compared to the sequential case. Mainly, we plan to consider the Waveform-Relaxation approach more heavily, and also to consider new classes of dynamic distillation simulations with *Concurrent DASSL* [12].

Conclusions

We have developed a high-quality concurrent code, *Concurrent DASSL*, for the solution of ordinary differential-algebraic equations of low index. This code, together with appropriate linear algebra and simulation layers, allows us to explore the achievable concurrent performance of non-trivial problems. In chemical engineering, we have applied it thus far to a reasonably large, simple model of coupled distillation columns. We are able to solve this large problem, which is quite demanding on even a large mainframe because of huge memory requirements and non-trivial computational requirements; the speedups achieved thus far are legitimately at least five, when compared to an efficient sequential implementation. This illustrates the need for improvements to the linear algebra code, which are feasible because sparse matrices will admit multiple pivots heuristically. It also illustrates the need to consider hidden sources of additional time-like concurrency in *Concurrent DASSL*, perhaps allowing multiple right-hand sides to be attacked simultaneously by the linear algebra codes, and amortizing their cost more efficiently. Furthermore, the performance points up the need for detailed research into the novel numerical techniques, such as Waveform Relaxation, which we have begun to do as well [15].

Acknowledgements

The first author acknowledges the kind assistance and helpful cooperation of Lionel F. Laroche and Henrik W. Andersen in the area of dynamic simulation for chemical process flowsheets. We have spent many hours together over the last twenty months in the discussion of design goals, features, algorithms, on realizations, post-mortems and re-designs, and in over-

Table 0. Order 9009 Dynamic Simulation Data					
	(time in seconds)				
Grid Shape	Jacobian	A-mode	B-mode	Back-Solve	Solve
1x1	64672.2	5089.96	61.82	2.5	4.7
8x1	6870.82	1024.41	47.827	15.619	30.825
16x1	3505.13	547.625	52.402	19.937	39.491
32x1	1829.93	316.544	56.713	24.383	47.692
64x1	1060.40	219.148	77.302	39.942	59.553
32x4	491.526	181.082	71.482	57.049	101.994
64x2	520.029	161.052	82.696	46.013	86.935
128x1	608.946	170.022	90.905	37.498	67.982

Key single-step calculation times with the 1x1 case run an unloaded Sun 3/260 (similar performance-wise to a single Symult s2010 node) for comparison. The Jacobian rows were distributed in block-linear form, with $B = 9$, reflecting the distillation-tray structure. The Jacobian columns were scattered. This is an seven column simulation of eight alcohols, with a total of 1,001 trays. See [13] for more on data distributions.

coming the stumbling blocks in our respective simulation codes. Thanks also to Prof. A. W. Westerberg of CMU, who offered helpful suggestions when he visited Caltech in 1989.

Thanks to Drs. K. E. Brenan, S. L. Campbell and Linda Petzold, for sharing advance drafts of their monograph *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations,* which proved very helpful in the creation of *Concurrent DASSL.*

The first author acknowledges partial support under DOE grants DE-FG03-85ER25009 and DE-AC03-85ER40050.

Concurrent DASSL was developed using machine resources made available by the Caltech Computer Science sub-Micron System Architectures Project and the Caltech Concurrent Supercomputer Facilities (CCSF).

References

[1] G. Alaghband. Parallel pivoting combined with parallel reduction and fill-in control. *Parallel Computing,* 11:201–221, 1989.

[2] H. W. Andersen and L. F. Laroche, 1988–1990. — Private Communications on *Chemsim.*

[3] S. Ashby, 1990. — Private Communication on *Iterative DASSL.*

[4] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations.* North Holland Elsevier, 1989.

[5] P. N. Brown and A. C. Hindmarsh. Reduced storage matrix methods in stiff ODE systems. *J. Appl. Math. & Comp.,* (to appear).

[6] W. J. Cook. A modular dynamic simulator for distillation systems. Master's thesis, Case Western Reserve University, 1980. Chemical Engineering.

[7] I. S. Duff. MA28 – a set of fortran subroutines for sparse unsymmetric linear equations. Technical Report R8730, AERE, HMSO, London, 1977.

[8] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, 1986.

[9] C. A. R. Hoare. Communicating sequential processes. *CACM,* 21(8):666–677, August 1978.

[10] S. Kuru. *Dynamic Simulation with an Equation Based Flowsheeting System.* PhD thesis, Carnegie Mellon University, 1981. Chemical Engineering Department.

[11] L. R. Petzold. DASSL: Differential algebraic system solver. Technical Report Category #D2A2, Sandia National Laboratories — Livermore, 1983.

[12] A. Skjellum. *Concurrent Dynamic Simulation: Multicomputer Algorithms Research Applied to Differential-Algebraic Process Systems in Chemical Engineering.* PhD thesis, California Institute of Technology, May 1990. Chemical Engineering.

[13] A. Skjellum and A. P. Leung. LU factorization of sparse, unsymmetric jacobian matrices on multicomputers: *Experience, Strategies, Performance.* In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5). in press*, April 1990.

[14] A. Skjellum and A. P. Leung. *Zipcode:* a portable multicomputer communication library atop the *reactive kernel.* In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5). in press*, April 1990.

[15] A. Skjellum, M. Morari, and S. Mattisson. Waveform Relaxation for Concurrent Dynamic Simulation of Distillation Columns. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications (HCCA3)*, pages 1062–1071. ACM Press, January 1988.

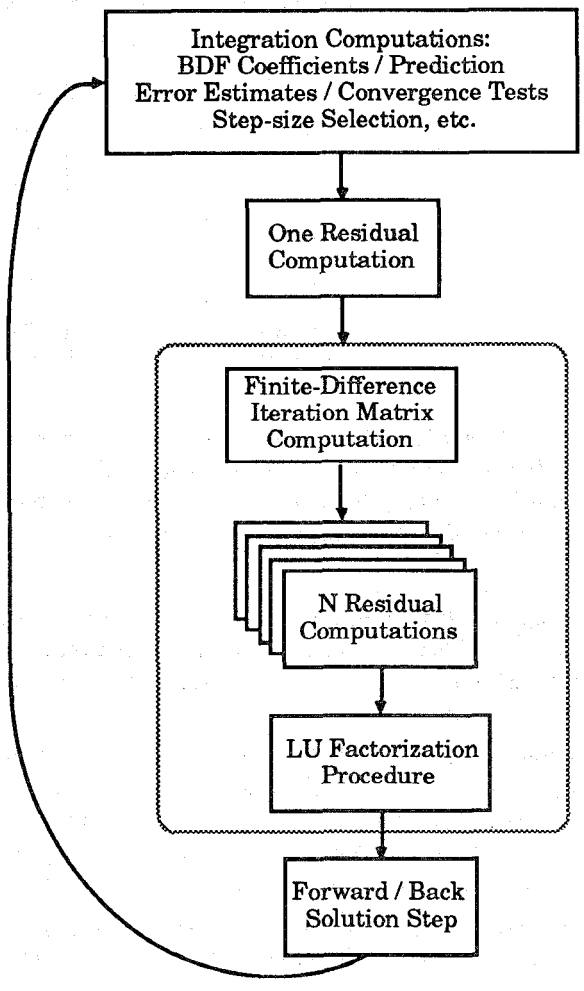
[16] A. Skjellum, M. Morari, S. Mattisson, and L. Peterson. *Concurrent DASSL:* Structure, Application, and Performance. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications (HCCA4)*, pages 1321–1328. Golden Gate Enterprises, March 1989. Simulation Minisymposium.

[17] H. S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, 1987.

[18] E. F. Van de Velde and J. Lorenz. Adaptive data distribution for concurrent continuation. Technical Report CRPC-89-4, California Institute of Technology, 1989. Caltech/Rice Center for Research in Parallel Computation.

[19] A. W. Westerberg, H. P. Hutchison, R. L. Motard, and P. Winter. *Process flowsheeting.* Cambridge University Press, 1979.

Figure 0. Major computational blocks of a Single Integration Step.



A single step in the integration begins with a number of BDF-related computations, including the solution "prediction" step. Then, "correction" is achieved through Newton iteration steps, each involving a Jacobian computation, and linear-system solution (LU factorization plus forward- / back-solves). The computation of the Jacobian in turn relies upon multiple independent residual calculations, as shown. The three items enclosed in the dashed oval (Jacobian computation (through at-most N Residual computations), and LU factorization) are, in practice, computed less often than the others — the old Jacobian matrix is used in the iteration loop until convergence slows intolerably.