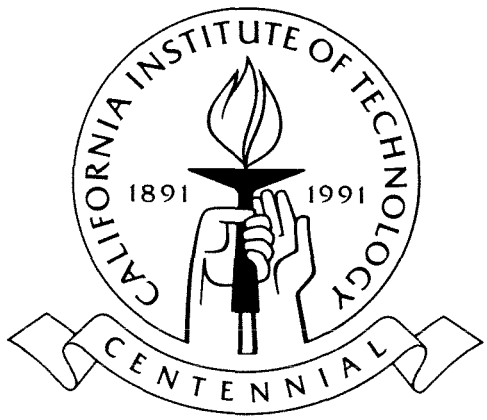
CONCURRENT DYNAMIC SIMULATION:
MULTICOMPUTER ALGORITHMS RESEARCH APPLIED TO ORDINARY DIFFERENTIAL-ALGEBRAIC PROCESS SYSTEMS IN CHEMICAL ENGINEERING

Dissertation by
Anthony Skjellum

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Division of Chemistry & Chemical Engineering
Pasadena, California

1990
(Submitted May 21, 1990)



This work is dedicated respectfully to the memory of

Professor William H. Corcoran,

and also in memory of

Professor William F. Bergquist,

Roland A. Haugh,

and

John Manley

for knowledge, ideas and ideals...

Sonnet LX

Like as the waves make toward the pebbled shore,
So do our minutes hasten to their end;
Each changing place with that which goes before,
In sequent toil all forwards do contend.
Nativity, once in the main of light,
Crawls to maturity, wherewith being crown'd,
Crooked ellipses 'gainst his glory fight,
And Time that gave doth now his gift confound.
Time doth transfix the flourish set on youth
And delves the parallels in beauty's brow,
Feeds on the rarities of nature's truth,
And nothing stands but for his scythe to mow;
    And yet to times in hope my verse shall stand,
    Praising thy worth despite his cruel hand.

— Shakespeare

Sonnet XIX

When I consider how my light is spent,
  Ere half my days, in this dark world and wide,
  And that one Talent which is death to hide,
  Lodg'd with me useless, though my Soul more bent
To serve therewith my Maker, and present
  My true account, lest he returning chide;
  "Doth God exact day-labor, light denied,"
  I fondly ask; But patience to prevent
That murmur, soon replies, "God doth not need
  Either man's work or his own gifts; who best
  Bear his mild yoke, they serve him best; his State
Is Kingly. Thousands at his bidding speed
  And post o'er Land and Ocean without rest:
  They also serve who only stand and wait."

— Milton

Acknowledgements

Many years have passed since I began my studies at Caltech... more than a decade. Respectfully, I wish to acknowledge collectively all the people, whether mentioned below or not, who have helped me and/or became friends along the way.

Foremost, I wish to thank my advisor Prof. Manfred Morari. His guidance, alternate prodding and patience during the uncertain, difficult "middle years" are all deeply appreciated. His "conceptual-block-busting" approach to research has taught me much about what it means to be a researcher. For this, and for the opportunity he gave me to study toward my Ph.D. at Caltech, I am grateful. I am also thankful to Profs. John Seinfeld and George Gavalas for allowing me to continue in the Ph.D. program after my M.S. degree in Chemical Engineering, and for the 1985 Rockwell Fellowship they awarded me in this connection.

Equally, I wish to thank my parents for their unwavering spiritual (and financial) support, selflessness and kindness over my years at Caltech. They are undoubtedly the two pillars of strength, without which my studies would never have been as successful. Their "support system" to me during my last months has made an otherwise tension-packed time livable.

Regards also to Profs. Fred Shair, Joel Franklin and Charles Seitz for their encouragement and moral support over many years, and for their general suggestions toward my research as the members of my thesis committee. I am also indebted to Dr. Paul Messina for his many helpful suggestions as the fifth member of my committee, who participated meaningfully despite myriad commitments.

* * *

I wish heartily to thank Mr. Lee F. Browne, Director of Secondary School Relations and Special Student Programs. Through his intensive summer courses in chemistry, physics, and calculus, I gained the proficiency needed to begin as a Caltech undergraduate; I also remember the support of Profs. Tom Apostol and Jerry Pine for my admission to Caltech. I especially wish to remember Prof. Ricardo Gomez for his friendship, good advice, and for the hospitality he and his wife Clara extended over the years, including "serious" croquet matches, a tradition I try to carry on. I wish to thank Yekta Gürsel for his herculean efforts as physics teaching assistant nonpareil, tutor at-large, and for his friendship.

I wish to ackowledge Prof. Eugene Cowan for his valuable advice as my second undergraduate advisor, after the untimely death of Prof. William Corcoran. Prof. Cowan's extensive assistance with my experimental senior thesis was extremely kind and beyond all reasonable expectations; this work proved an important lesson in the difficulty of research. I wish to acknowledge Prof. Geoffrey Fox for his support of my 1983 SURF project, and for his help and advice in those years.

I wish to acknowledge former Caltech Profs. Eric Herbolzheimer and Greg Stephanopoulos for the best taught courses in engineering, ChE 103abc, *Transport Phenomena*. They really made an effort to make the topics understandable and interesting. Special thanks to former Prof. Jens Lorenz for his excellent Applied Math courses in optimization and numerical methods for PDE's. These were far and away the best teachers I encountered during my ten plus years at Caltech.

I wish to thank Prof. Charles Seitz for his efforts in creating the multicomputer revolution and for practical concurrent systems software and machines, without which this research could only be hypothetical. I am grateful to Sven Mattisson ($\bowtie$) and Lena Peterson ($\smile$) of Lund Institute of Technology, Dept. of Applied Electronics (and irregularly of Caltech Computer Science) for their extensive collaboration, *CONCISE* simulation software, and continuing dialogue. Eric Van de Velde kindly shared

his dense and prototype sparse concurrent linear algebra software and multicomputer programming ideas with me through his AMa/CS course, serving as springboards for some of the concurrent numerical libraries I have created. Alvin Leung contributed genuinely to this research through his efforts as a 1989 Caltech Summer Undergradute Research Fellow (SURF).

Wen-King Su and Chris Lee (of Computer Science) have been instrumental in surmounting difficulties with the Symult s2010 multicomputer, from a software point-of-view. Sharon Brunett of Caltech Concurrent Supercomputer Facilities (CCSF) has complementarily been a tremendous help in the systems and hardware aspects of using this multicomputer. Each of them contributed importantly, and often in very timely ways, to the furtherance of this work. Arlene DesJardins was very helpful and patient in Sun-usage and disk-consumption-related issues. Thanks guys!

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 the *Concurrent DASSL* system described herein (chapter 6).

Thanks also to Prof. Michael F. Doherty of the University of Massachusetts, Amherst, who kindly shared his group's Fortran-based thermodynamic routines and database with the Morari research group. From this code, I developed a numerically compatible C version used by the *proto-Cdyn* multicomputer simulator also described in chapter 6.

Profs. Gustaf Söderlind and Sven Mattisson (both of Lund Institute of Technology) offered timely, helpful input concerning the underlying numerical properties of *CONCISE*, which is essential to the contents of appendix E, and to our future planned extensions of Waveform Relaxation to chemical engineering problems.

Dr. John Gustafson of Ames Laboratory-ISU (formerly of Sandia National Laboratories), kindly shared the machine-readable form of his well-known scaled performance

diagrams, some of which appear in the supplementary discussion on performance in appendix F.-

I wish to acknowledge partial support under the following DOE contracts: DE-FG03-85ER25009 and DE-AC03-85ER40050, representing one-quarter of my total graduate support over the six years, and including additional travel support. Thanks to Geoffrey Fox for making this money available to my advisor expressly for my support. I acknowledge the Caltech Computer Science Submicron System Architectures Project and CCSF for making multicomputer resources and host machines available.

* * *

Turning to the Morari research group and the department... The friendship and advice of my officemates Profs. Claudio Scali and Sigurd Skogestad during their salad days at Caltech are recalled fondly. My officemates Lionel ("gee-guys") Laroche, Jay ("oh-no") Lee, and Eliana ("M-A-K-H-L-O-U-F as in Frank") Makhlouf, have been continual sources of fun and excitement since their arrival in the office in 1986-87. Jay and Lionel are now "senior group members," with the awful burden of upholding the best traditions of the group (and holding down the rest). Thanks also to Jay for many full-contact *table tennistsu* matches.

I wish to recall the significant help and comraderie of Henrik Andersen, Lionel Laroche and Jay Lee, and the friendship of Dan Laughlin, Richard Colberg, Pierre Grosdidier, and Marc Gelormino (who also endured many enervating, *High Speed* pinball tourneys). Finally, Evanghelos ("EZ") Zafiriou and Dan Rivera were irresponsible for including me in their running battles on the nature of life, the universe, artificial intelligence, evolution, religion, mysticism, psychology, the opposite sex, *STAR TREK*, politics, robust control, Vanna White ($\smile$), and obscure nonlinear combinations of these topics. Their repartee evidently continues to-date, mainly unresolved, and occasionally re-implicating Jorge Mandler, to his utter chagrin.

* * *

As far as Caltech at large, thanks to Cindy Akutagawa for many years of friendship, *many* thanks to Carol Mastin and her Graduate Office co-workers for their strong support of my multi-year activities involving the Graduate Student Council (GSC), including three editions of *The Technique*, and for help in my graduation process; thanks to Stan Borodinsky for GSC fiscal help, to Edith Huang for *troff*-related help in the dim times; grateful thoughts to Donna Johnson, April Olson, Kathy Lewis, Pat Houseworth, Helen Dewitt and Christina Conti, in the department, Carolyn Merkel of SURF, and Sylvia Ford, Barbara Anderson, and Fred Kemp *et al.* at the Bookstore, all for many little instances of assistance. The staff of Millikan library's Inter-library Loan Office have assisted me throughout my years at Caltech; Ken Sweet and his staff of the Millikan bindery service have been extremely helpful, conscientious, and resourceful. Thanks to everyone at the Caltech Alumni Association and fellow members of its Board of Directors for their well wishes and sincere friendship, especially Gary Stupian, Rhonda MacDonald, and Arlana Bostrom. I extend fond wishes to my teammates from the last three Divergent Grads (∇^2) C-league softball teams.

Special thanks to LGL for visiting Hawaii in June, 1985.

Concurrent Dynamic Simulation:
Multicomputer Algorithms Research Applied to Ordinary Differential-Algebraic Process Systems in Chemical Engineering

by

Anthony Skjellum

Abstract

We consider systematic parallel solution of ordinary differential-algebraic equations (DAE's) of low index (including stiff ODE's). We target multicomputers, message-passing concurrent computers, such as Intel's iPSC/2 hypercube and the Symult s2010 2D mesh. The programming model is reactive and/or loosely synchronized communicating sequential processes.

We present new approaches to efficient application-level message passing through the *Zipcode* communication layer (built upon the Caltech *Reactive Kernel*), which is shown to be both portable and effective for complex multicomputer codes. *Zipcode* promotes the elegant expression of message passing in large applications, an important sub-goal.

We present closed-form $O(1)$-memory, $O(1)$-time data distributions providing parametric control over degree of coefficient blocking and scattering. These new distributions permit effective formulations of the DAE's and higher sparse linear algebra performance.

We present results for concurrent sparse, unsymmetric linear algebra. A two-phase approach is used, like Harwell's *MA28*. New results include: reduced communication

pivoting and improvement of triangular-solve performance via the parametric distributions: LU-factorization load balance is traded against solve performance. Overall performance is thereby increased. Good factorization speedups are attained for examples, but exploitation of multiple concurrent pivots remains a needed extension. Triangular solves prove disappointing on an absolute scale, despite significant effort.

Two approaches to concurrent simulation are developed: the Waveform Relaxation (Picard-Lindelöf) methodology extends to binary distillation simulation and further; it is inherently very concurrent. We address the achievable concurrent performance of sequential approaches via *Concurrent DASSL*, which extends Petzold's *DASSL* algorithm to multicomputers. A simulation driver for arbitrary networks of distillation columns is described. For a 9009-integration-state system with seven distillation columns, we demonstrate a speedup of approximately five. The low speedup is attributable to the simplicity of the thermodynamic model used, and the nearly narrow-banded Jacobian structure. Other chemical-engineering systems could perform substantially better.

We suggest Waveform Relaxation as the key focus of future research for the particular distillation problem class cited. We indicate future areas for application of *Concurrent DASSL*, and suggest ways to improve its concurrent performance, coupled with improvements in sparse linear algebra.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Abstract

This introductory chapter is divided roughly into three parts: first, a motivating discussion of concurrent computation and multicomputers, and, second, a discussion of the potential applications and benefits of, and research challenges to highly concurrent computation in chemical engineering. Third, we outline the contents of the thesis, indicate the temporal progression of the research summarized here, and indicate further background and motivations for this work.

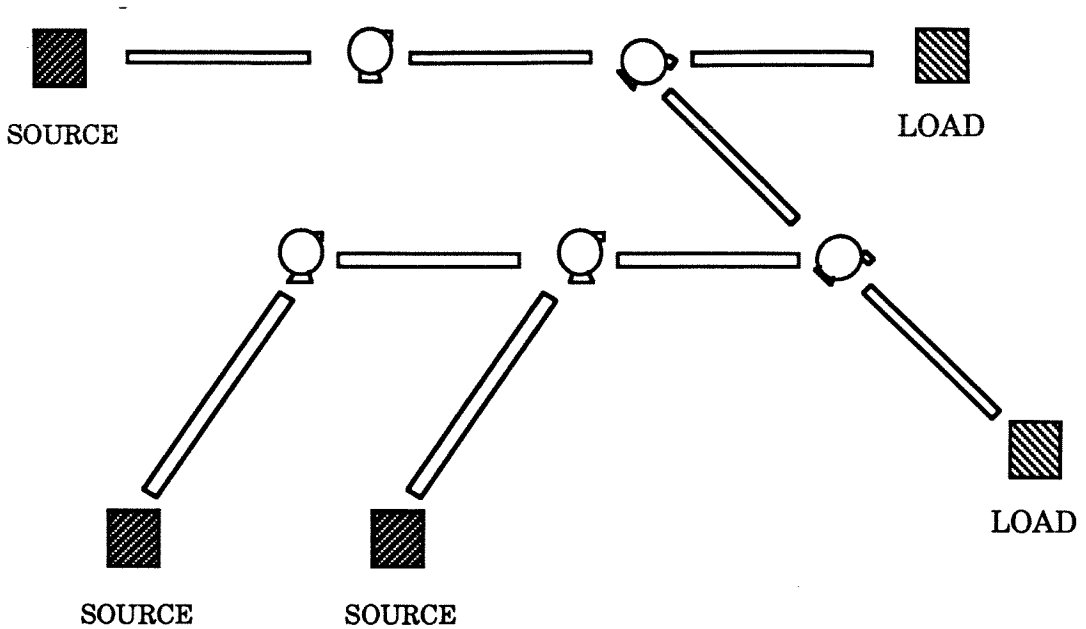
1.1 Concurrent Computation Research for Chemical Engineering

The topic of this thesis is large-scale, concurrent dynamic simulation in chemical engineering. We have centered on distillation column networks as the application, but high-performance solution of this unit operation is but our exemplar, not our exclusive end-goal. We are intent rather on opening a new area of chemical-engineering research.

1.1.1 Motivations and Philosophy

We see high-performance computation research as an important intellectual activity in chemical engineering that builds on the strong extant chemical-engineering tra-

Figure 1.1. Schematic Natural Gas Pipeline Distribution Network



ditions of modeling and simulation, and on the application of applied mathematics, computer science, electrical/computer engineering, and computer-aided design to important chemical-engineering problem domains. We see the current need for such high-performance computation in order to generate solutions for fundamental and practical investigations alike, and to help generate better understanding of chemical-engineering systems and models in the research environment. For the future, we expect these systems to function in on-line capacities also, yielding more efficient, economical operations in plants and other large-scale systems. In on-line applications, we imagine that concurrent supercomputer simulations and optimizations could conceivably account for significant improvements in operational flexibility, quality control, safety, emissions standards, competitiveness and, ultimately, profitability. Massive concurrency is arguably the most important source of such future high-performance computation, because it skirts important physical limitations of sequential devices.

These potential benefits will be impossible without careful research into concur-

rent computation and numerical algorithms. We must, however, simultaneously take the structure and features of chemical-engineering applications into account. As such, this is to be a genuine interdisciplinary activity, impractical without the motivation of chemical-engineering problems, and impossible without high-quality computer-science methodologies and applied-mathematical techniques. Finally, we must provide for technology transfer to applied research and to industry, in the form of effective concurrent software tools. Though algorithms research has been and is to remain the fundamental goal of our work, practical representations of our methods and findings are needed, not only to prove their supposed validity, but because it will undoubtedly prove impractical for commercial ventures to consider such costly and elaborate efforts on their own, at least for the foreseeable future. In effect, we must help lower the economic barriers to concurrent computing, as well as the conceptual barriers. It is clear that the cost of concurrent supercomputer hardware will be small compared to the true cost of the high-quality software needed to drive the hardware to high performance for non-trivial applications.

1.1.2 Applications in Chemical Engineering

We foresee a number of important applications for concurrent computers in chemical engineering: first, as reflected by the work of this thesis, in dynamic flowsheet simulations; second, in the combined economic optimization and simulation of such systems, eventually on-line. Furthermore, we see the application of concurrency research in fluid mechanics to the many fluid-mechanics problems that arise in chemical engineering, although we have not considered such problems in the present work. Thirdly, we imagine applying methods for simulation and optimization to systems that are themselves large-scale spatially, such as the trans-Canada gas pipeline. Specifically, the economic optimization of natural gas distribution networks is an important goal, and a concurrent supercomputer would provide a route to such improved performance (see

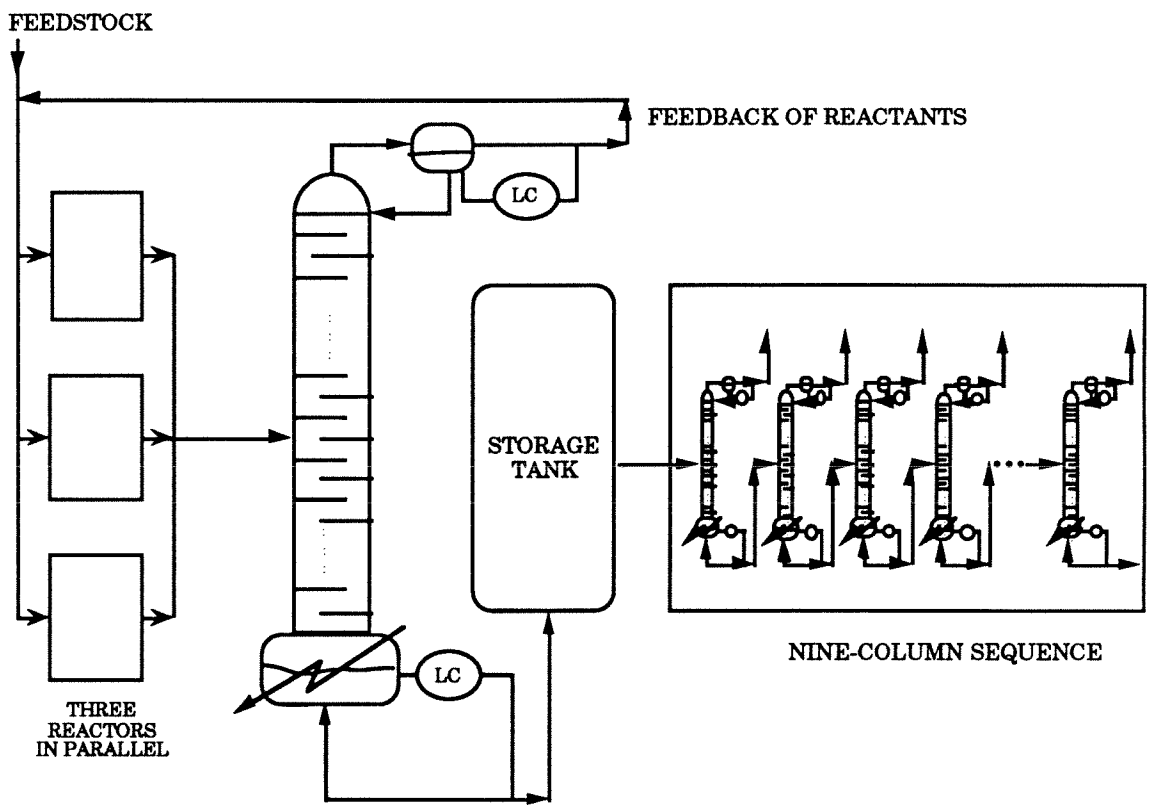
Figure 1.1). In an ideal situation, we would apply model-predictive receding-horizon control to the system, utilizing the simulations generated by the concurrent computer as the nonlinear model for the large-scale system (see [26,31]).

There are a large number of parameter-estimation applications arising in various specialties within chemical engineering; identification is also an important related area. For these fields, it would be interesting to investigate the potential impact of concurrency on present practice. In general, for control, the availability of faster and more detailed simulations is known to be of value. For example, our work leads the way to higher speed simulation of coupled distillation columns. The availability of faster accurate simulations will permit control studies for these systems. In particular, we are interested in solving an industrial problem as depicted in Figure 1.2. The main part of this example involves about five thousand integration states. We have already successfully solved problems with an excess of nine thousand states with our concurrent simulator (see chapter 6), although not of the complex structure depicted in the figure. We are therefore confident of our ability to address this coupled reactor-distillation sequence effectively within the next one to two years, based on current algorithms, software sophistication, and future planned work.

Our algorithms and software realizations are not limited to chemical engineering systems specifically, except for the distillation simulation driver *proto-Cdyn* detailed in chapter 6. The *Concurrent DASSL* integration engine (see chapter 6), and sparse concurrent linear solver (see chapter 5) discussed here are suitable for a number of applications. We imagine applying this technology to electrical power network grid simulations, and possibly directly to natural gas pipeline simulation, as suggested by Chua [13]. Petzold's original Fortran-based *DASSL* (upon which our concurrent C code is modeled) has also been applied extensively to combustion problems, and to many other systems involving ordinary differential-algebraic equations of low index, and including parabolic PDE's solved by the method of lines [9]. Our work opens

the possibility of parallelizing this large body of applications readily while maintaining the present, high-quality numerical characteristics of *DASSL*. Achievable performance for each application under this transformation to concurrency will vary. Higher performance will, of course, require special attention to problem characteristics and peculiarities, on a one-by-one basis.

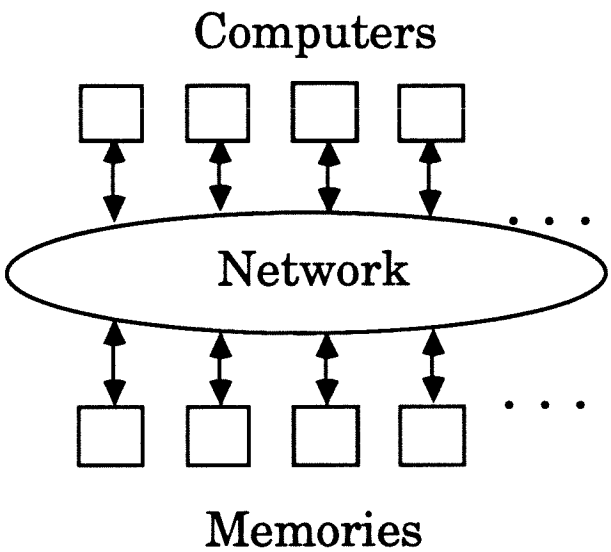
Figure 1.2. Coupled Reactor-Distillation Flowsheet



1.2 Concurrency & Multicomputers

The primary goal of this effort is to obtain *orders-of-magnitude* speedup in wall-clock execution time by using large, distributed-memory, medium-grain computer ensembles ("parallel processing") [40,41,6,20]. The secondary goal is to harness multicomputers to solve large problems that prove too large for effective solution on

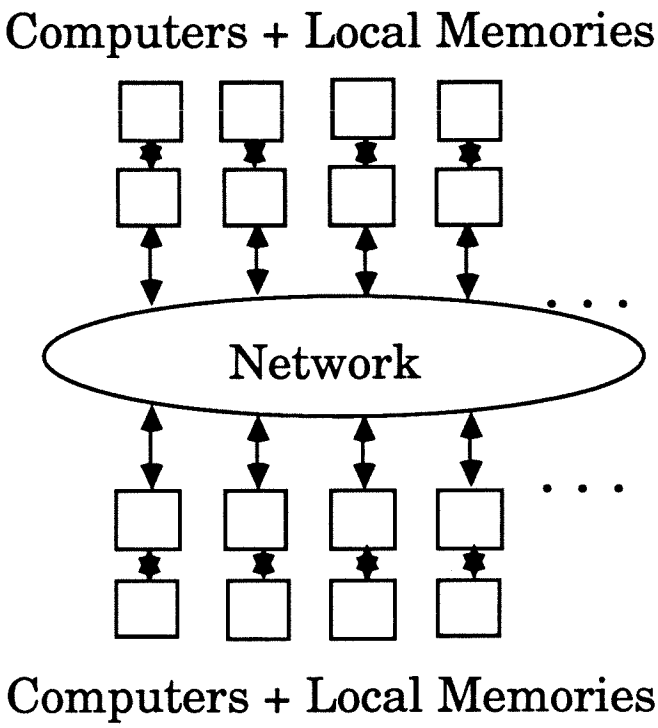
Figure 1.3. Multiprocessor Schematic



mainframe technology (see Figure 1.4.). Though solution speed is still important for the latter case, the higher priority is the ability to solve these previously out-of-reach problems at all. The availability of greater computational power should also drive the modelling process: engineers will be able to consider more complex, computationally demanding models than are currently feasible. The tertiary goal is the production of high-quality software realizations of our algorithms suitable for wider use and expansion. In this way we dramatically lessen the entry cost of further work in multicomputation for chemical engineering. We are largely *unconcerned* with the aggregate of CPU cycles used to accomplish these computational ends. Appropriate concurrent algorithms obtain cost-effective, high-end computing for specific problems and advance the understanding of how distributed-memory, message-passing multicomputers can be efficiently applied to additional complex engineering applications.

Present technology offers hardware *scalability* to large ensembles and currently available commercial products incorporate up to 2,048 computational nodes [25], each with workstation power. These machines support medium-grain computational tasks,

Figure 1.4. Multicomputer Schematic



requiring nominally fifty to six hundred floating-point operations between communications for fifty-percent efficiency. Extant high-end machines offer one gigabyte of random-access storage (distributed among the ensemble nodes), 100,000 flops to two megaflops per node (scalar floating point) and, optionally, nodal vector processing capability (not covered here) [6].

The next generation multicomputer machines will provide ten megaflops per node scalar speed, creating true supercomputers in ensembles with a handful of nodes. Initially, such machines will have relatively slow communication, but this will improve over time [7,6]. We will be ready to utilize these machines when they become widely available in two to three years' time.

1.2.1 Multicomputers vs. Multiprocessors

A multicomputer is a message-passing concurrent computer, while a multiprocessor is a shared-memory concurrent computer (see Figures 1.3., 1.4.). Our keen interest in multicomputers stems from the ability to scale such machines to large ensemble sizes. Eventually, hybrid machines incorporating small node-count multiprocessors are likely. In such systems, the large-scale concurrency will pertain to the multicomputer-like features, with fine-grain concurrency exploited within each multiprocessor cluster.

The most common multicomputers are binary n-cube machines. These machines, first realized in the Caltech *Cosmic Cube* [40], have been commercialized by a number of vendors. Many machines based on two-dimensional meshes are also in commercial production, based mainly on the Inmos transputer technology. A binary n-cube ("hypercube") and two-dimensional mesh architecture are represented symbolically in Figures 1.6, 1.7. The Intel iPSC/2 is an example of the former, while the Symult (Ametek/ketemA) s2010 machine is an example of the latter. Both utilize advanced message-passing technology developed at Caltech [6]. The work described in this thesis is compatible with both of these machines.

While for first generation systems nearest-neighbor communication proved much cheaper than far-neighbor communication, advances in routing technology have reduced this effect dramatically [6]. As such, we advocate programming the machines without attention to their underlying hardware connectivity. Currently, the majority of multicomputer research is really "hypercube" research, done with attention to the specifics of the underlying architecture, which is certain to decrease the lifetime of the resulting applications dramatically (while skewing their directions in ways likely to be unimportant in the very near term). For example, in a later prototype, the DARPA Touchstone machine, initially a binary-n-cube-connected machine, will advance to a faster version of the identical mesh-backplane technology used by the Symult s2010

[7].

1.2.2 Operating Systems

The *Reactive Kernel / Cosmic Environment* node operating system developed by Seitz *et al.* at Caltech is likewise a high performance solution to portability between multicomputers, and an elegant basis for complex multicomputer applications. Though manufacturers are still quibbling over their parochial operating systems, and commercial companies are likewise promulgating further monstrosities, we expect this operating system to emerge over time, just as Unix has emerged as the vendor-independent operating system for workstations [42,6,41] (see also chapter 3).

The elegance of the *Reactive Kernel / Cosmic Environment* is exemplified in part by its ability to provide a unified interface for many different types of multicomputers, multicomputer emulation (and hence debugging), and distributed-parallel computation over networks of NFS-networked machines (see Figure 1.5).

1.3 Thesis Overview

From 1987-1988, we studied Waveform Relaxation, as an outgrowth of earlier interest in "tearing" methods for dynamic simulation in chemical engineering. Our work thus far in this area is contained in chapter 7. Its strategic location near the end of the thesis is not to relegate it, but to suggest that it is the more forward-looking area of research, and relates most closely to planned future work.

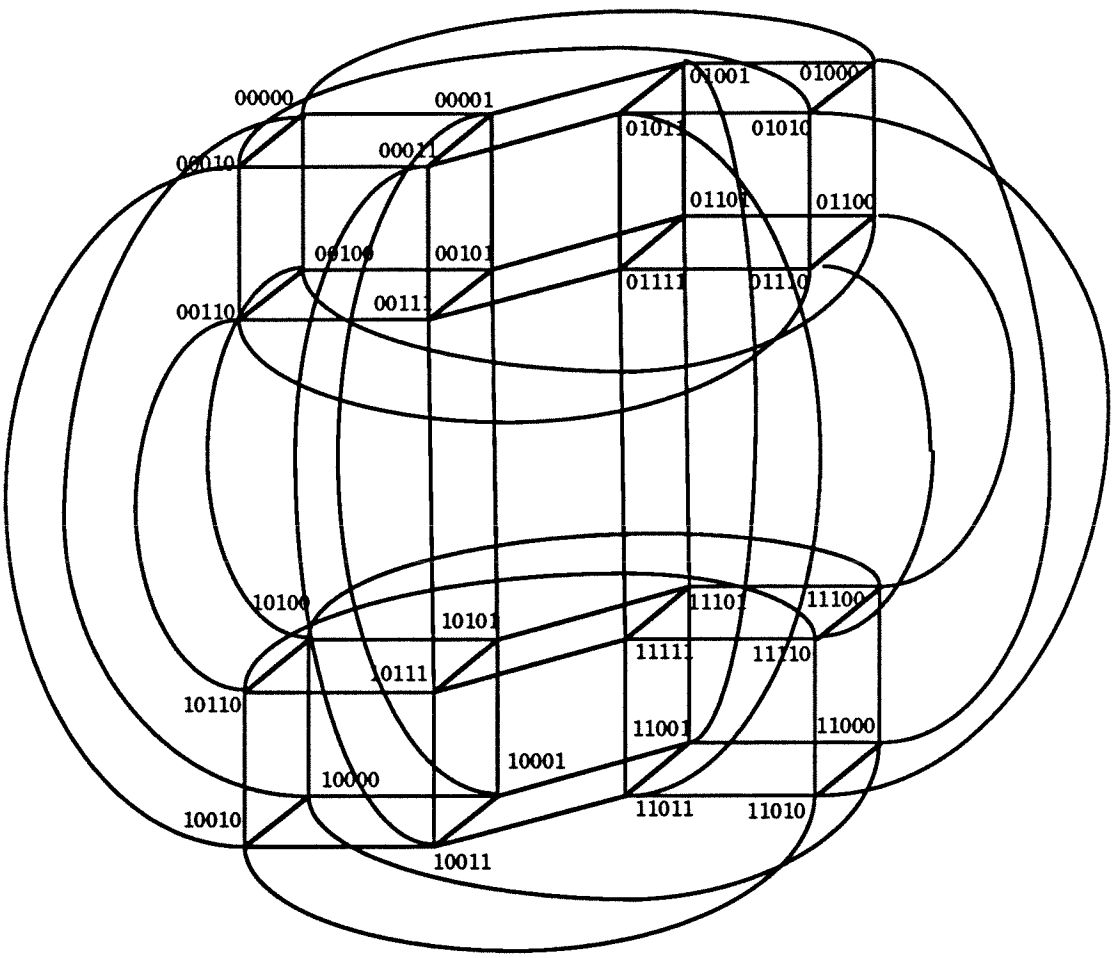
From 1988 on, we have worked on the development of efficient parallelizations of high-quality sequential algorithms. To do so, we began by analyzing the communication needs for complex multicomputer applications. The outgrowth of this study is the *Zipcode* communication layer described in chapter 3. All of the remaining software developed for this thesis depends on the underlying *Zipcode* layer, which is itself built upon the Caltech *Reactive Kernel.*

Figure 1.5. Prototypical Multicomputer Dæmon Display

```
*--------- Scheduling policy for the 192-node S2010 (:S2010) ---------*
| Mon-Fri 0900-1800: <1hr runs, open scheduling.  ALL OTHER TIMES: by |
| reservation only; send requests to sharon@perseus.ccsf.caltech.edu. |
| To see the schedule, type 'peek schedule sharon'.                   |
*---------------------------------------------------------------------*

CUBE DAEMON version 7.2, up 86 days 23 hours on host ganymede

   {                  }  6n s2010       , b:0000 [perseus :S2010       ] 39.7m
   {csp25A_12x3 tony  } 37n s2010       , b:0014 [perseus :S2010       ] 42.0s
   {csp25B_12x3 tony  } 37n s2010       , b:0009 [perseus :S2010       ] 42.0s
   { csp25_12x3 tony  } 37n s2010       , b:0008 [perseus :S2010       ] 48.0s
   {csp25B_12x2 tony  } 25n s2010       , b:0013 [perseus :S2010       ] 15.1m
   {      kalon tony  } 25n s2010       , b:0011 [perseus :S2010       ] 15.2m
   {     zephyr tony  } 25n s2010       , b:0010 [perseus :S2010       ] 15.2m
   {                  } 20n s2010       , b:0000 [ psyche :ginzu       ] 19.7h
   {      SVEN1 concise}  2n s2010       , b:0014 [mercury :ginzu       ]  3.3h
   {      LENA4 concise}  2n s2010       , b:0012 [ mosaic :ginzu       ]  8.7m
   {      LENA5 concise}  3n s2010       , b:000c [ mosaic :ginzu       ] 11.2m
   {      LENA3 concise}  2n s2010       , b:000f [ mosaic :ginzu       ]  2.1h
   {      LENA6 concise}  3n s2010       , b:000e [ mosaic :ginzu       ]  2.6h
   {                  } 16n ghost cube  , b:0000 [ mosaic :sparcomatic]  1.1d
   {                  } 16n ghost cube  , b:0000 [   solo :mimic       ]  1.3d
   {                  }  4d ipsc2 cube  , b:0000 [  ipsc2 iPSC2        ]  2.0d
   {                  }  3d cosmic cube, b:0000 [  venus fly trap     ]  3.9d
   {                  }  2n ghost cube  , b:0000 [mercury skjellum    ] 10.0d
   {      first dasilva}  2d non cube    , b:0000 [ stun3h             ]  5.6d
   {   schedule sharon }  3d non cube    , b:0000 [perseus             ]  1.2d

GROUP {csp25_12x3 tony}  TYPE reactive  IDLE 0.0s

   ( -1   0)   newcsparsetest1   126s    17r   0q   [perseus 15797] 39.0s
   ( -1  -1)            SERVER    17s    17r   0q   [perseus 15786] 45.0s
   ( -1  -2)          FILE MGR     0s     0r   0q   [perseus 15788] 45.0s
   (--- ---)           CUBEIFC     9s   179r   0qw  [perseus 15225] 47.0s
```

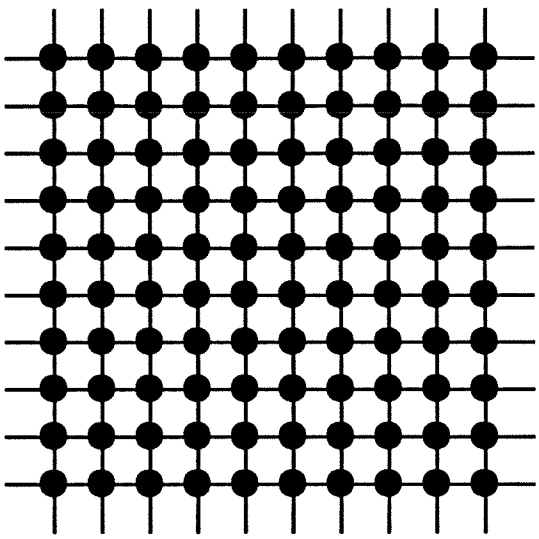
The *Reactive Kernel / Cosmic Environment* supports remote hosting transparently over NFS-networks. Users on different machines across the network may space-share a wide variety of real concurrent machines (Symults, iPSC/2's, iPSC/1's), and simulated machines ("Ghost" and "non" cubes). Ghost cubes provide distributed parallel processing across many networked CPU's; non cubes operate on a single workstation.

Figure 1.6. Five-Dimensional Binary n-cube Multicomputer Schematic



Cube vertices represent computer nodes. Each node of a binary n-cube, or "hypercube," has exactly n connections to "nearest neighbors." Here, each vertex is connected to five other vertices.

Figure 1.7. Schematic of a Two-Dimensional Mesh Multicomputer



The Symult s2010 architecture uses a two-dimensional communication network with parallel channels along with an improved routing mechanism for much higher message-passing performance than first-generation multicomputers, as described in [6]. Mesh "edges" provide a rich potential source of input-output bandwidth.

Beginning in Autumn 1988, and continuing through the end of that year, we developed the first release of *Concurrent DASSL*, based on earlier study of the *DASSL* integration code, a standard solver for ordinary differential algebraic problems of low index [9]. This work was intended to provide a general purpose simulation tool, and was to include concurrency features to enable reaching the "parallelized" achievable concurrent performance of applications built upon it. At that time, we interfaced a version of Eric Van de Velde's dense concurrent linear algebra to it, to form an initial simulation engine [56].

During the first half of 1989, we developed a single-column simulation driver above *Concurrent DASSL*. This early prototype was replaced by a more powerful prototype during the subsequent six months, which we call *proto-Cdyn*. As its name indicates, this too is a prototype simulation driver for *Concurrent DASSL*; currently, it can generate simulations of arbitrary networks of distillation columns with a fixed tray

model.

Simultaneous to the creation of *proto-Cdyn*, we developed a new concurrent sparse solver, whose main results are summarized in chapter 5. This work was done together with Alvin P. Leung, a Caltech "SURF" fellow. Work on improving this algorithm continued through the end of 1989.

During the early part of 1990, we completed the union of *Concurrent DASSL* and the sparse solver, and initiated various tests and runs. Chapters 6 and 5 summarize some of these results, respectively.

The several appendices include additional details on the thesis work, including further discussions of the sparse solver, *Concurrent DASSL*, *Zipcode*, as well as data distribution derivations.

Chapter 2

Concurrent Simulation Paradigms

Abstract

In this chapter, we cover the basic driving forces in concurrent computation (bottlenecks, granularity, scalability, load imbalance), and the importance of the chosen numerical algorithm. Abstractly, we describe the numerical challenges to be met and indicate concurrency issues. In this connection, we outline a conservative philosophy for passing from the sequential to the concurrent regime based on rough measures of achievable concurrent performance; this route to concurrency attains parallelism in existing numerical algorithms without changing convergence properties. New numerical techniques, in constrast, potentially offer higher concurrent performance, yet forsake the "safe ground" of known numerical properties, requiring significant new numerical analysis, heuristic techniques, and experimentation.

2.1 Introduction

By a concurrent simulation paradigm, we mean a fuzzy set of numerical methods applicable to a fuzzy target set of problems. For example, we know that ODE-methods for numerical solution of ordinary differential-algebraic equations (DAE's) of low index are applicable to a large set of problems in engineering. Within this collection of methods, there are a number of ways to set up the predictor-corrector equations, and linear algebra can be solved directly or iteratively via various algorithms (de-

pending on problem structure). Furthermore, there are competitive algorithms for error, step-size and order control. We can parallelize various aspects of these methods while preserving numerical properties, and the methods will still apply to the same fuzzy target set as they did sequentially. We also recognize that the concurrent performance will depend heavily on the particular properties of the problems to be solved, even though convergence will be unchanged as we pass to the new concurrent solution approaches.

If we seek higher performance for specific problems, we will have to modify the numerical methods to account for problem characteristics. For example, we may find ways to extract timelike parallelism in the algorithms. Then, the fuzzy target set will get smaller (because we introduce some stronger assumptions). Alternatively, if we start with a new method designed explicitly for high concurrency like Waveform Relaxation, then we will have to explore its effectiveness for many applications in order to gauge its fuzzy target set of applicability somewhat. We will still have to develop special methods for improved performance for particular problems. However, in this case, we are working on less certain grounds.

In this thesis, we investigate aspects of both approaches, though we have concentrated more heavily on the parallelization of the extant numerical techniques for DAE solution. The dual simulation paradigms – existing numerical methods, and new numerical methods – should be considered for each new application class we need to address. Parallelized existing numerical methods establish first the lower bound for concurrent performance and are consequently a conservative, though important, starting point.

Understanding the fundamental driving forces of concurrency is important to our investigation. They are covered in the next section. Then, we return to the idea of the "choice of algorithm" in the following section, including examples of two approaches to a simple system of differential equations. On a more practical note, we conclude

this chapter with a discussion of multicomputer programming style, including grid-oriented data distribution and example concurrent operations. Understanding the basic operations is also important to framing effective algorithms for multicomputers.

2.2 Basic Computing Issues

Amdahl's law is an often-quoted (and misquoted) measure of the upper limit on achievable performance for a concurrent algorithm [3,19,32]. It is a fundamental statement that sequential bottlenecks limit the ability of concurrency to provide *speedup.*

2.2.1 Nomenclature

Definition 2.1 (Relative Speedup) *Given a fixed algorithm $\mathcal{A}$ and problem (or problem-size) $\mathcal{P}$ with execution time T_p when solved by means of p independent processes, the relative speedup is defined as*

$$S_p \equiv \frac{T_1}{T_p}. \tag{2.1}$$

Definition 2.2 (Fair Speedup) *Given a fixed algorithm $\mathcal{A}$ and problem (or problem-size) $\mathcal{P}$ with execution time T_p when solved by means of p independent processes, the fair speedup is defined as*

$$\hat{S}_p \equiv \frac{T_{seq}}{T_p}, \tag{2.2}$$

where T_{seq} is the time for the most efficient sequential algorithm executing on a single process, assuming sufficient storage capacity (and with no memory-hierarchy effects).

Definition 2.3 (Relative Efficiency) *The relative efficiency of a fixed algorithm $\mathcal{A}$ with problem $\mathcal{P}$ (solved by means of p independent processes) is given by*

$$\eta_p \equiv \frac{S_p}{p}. \tag{2.3}$$

Definition 2.4 (Fair Efficiency) *The fair efficiency is defined as*

$$\hat{\eta}_p \equiv \frac{\hat{S}_p}{p}, \tag{2.4}$$

in analogy to the relative efficiency.

Definition 2.5 (Amdahl's Law) *Let T_p, the time for solving a problem $\mathcal{P}$ with fixed algorithm $\mathcal{A}$, be parametrized in $\alpha \in [0,1]$ by*

$$T_p \equiv \alpha T_1 + \frac{(1-\alpha)T_1}{p}, \tag{2.5}$$

where α is the (presumed fixed) inherently sequential fraction of computation. Then, the relative speedup S_p is limited by

$$S_p = \frac{p}{1+(p-1)\alpha} = \frac{1}{\alpha+(1-\alpha)/p} \leq S_\infty \equiv \alpha^{-1} \tag{2.6}$$

for any p. S_∞ is also the order-of-magnitude of the number of independent processes that may be efficiently used to solve $\mathcal{A}$. Evidently, a large sequential fraction α severely limits performance, as one might expect intuitively.

Definition 2.6 (Memory-Imposed Performance Limitations) *On a real machine, there is a minimum number of independent processes R_{min} (on separate computational nodes) needed to effect Algorithm $\mathcal{A}$ on problem $\mathcal{P}$, because of memory requirements. If $R_{min} \ll \alpha^{-1}$, then memory requirements don't constitute an important limiting effect. However, if $R_{min} \sim \alpha^{-1}$, or $R_{min} \gg \alpha^{-1}$, memory requirements*

probably pose an important limitation on achievable concurrent performance. Too many nodes are likely to slow a calculation (see section 2.2.2).

We seek algorithms capable of using hundreds or more independent processes, and Amdahl's law establishes that any algorithm capable of high performance cannot inherently possess a significant sequential fraction. Sometimes, other performance quantities are mentioned in connection with concurrency (skirting this fact). For example, one can *vary* the amount of per-process work while holding the number of processes fixed. Alternatively, the amount of per-process work can be held fixed while indefinitely increasing the number of processes. Neither of these indicates the speedup potential of a particular algorithm for a particular problem (or problem size) at fixed accuracy, the measure we consider most important for our applications. Consequently, it is unsurprising that these other measures, which are outside the assumptions of Amdahl's "law," can readily violate its limitations. See appendix F, as well as [25].

Finite communication bandwidth imposes a further limitation on the practically achievable performance of a concurrent algorithm. Except for problems totally lacking communication between processes (often called *embarrassingly parallel*), communication costs must be accounted for and efficient strategies are normally important to the overall performance of the algorithm. A simple design equation, denoted the "granularity design equation," helps in the semi-empirical performance evaluation of existing algorithms and in the top-down design of new concurrent algorithms.

Definition 2.7 (Granularity Design Equation) *Let γ be the startup cost in μs needed for transmitting a message between any two processes. Let δ be the per-operand (*i.e., *per 8-bytes for double precision real numbers) incremental cost of a message transmission, also in μs. Let τ be an appropriate measure in μs for the cost of a basic operation (*e.g., *double-precision multiplication), let n be the number of operands forming the message, and let o be the operations per operand needed to form the*

message. Then, the granularity design equation is as follows:

$$T_{oper}(n, o) \equiv \frac{\gamma + n\delta}{\tau o}, \tag{2.7}$$

where T_{oper} is a measure of the per-operand cost of the message transmission. For typical multicomputers, $\gamma \sim 250\mu s$, $\delta \sim 1\mu s$, and $\tau \sim 5\mu s$. Consequently, assuming these values, for a message of ten operands, with ten operations per operand, $T_{oper}(10, 10) = 5.16$. Thus, for this scenario, the cost of computation and communication are 10:1 in favor of computations per operand. By comparison (with the same hardware parameters), $T_{oper}(1, 1) \approx 50$, but this is much too harsh a measure of the natural granularity of the system.

This design equation should be applied wherever possible a priori in the top-down design of concurrent algorithms to determine the communication and computation characteristics needed to achieve high performance for a particular application. At that design stage, the computer (and hence the τ, γ, δ may be chosen within bounds), as can the n, o, representing the choice of algorithm, and the grain size for the problem formulation.

Definition 2.8 (Process Grain Size) *For a given operation, if its $T_{oper} \sim T_{oper}(1, 1)$, it is said to be fine-grain, while if its $T_{oper} \ll T_{oper}(1, 1)$, it is said to be coarse grain.*

Definition 2.9 (Systemic Grain Size) *Given a fixed, large amount of memory, we could divide it between myriad simple processors each with a few kilobytes. This constitutes a fine-grain machine. A coarse-grain multiprocessor, like a Cray, would store the entire memory in a few (1-8) nodes. In between are the* medium-grain *multicomputers we consider in this thesis. They have several megabytes of memory, and up to several hundred nodes. See also [37].*

The final generic source of inefficiency[1] in a concurrent algorithm is load imbalance among processes. Whenever processes must wait for the receipt of data, there are lost CPU-cycles that could otherwise have contributed to speedup. There are many sources of load imbalance in the complex, inhomogeneous calculations arising in engineering. For instance, the cost of model evaluation is a strong function of the physical device (*e.g.*, transistor *vs.* resistor) or unit (*e.g.*, variable cost of distillation tray thermodynamics). Load imbalance also occurs in homogeneous operations (like linear algebra computations) because of static distribution divisibility problems. Real-world applications won't normally divide evenly between the processes involved and there are consequently an unequal number of equations per process. Furthermore, processes become dynamically imbalanced in procedures like Gaussian elimination (equivalently, LU Factorization) where matrix elements become inactive as the elimination proceeds. Static techniques for the improvement of load balance are mentioned in chapter 4 and, in brief, below.

Flatt ([19]) indicates extended Amdahl's-law arguments for computational models including communication and load-imbalance effects, which we build upon here. We begin again with a more detailed model of the execution time:

$$T_p = \left(\alpha + \frac{(1-\alpha)}{p} + \varsigma(p)\right) T_1, \tag{2.8}$$

where $\varsigma(p)$ is a measure of overheads, such as communication and load imbalance; $\varsigma(1) \equiv 0$. Substituting Equation 2.8 into the definition for speedup, Equation 2.1,

$$S_p = \frac{p}{1 + (p-1)\alpha + p\varsigma(p)} = \frac{1}{\alpha + (1-\alpha)/p + \varsigma(p)}. \tag{2.9}$$

[1] See [25, pages 636-637] concerning other sources of inefficiency.

The overhead $\varsigma(p)$ itself can be modelled as follows for $p \geq 1$:

$$\varsigma(p) = \varsigma_0 \log_2 p + \varsigma_1(p-1) + \varsigma_2\left(\frac{1}{p} - 1\right). \tag{2.10}$$

The only "reasonable" term is ς_2/p, since it decreases as we increase the number of computers. We wish to create algorithms with high ς_2, and low α, ς_0, ς_1. We interpret ς_0 as global-communication costs from *broadcasts* and *combines*. These fortunately grow only slowly in p. The ς_1-term comes from several possible sources: first, communication between the host and node processes, which must evidently be held to a minimum; inefficient communication structure between nodes (or graphically dense problem connectivity), and load imbalance effects worsening with increasing p. The ς_2-term represents overheads like packing and unpacking of data (for shipment between processes), that parallelize as we increase the number of processes. The next section views sequential fraction and overhead in a qualitative way, through a "Concurrency Diagram."

2.2.2 The Concurrency Diagram

Flatt ([19]) points out, "The characteristics of the algorithms used or required for a given problem may be much more important than the details of the hardware implementation of the parallel system." We reflect on this theme in the "Concurrency Diagram," Figure 2.1. In the diagram, ideal performance is indicated by two thick, slope-minus-one lines for two hypothetical algorithms – a "best" sequential algorithm when parallelized intelligently, and a "best" concurrent algorithm for the same problem (or problem size). Actual performance curves for the hypothetical algorithms appear above the ideal lines, as labeled. The x-axis depicts the quantity of node resources dedicated to the solution of the problem, whereas the y-axis depicts the benefit (decreased time) as a function of resources. On this log-log plot, we have normalized

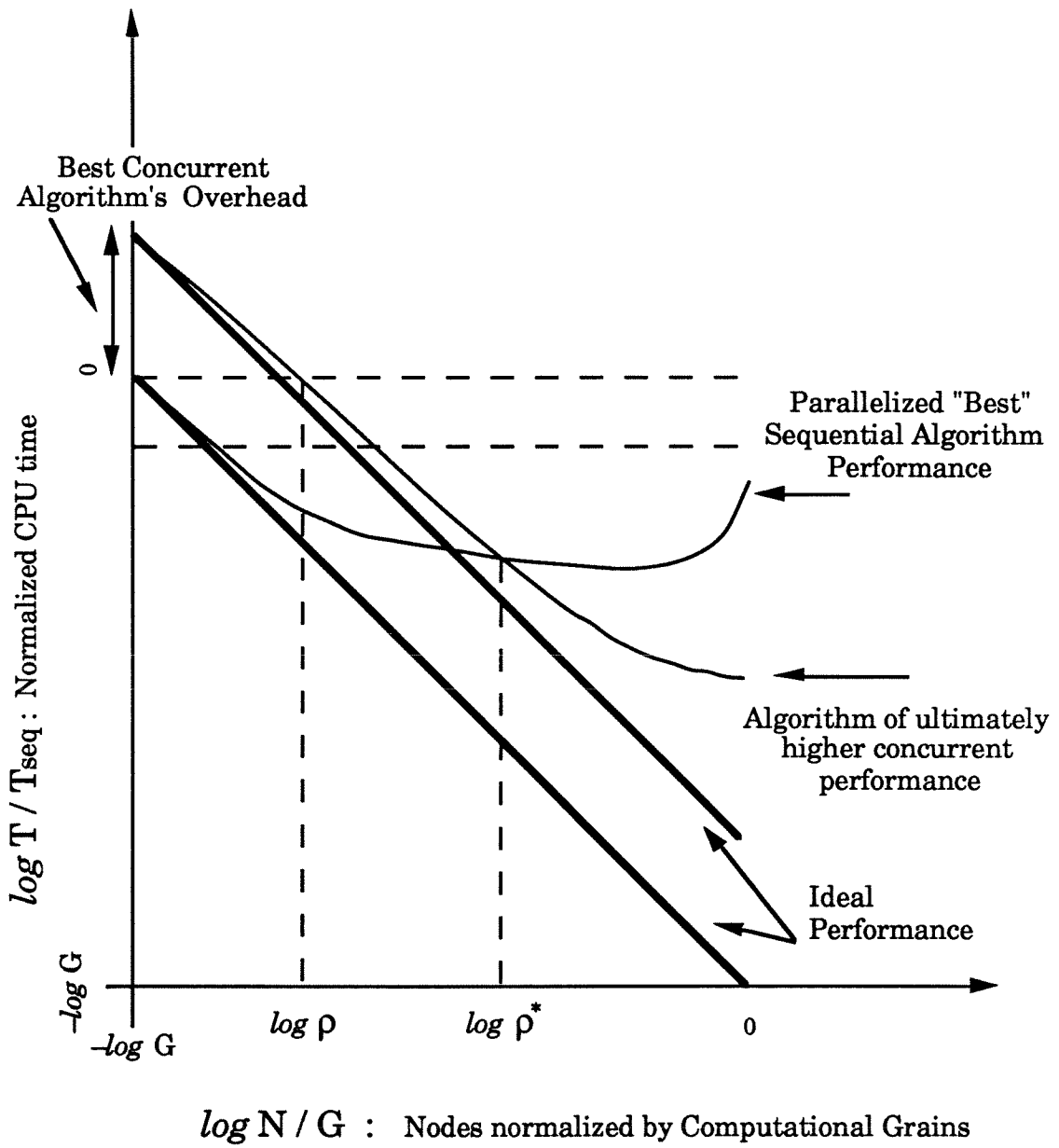
the quantity of resources applied by the number of computational grains, and the time by the sequential time of the "best" sequential algorithm; these normalizations are of secondary importance.

Evidently, the "best" sequential algorithm will tend to have a higher α, and lower overall ς than the "best" concurrent algorithm. Hence, it will outperform the latter initially. In fact, as a result of the overheads in the "best" concurrent algorithm, this procedure will not even break even with T_{seq} until we expend resources ρ. (So, for machines of low parallelism, we will never see advantage from the "best" concurrent algorithm.) However, as the sequential fraction begins to dominate for the "best" sequential algorithm, its actual performance becomes lazy, eventually tailing upward. This happens much later for the "best" concurrent algorithm, which is still speeding up well for resources comparable to ρ^*. At ρ^*, the algorithms have equivalent performance; beyond that, the "best" concurrent algorithm proves superior.

Of course, we have to work to design the "best" concurrent algorithm so that it has these desirable qualities. This will prove non-trivial, in general. We also note that, sometimes, the process of searching for new concurrent algorithms actually generates a better sequential algorithm. In this case, we must relabel our diagram so that, again, the "best" sequential algorithm has the lowest time for the single-processor limit (see, for example, [58]). Otherwise, we are artificially inflating our results – we do not wish to claim "superlinear speedup" any more than we could credibly claim perpetual motion.

The "Concurrency Diagram" is a powerful means of expression of concurrency ideas, including the ability to compare algorithms run on different machines. The log-log plot has the distinct advantage that we can completely drop normalization constants if desired (plotting CPU time *vs.* resources), and remove the arbitrary scale factor that invariably plagues speedups. As such, we recommend it as the standard for presentation of performance results in concurrent computation.

Figure 2.1. The Concurrency Diagram



The Concurrency Diagram illustrates the trade-offs between the "best" parallelized sequential algorithm and the "best" concurrent algorithm. The former has a higher sequential fraction, but lower overhead compared to the latter. The "best" concurrent algorithm has additional (parallelizable) overhead, but a smaller sequential fraction, allowing it to achieve higher speedups when many nodes are used (large-resource limit, beyond ρ^*).

2.3 Choice of Algorithm

Beginning in the late 1960's and continuing through the early 1980's, many "physically motivated," hierarchical *coordination-decomposition methods* (*e.g.*, [43]) were forwarded for the solution of large-scale engineering problems. These methods, based on a partition of an optimization or simulation problem into multiple subproblems (each corresponding, perhaps vaguely, to some subsystem of the modelled system), relied heavily upon *central coordination*, the process by which, through iteration, the subproblem solutions could ultimately be brought into agreement (reach a global solution). This style of problem formulation is most often unattractive, in retrospect, for several reasons: non-trivial central coordination imposes an unacceptably large sequential fraction on a computation, thereby stunting speedup; the formalism doesn't begin to expose the important (spacelike and/or timelike) concurrency in the problem, problems must be 'contorted' to fit within the offered numerical framework, and there is no established *performance* for these iterative methods (*e.g.*, for a restricted problem class), even assuming they eventually converge to the correct solution. Kuru comments on their shaky mathematical justification as well [29].

The lesson derived from the abovementioned research efforts is that, for multicomputers, we should first evaluate the *achievable performance* of high-quality sequential algorithms that have been suitably generalized to incorporate concurrency. Sometimes, there are several well-known sequential procedures, but the best concurrent algorithm need not correspond to the most efficient sequential algorithm. Therefore, the search needn't be limited to just the considered "best" algorithm. In any event, this conservative pattern of migration to concurrent computing preserves the numerical properties of the corresponding sequential algorithm and this invariance is highly desirable. We may furthermore discover that the achievable concurrent performance is sufficiently good that further efforts become practically unjustified. If so, our effort terminates at this point with a useful concurrent production code.

In other instances, we find that the applications contain *hidden* concurrency. For example, important chemical and electrical applications are modelled by large, stiff systems of ordinary differential-algebraic equations. Integration procedures that incorporate a global timestep must be limited by the high-frequency effects in order to provide desired accuracy (assuming implicit solution methods), even though these dynamics may be unimportant or parasitic. In random-access memory devices, for example, there is a high degree of *latency*; that is, only a small fraction of the bits are changed per unit time, and there is normally a pattern to the storage accesses. In a chemical plant, a number of units might, for some minutes or hours, operate sensibly at a steady state while others are changing dramatically. It makes perfect sense that latent parts of the system should be simulated with much larger integration time steps with active parts being integrated with appropriately smaller steps. This policy would avoid the wasteful work of model evaluation in the quiescent part of the system at the very least. The Waveform Relaxation algorithm described in [46,48] can potentially address this need and be applied both to chemical as well as electrical engineering simulations. Of course, the numerical analysis underlying this class of methods is still a subject of intensive research, as are the heuristics needed for high-performance implementations.

2.3.1 Waveform Relaxation Motivating Example

The basic Waveform Relaxation method (Picard-Lindelöf iterations) is analogous to the standard Gauss-Jacobi or Gauss-Seidel iterations used to solve a linear system of equations. However, Waveform Relaxation operates on groups of function approximations rather than on groups of real *values*. As an example, expanding on Vandewalle ([59]), consider the following problem (a pair of coupled ODE's) where

we shall iterate in the Gauss-Seidel sense:

$$\dot{x} = y, \quad x(0) = 0, \tag{2.11}$$

$$t \in [0, T],$$

$$\dot{y} = -x, \quad y(0) = 1. \tag{2.12}$$

Let the initial guess (iteration $k = 0$) be

$$x_0(t) = 0, \quad y_0(t) = 1, \tag{2.13}$$

and solve for $k = 1$

$$\dot{x}_1 = y_0 \Rightarrow x_1(t) = t, \tag{2.14}$$

$$\dot{y}_1 = -x_1 \Rightarrow y_1(t) = 1 - \frac{t^2}{2!}, \tag{2.15}$$

where, in the Gauss-Seidel sense, the new x-approximation, $x_1(t)$, is used immediately in the computation of $y_1(t)$. Repeating the procedure for $k = 2$:

$$\dot{x}_2 = y_1 \Rightarrow x_2(t) = t - \frac{t^3}{3!}, \tag{2.16}$$

$$\dot{y}_2 = -x_2 \Rightarrow y_2(t) = 1 - \frac{t^2}{2!} + \frac{t^4}{4!}. \tag{2.17}$$

In the limit as $k \to \infty$, we get

$$x_\infty(t) = \sum_{j=0}^{\infty} \frac{(-1)^j t^{2j+1}}{(2j+1)!} = \sin(t), \tag{2.18}$$

$$y_\infty(t) = \sum_{j=0}^{\infty} \frac{(-1)^j t^{2j}}{(2j)!} = \cos(t). \tag{2.19}$$

By inspection, the Waveform Relaxation method has this key property: Each sub-

system is computed independently. In this example, a system of two equations is reduced to two scalar systems whose iterative integrations are independently computed. Iterative solution is the price of decoupling.

There is an important catch in this example. Because we chose the Gauss-Seidel update mechanism, there is inherent sequentialism in the completion of each iteration. The x-equation will always be one iteration ahead of the y-equation (in this variable ordering). Gauss-Jacobi iteration removes this sequentialism at the expense of slower convergence (twice as many iterations here):

$$\begin{aligned}
\dot{x}_1 &= y_0 \;\Rightarrow\; x_1(t) = t, \\
\dot{y}_1 &= -x_1 \;\Rightarrow\; y_1(t) = 1, \\
\dot{x}_2 &= y_1 \;\Rightarrow\; x_2(t) = t, \\
\dot{y}_2 &= -x_1 \;\Rightarrow\; y_2(t) = 1 - \frac{t^2}{2!}, \\
\dot{x}_3 &= y_2 \;\Rightarrow\; x_3(t) = t - \frac{t^3}{3!}, \\
\dot{y}_3 &= -x_2 \;\Rightarrow\; y_3(t) = 1 - \frac{t^2}{2!}, \\
\dot{x}_4 &= y_3 \;\Rightarrow\; x_4(t) = t - \frac{t^3}{3!}, \\
\dot{y}_4 &= -x_3 \;\Rightarrow\; y_4(t) = 1 - \frac{t^2}{2!} + \frac{t^4}{4!} \\
&\ldots
\end{aligned}$$

At each k-iteration, the equations may be solved independently and simultaneously, but converge more slowly.

In practical implementations, ordered collections of time-value pairs are used to represent function approximations ("waveforms") on finite intervals. Unlike our formal integration in the above examples, implicit numerical integration procedures independently select time steps in each subsystem. These time steps are optimized

to reflect local error criteria. Interpolation provides approximate function values at desired points. For large, sparse systems that arise in circuit simulation and flowsheet simulation, the advantages of a multirate integration method may more than compensate for the extra work of multiple outer iterations incurred by the splitting into independent subsystems (for instance, "global" linear algebra is avoided).

From the above examples, we can already see potential complications with Waveform Relaxation and it's only fair to indicate these from the outset. First, we have to wonder if the method will converge at all for interesting problems. Fortunately, convergence is provable under mild circumstances for purely differential equations, and is also under investigation for differential-algebraic equations ([30,36,35]). Simple distillation networks, and general circuit networks modelled by *nodal analysis*, meet these criteria (see [46,32]). In more complicated circumstances (like discontinuous model equations and systems with purely algebraic equations), convergence is not clear cut. Furthermore, the central issue beyond convergence is performance. If many outer iterations are required, all the benefit accrued from the splitting will be lost.

We note finally that, for the above example, the best ∞-norm approximations for n-term representation of the cosine and sine functions are *not* the Taylor expansions (as evolved by the example procedures) but, instead, particular Chebyshev polynomials of order n. Consequently, the approximation error produced by Waveform Relaxation will in no sense be uniform over the interval of integration, strongly motivating algorithmic heuristics to improve performance. In particular, we will have to partition the problem into sub-intervals ("time windows") in order to obtain reasonable accuracy and performance [32].

2.3.2 A Direct Example

Consider again the example Equations 2.11, with initial conditions Equations 2.13. We consider the numerical solution by an ODE method with fixed step size h. For

compactness of notation, let

$$\mathbf{z} = \begin{pmatrix} x \\ y \end{pmatrix}, \tag{2.20}$$

$$A = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \tag{2.21}$$

and

$$\dot{\mathbf{z}} = A\mathbf{z}. \tag{2.22}$$

We discretize the time-derivative by the backward Euler method (which is consistent):

$$\dot{\mathbf{z}}(k+1) \approx \frac{\mathbf{z}(k+1) - \mathbf{z}(k)}{h}, \tag{2.23}$$

where $t = hk$. Then, briefly,

$$\mathbf{z}(k+1) = (I - hA)^{-1}\mathbf{z}(k), \quad k = 0, 1, \ldots \tag{2.24}$$

with

$$\mathbf{z}(0) \equiv \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{2.25}$$

For small enough h, we will recover an accurate solution ($O(h)$-accurate method); the method is stable for large h as well, though inaccurate.

For this approach to the numerical solution, we were forced to solve a linear system of equations at each k-iteration:

$$(I - hA)\mathbf{z}(k+1) = \mathbf{z}(k). \tag{2.26}$$

Were the original ODE's nonlinear, we would have had to solve a pair of nonlinear

equations at each k-iteration via Newton-Raphson iteration, in turn requiring multiple linear-equation solutions for convergence to the solution. The linear equations could be solved by LU Factorization, in which case the linear-system solution represents a strong synchronization between the equations – there is limited concurrency in the factorization, and less in the triangular solves, which are particularly sequential. However, this solution approach has rather nice numerical properties. Step-size is selected simply to control the solution error; stability is not a problem because we selected an implicit method.

These two approaches point up a fundamental trade-off in concurrency. Concurrency must usually be traded-off against convergence.

2.4 Multicomputing Programming

A straightforward programming model is offered by the multicomputer operating system of choice, Caltech's *Reactive Kernel / Cosmic Environment.* This environment is available on a number of parallel and standard computers. Thus, our applications port immediately between Symult s2010 and Intel iPSC/1, iPSC/2 multicomputers as well as Sequent multiprocessor systems and conventional ether-networks of Sun workstations. On real multicomputers, the *space-sharing* concept is supported. This feature allows multiple users to share a single multicomputer; each user gains exclusive access to a portion of the system, which logically appears as a smaller, independent multicomputer [41].

The programmer defines a single host process that provides an interface to the outside world, spawns node processes, and most often serves as the light scheduler (or coordinator) for the entire calculation. Each computer node may support one or more independent time-shared processes. An important feature of the system is that program correctness is independent of the mapping of logical processes to physical processes on actual computer nodes. Conventional, unaugmented languages such as

C and Fortran-77 are supported (although Fortran-77 is poorly suited to the needs of multicomputer programming, notably for its lack of data structures, pointer variables, and dynamic memory allocation). Each process is, in short, a traditional sequential program with the added ability to transmit and receive messages via subroutine-based primitives (communicating sequential process) [27]. These primitives provide untyped, blocked and unblocked message passing of arbitrary length messages. In our applications, we include an application-oriented communications layer atop the *Reactive Kernel* functions and thus avoid direct reference to the fundamental primitives. We cover this in detail in chapter 3.

The programming model guarantees that message order is preserved between any pair of processes. There are no synchronization primitives, shared memory, or other multiprocessing mechanisms. Both node and host processes have access to Unix-like system commands in addition to the basic communication primitives.

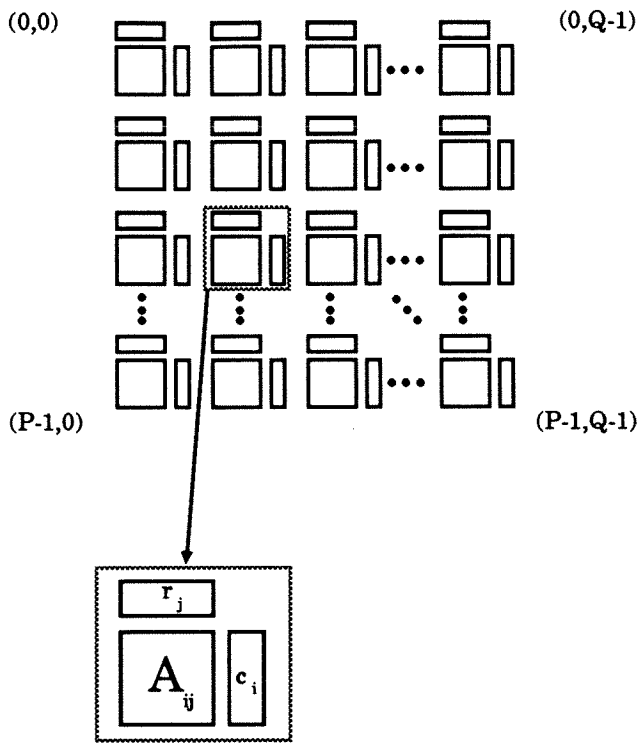
2.5 Fundamental Building Blocks

This section describes fundamental concurrent procedures that underly our application codes. In this connection, we abstract to logical process grids, see Figure 2.2. We describe linear-algebra procedures at a qualitative level within the grid formalism. The ideas developed here are helpful when we describe the *Concurrent DASSL* integration algorithm in chapter 6.

2.5.1 Grids and Communication, Primitive Operations

The key operations that connect parts of an ensemble calculation in a multicomputer are the communication steps. The *Reactive Kernel* provides very low-level commands with the intent that applications should tailor their own primitives on top of the basic operations. We've defined a set of higher-level operations realized in the *Zipcode* system, some qualitative features of which are described here.

Figure 2.2. Schematic of a Logical Two-Dimensional Process Grid



The two-dimensional process grid is the canonical format for vectors and matrices in multicomputer algorithms. Vectors are either row- or column-distributed. They are duplicated in the direction orthogonal to their distribution. Matrices are distributed but not duplicated.

Motivation

Application-level primitives have to provide various simplifying features in order to hide the effort of information sharing. A calculation is distributed among a collection of *R logical* processes, themselves assigned over one or more *physical* concurrent computer nodes. Process *lists* reflecting this distribution are consequently a fundamentally important data structure. In short, each process has to know how to contact all the other processes with which it must interact. The list is like an address book, with entries specifying the "mailing address" for the processes. Communication primitives of need specify the addressee (or addressees) when the message is posted; in

contrast, addressees need the ability selectively to "screen" their messages based on the sender, and other information that we'll discuss next.

In general, we don't think of the R processes in a process list as a linear collection numbered $0 \ldots R-1$ but rather attach a higher-level of abstraction. We define one or more process *grids* based on this process list. A process grid of $P \times Q = R$ maps two integers $(p, q) \mapsto (0 \ldots R-1)$, the actual process list range. Importantly, this effort is hidden from the application. Any reasonable number of grids can be defined at the outset of program execution, and message selectivity provides for screening messages based both on their source and their grid association. To post a message, a process selects a grid and specifies one or more coordinate pairs (*e.g.*, (p_1, q_1)) as the message's destination (or destinations). Broadcasting a message to an entire process grid is also supported.

A natural consequence of the grid abstraction is the definition of subgrids. In our experience to date, we have worked exclusively with row and column subgrids. As we shall see when exploring simple concurrent algorithms, data will often have to be shared between members of a process column or between members of a process row. Broadcasting information from one row (column) process to the entire row (column) is an important example. Forming a sum of data (*e.g.*, a norm) is another row- or column-oriented operation.

Grid-Independent Programming

It is germane to point out the importance of our support for a multiplicity of grids involving the same or distinct lists of processes. Previous message systems (with which we are familiar) have neglected this capability, thereby condemning application programs to select one grid and retain it for the entire calculation. Single-grid concurrent programming is not adequate for complex applications because different computational phases may justifiably require different logical grid shapes (and, more

generally, sizes).

In our convention, every distributed data structure is associated with a grid at creation. Vectors are either row- or column-distributed within a two-dimensional grid. Row-distributed vectors are *replicated* in each process column, and distributed in the process rows. Conversely, column-distributed vectors are replicated in each process row, and distributed in the process columns. Matrices are, however, distributed both in rows and columns, so that a single process is alloted a subset of matrix rows and columns. Distribution of the data is a natural consequence of our desire to apportion computing between multiple processes and, ideally, to derive non-trivial speedup.

We also are at liberty to decide *how* the coefficients of a vector are to be distributed. For example, for a row-distributed N-vector $\mathbf{z}$, each member of the zeroth process row could receive the first r elements[2] $(z_0, \ldots, z_{r-1})$, the first process row, $(z_r, \ldots, z_{2r-1})$, and so forth, with the last $(P-1\text{st})$ process row receiving the final elements $(z_{N-r}, \ldots, z_{N-1})$. This is called a *linear distribution* of the coefficients and has the effect of keeping neighboring coefficients in a vector together as much as possible. For many types of vector-vector operations, this distribution will prove adequate.

Linear row and column distributions of a matrix are generally inefficient for LU factorization. It's common in LU factorization for nearby rows of the matrix to be eliminated in turn. Elimination will tend to deactivate processes completely as it proceeds, thereby reducing the overall efficiency of the concurrent computation by grossly imbalancing the work load of various processes. For such operations, we have to *scatter* coefficients, placing coefficient neighbors in separate processes. Below, we'll discuss two vector-oriented operations where linear distributions will be acceptable. For LU factorizations, we imagine using scattered row and column distributions. Contrarily, the linear distribution is best column-distribution for the triangular solves proceeding LU factorization (see chapters 4 and 5).

[2]For simplicity of exposition, assume here that $N = rP$, something that won't generally happen.

The choices of process grid and linear/scattered coefficient distributions are designed to improve concurrent performance. Individual process calculations can be written to work within a given distribution without particular attention to its details. For example, a process instantiation at location (p, q) of a grid can adapt to the size and shape of the grid; most often, it will be unconcerned whether its data is scattered or linearly distributed: functions that transform between global and local indices hide this complexity (see chapter 4). So, for example, the identical LU factorization code would function (though inefficiently) on a linear-linear distribution of 2×8 processes and, in another instance, on a scattered-scattered distribution of 9×3 processes, or in a single process on a 1×1 grid.

Grid- and data-distribution-independent programming is essential to the construction of complex multicomputer applications. Without this underlying support, too many repetitions of programming effort are inevitable, seriously reducing the cost effectiveness of this computing technology and, at the very least, severely slowing the pace at which it can be brought into practical, widespread use.

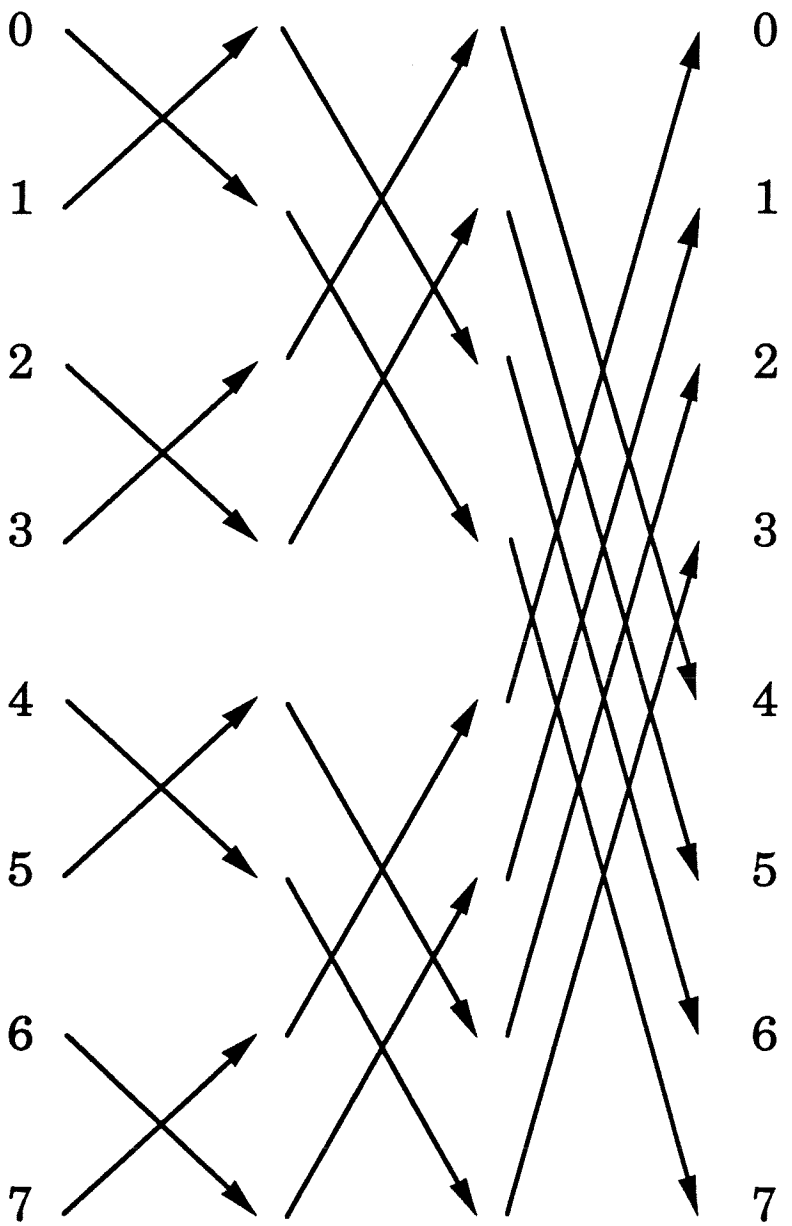
2.5.2 Selected Concurrent Operations

The "Combine" Operation — Recursive Doubling

Recursive doubling is the ubiquitous operation of multicomputer programming that lets us accumulate data in an efficient way. It is defined as follows:

Definition 2.10 (*Combine:* **Recursive Doubling**) *Given R processes (belonging to a process list $\mathcal{L}$, or grid $\mathcal{G}$), a homogeneous data object O containing r (ostensibly unrelated) items in each process, and an* associative, commutative, *binary combination procedure $\mathcal{F}$ that successively combines the corresponding items in two instances of O (*e.g., *a function that performs normal vector addition on two r-vectors of real numbers), the recursive doubling procedure produces the combined result in* each *of the R processes in $\lceil \log_2 R \rceil$ steps. This procedure is symbolized in Figure 2.3. See also*

Figure 2.3. Recursive Doubling Schematic



With 2^3 participants, the recursive doubling procedure completes in three steps, illustrating its logarithmic complexity in the number of participants.

[50].

Each step involves the cost of locally combining two instances of O to yield O', the cost of transmitting the object O' via $\mathcal{F}$, and the cost of receiving another object O''. It is crucial that the overall cost have a logarithmic dependence in R, because the communications are a direct overhead of concurrency.

For example, we can apply *combine* to a row- (column-) distributed vector to produce the norm of that vector by suitable choice of the combination function, $\mathcal{F}$. First, the sum of squares of all the coefficients local to each process are formed. Then, the local sums (one scalar in each process) are in turn summed via *combine*. As it turns out, we will always apply *combine* to a process grid (or subgrid).

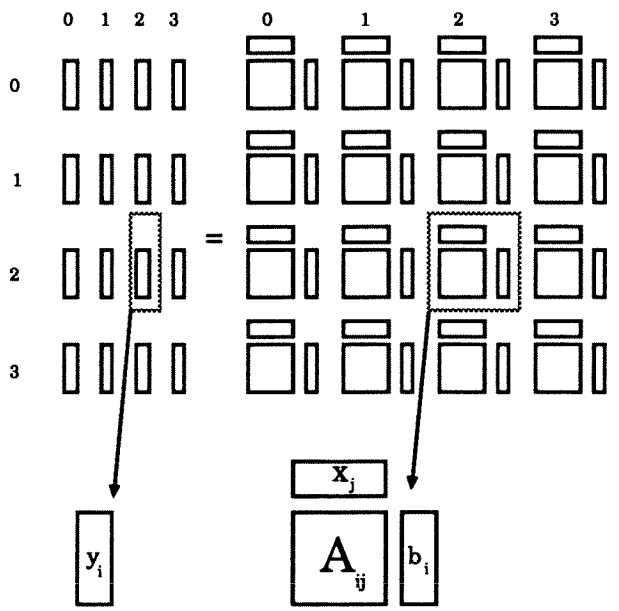
Broadcasts

Broadcasts, or *fanouts*, differ from *combines* in that a single process possesses the entire result initially, and wishes to share this with other processes as quickly as possible. All processes must know who the originator is for deterministic implementation of the procedure. This type of operation occurs prevalently in multicomputer applications, for example, in linear algebra codes. Although it can be emulated with *combine*, *broadcasts* are less inefficient than *combines* when a non-power-of-two participants are utilized. Hence, it is an important operation in its own right.

Figures 2.5, 2.6, and 2.7 depict threes styles of *broadcast* possible among eight participants. The time for completion of the first is $\lceil log_2 R \rceil$ for R participants. The second and third variations require one more communication phase each, $\lceil log_2 R \rceil + 1$. Here we indicate the style of transmission, but we say nothing about which processor should appear where in the communication tree (this is beyond our current scope). Though the Type #1 *broadcast* is currently implemented in our production codes, the Types #2 and #3 approaches are also of interest. Because they off-load the originating process, these other forms of *broadcast* might be utilized adaptively in

algorithms where the originator is naturally load-imbalanced (overworked) compared to the recipients. This occurs in linear algebra for the column of processes containing the pivot element. These ideas remain for future investigations.

Figure 2.4. Matrix-Vector Product Schematic on a 4x4 Grid



The vector $\mathbf{x}$ is replicated in the process rows and distributed in the process columns compatibly with the columns of A. The vectors $\mathbf{b}$ and $\mathbf{y}$ are replicated in the process columns and distributed in the process rows compatibly with the rows of the A matrix.

The Weighted-Vector Sum

Many vector-oriented operations occur in our applications, requiring the summation of distributed vectors.

Definition 2.11 (WVS: Weighted-Vector Sum) *Given three vectors* $\mathbf{x}$, $\mathbf{y}$, *and* $\mathbf{z}$, *all compatibly distributed according to grid* $\mathcal{G}$ *(row (column) replicated and column (row) distributed), and two scalar quantities* c_1, c_2, *the weighted-vector sum is defined as* $\mathbf{z} = c_1\mathbf{x} + c_2\mathbf{y}$.

As might be expected, completion time is proportional to the length of the largest local vector. The **WVS** operation involves no communication.

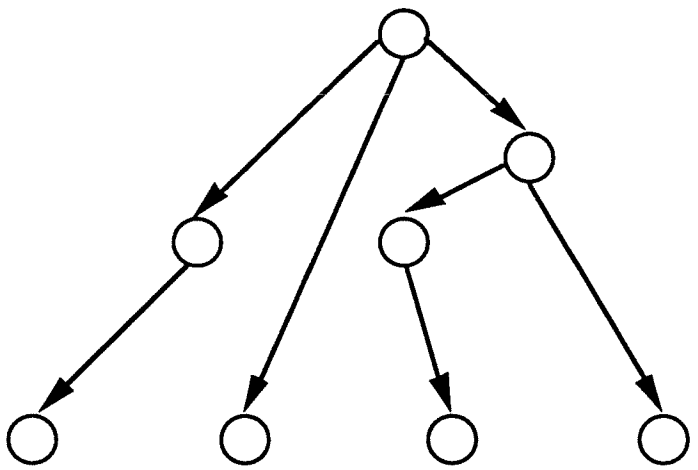
The Matrix-Vector Product

This fundamental operation comes into play as a kernel for iterative linear algebra. It clearly illustrates the importance of vector distribution within a grid and is consequently instructive.

Definition 2.12 (MVP: The Matrix-Vector Product) *Given a grid $\mathcal{G}$, an $M \times N$ matrix A distributed on $\mathcal{G}$, a column-distributed N-vector* $\mathbf{x}$ *(distributed as if it were a row of A but replicated in each process row) and row-distributed M-vectors* $\mathbf{y}$, $\mathbf{b}$ *(each distributed as if they were columns of A but replicated in each process column), the matrix-vector product is defined as $y = Ax + b$.*

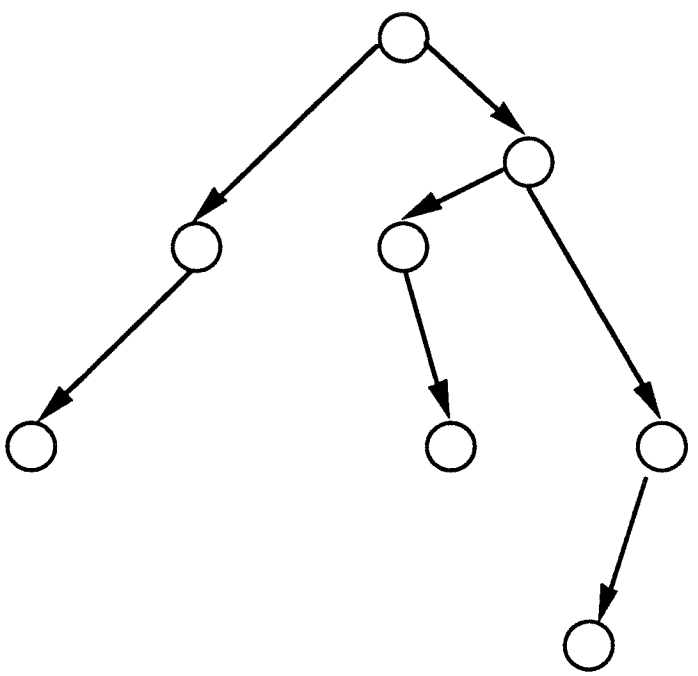
To effect this operation, each process performs its local matrix-vector product by "dotting" each row of the local matrix with the local $\mathbf{x}$ vector, and then adding the local vector $\mathbf{b}$ to this newly formed quantity. This sequence of operations produces the local contribution to $\mathbf{y}$ which, by recursive doubling across the process rows, produces the global $\mathbf{y}$ vector, distributed in the process rows and replicated in the process columns. This procedure is illustrated for a 4x4 process grid in Figure 2.4.

Figure 2.5. Broadcast Type #1 Schematic



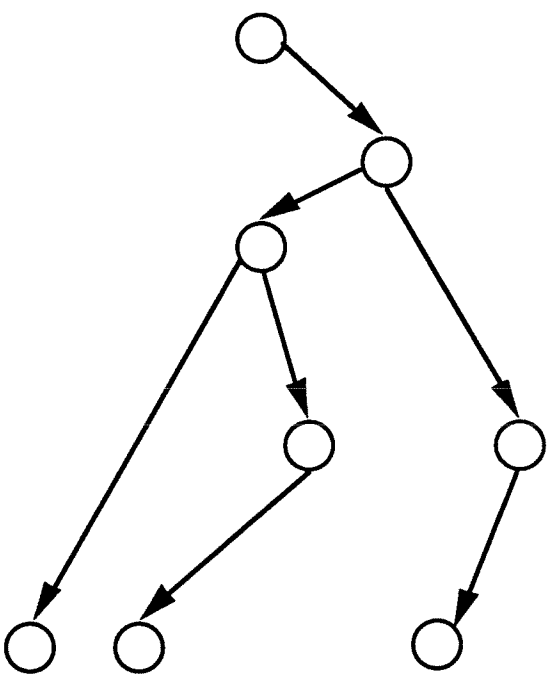
This *broadcast* (*fanout*) requires the optimum number of communication phases, $\lceil log_2 R \rceil$ for R participants, but loads the originating node most heavily. This is the mechanism for *broadcasts* currently used in the *Zipcode* layer described in chapter 3.

Figure 2.6. Broadcast Type #2 Schematic



This *broadcast* requires $\lceil \log_2 R \rceil + 1$ communication phases for R participants. However, it requires at most two sends from any node.

Figure 2.7. Broadcast Type #3 Schematic



This *broadcast* procedure is like Type #1, except that it accepts an extra communication phase in return for off-loading the originating process, which in this scenario sends only one message.

Chapter 3
Multicomputer Communication Layers

Abstract

Sophisticated multicomputer applications require efficient, flexible, convenient underlying communication primitives. In the work described here, *Zipcode,* a new, portable communication library, has been designed, developed, articulated and evaluated. The primary goals were as follows: high efficiency compared to lowest-level primitives, user-definable message receipt selectivity, as well as abstraction of collections of processes and message selectivity to allow multiple, independently conceived libraries to work together without conflict.

Zipcode works atop the Caltech *Reactive Kernel,* a portable, minimalistic multicomputer node operating system. Presently, the *Reactive Kernel* is implemented for Intel iPSC/1, iPSC/2, and Symult s2010 multicomputers and emulated on shared-memory computers as well as networks of Sun workstations. Consequently, *Zipcode* addresses an equally wide audience, and can plausibly be run in other environments.

3.1 Introduction

Wide experience with first-generation point-to-point multicomputer node operating systems (such as Intel's NX) demonstrates the inadequacy of basic typed message systems for large applications. That is, simple message typing does not provide enough degrees-of-freedom or notational elegance in message receipt selectivity for most situ-

ations. As is widely implemented in practical codes, an additional (typically one-shot) message-passing layer and queueing mechanism cover the naked primitives, providing additional flexibility at the application level. The overhead of an application-oriented layer can be made acceptably light, as we indicate below. However, the overheads associated with the underlying typed primitives are viewed excessive in that little or no value is attributable to the hard-wired typing provided by the node operating system itself.

The Caltech *Reactive Kernel* (*RK*), by Seitz and co-workers, was designed with this theme in mind [41,43,52]. These primitives provide *no* message typing at all; they are of high efficiency, but prove too low-level for direct application use. For determinism, pairwise message ordering is preserved. Multiple processes per node are supported, with correctness independent of process placement, subject to finite storage limitations. There is no intra-node shared memory. Finally, no explicit notion of the underlying communication network is enforced on the application (*e.g.,* binary n-cube-oriented limitations/strategies); process placement remains, however, at the discretion of the application.

Application-oriented layers are created to specialize and abstract from the *RK* level on a case-by-case basis; the layers' functionality and, hence, overhead are chosen by the application programmer as part of the overall software design process. The easy-to-understand, concise set of primitives in *RK* is easily ported and, alternatively, readily emulated. Consequently, applications based on *RK* stand an excellent chance of surviving changes of node architecture and communication network. Furthermore, as discussed below, these primitives provide a rational basis for programming medium-grain, shared-memory multiprocessors as well.

Unfortunately, individuality in design of the application message-passing layer leads not only to repetition of effort but also to portability problems between programmers and projects, just as incompatible vendor operating systems do between

diverse multicomputers. These effects are fundamentally unacceptable, because we intend to create high-performance, portable multicomputer codes with potentially long lifetimes. Furthermore, we want to create substantial libraries that can be used together in a single program without the chance of message-passing conflicts because of differing assumptions between those libraries, or with/within the application code itself. Consequently, it is desirable to define a single, encompassing application message-passing layer with high efficiency, portability and extensibility, that will be used well by a wide range of applications. These are the main goals of *Zipcode*.

The *Zipcode* philosophy is as follows. First, only the application can properly define the nature, style, and extent of message-passing receipt selectivity. There are arbitrarily many such patterns of selectivity – they cannot be foreseen or implemented by the node operating system a priori. Consequently, any node operating system that *types* messages is, in general, too restrictive, molds message-passing style and notation unnecessarily, and imposes overhead to overcome such built-in restrictions. Second, there may be arbitrarily many contexts of communication within a given multicomputer application which, for correctness, cannot clash; no node operating system of which we are presently aware supports multiple contexts. Third, the best node operating system is the one that constrains the application least, both in function and overhead. Thus far, *RK* has proven the most elegant underpinning because it imposes essentially no arbitrary restrictions on the communication process, and is not ridden with features of dubious value but noticeable cost.

Zipcode design features can be summarized as follows:

- Operates on process lists as the fundamental communication object with no predilection toward hypercubes, gray codes, or powers of two.
- Uses message classes to decide how process lists are to be abstracted.
- Uses message classes to decide in part on receipt selectivity.
- Uses message contexts to decide in part on receipt selectivity.

- Inheritance techniques are used to derive additional message contexts.
- Five standard pre-defined message classes are provided, including grids.
- "Global operations" – *combine* and *broadcast* – are defined for several of the standard message classes and are extensible to new classes.
- C macros are applied widely to avoid excess overheads.
- Message-debugging capabilities are inherent in message classes.
- The number of contexts is definable and extensible at run time.
- Classes can be added readily.
- Applications can set the current context and utilize terse, readable program notation for message transmissions.

In our empirical experience, carefully coded ad hoc message layers imply a 10-15% overhead in message startup cost compared to bare primitives. We observe comparable overheads for the *Zipcode* system (conservatively, about 20%). As a function of its design, rejection of a message during message selection is nearly as cheap as in ad hoc layers. Message acceptance cost is, however, a function of the complexity of the message class being requested. Queueing of received, undelivered messages is discussed; the present use of a single queue is justified and alternatives are mentioned.

Zipcode has been run extensively on the Intel iPSC/2 and Symult s2010 systems, and on networks of Sun workstations. Performance results (single message transmissions and global operations) are quoted as a function of message length for the Symult s2010 implementation.

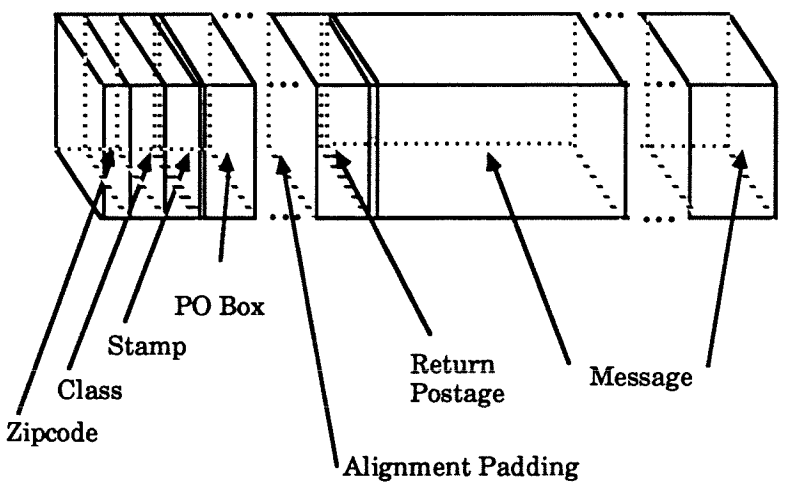
Nearly 70,000 lines of successful application code have already been developed relying on *Zipcode*. Use of the layer as a pedestal for portable scientific/engineering numerical tools is in progress. Thoughts on this and future planned improvements are mentioned in closing.

3.2 *Zipcode* Design Discussion

In second-generation multicomputers, improvements in routing technology allow programmers sensibly to ignore the underlying communication network and conceive of the computers as nodes on a completely connected graph with uniform transmission costs. As Athas and Seitz point out [6], this approximation holds well for small- to medium-sized multicomputers employing their cut-through, wormhole routing technology. Figure 3.2. illustrates performance of application-level primitives on the Symult s2010, which incorporates this routing technology.

Within the loose framework of communicating sequential processes [27], two programming paradigms are commonly used: reactive programming, where processes progress asynchronously with computational decisions driven by the number and variety of messages received, and loosely synchronous programming, where processes progress with intermittent, pre-specified synchronizations. *Zipcode* supports both styles of programming by building on unblocked and blocked *RK* primitives, respectively.

Figure 3.1. Schematic of a *Zipcode* Letter



Letter:	*Zipcode* format of a message
Zipcode:	Integer indicating the letter context
Class:	Integer indicating style of selectivity
PO Box:	Class-dependent structure for selectivity
Envelope:	Preamble structure containing PO Box, class, zipcode
Cover:	The envelope with alignment padding
Stamp:	Length in bytes of the cover

3.2.1 Type *vs.* Class *vs.* Context

A message class is a set of rules and a data specification used for defining message receipt selectivity, and for discriminating correctly among incoming messages. A hypothetical class of messages (call it 'A') might be "messages chosen based on their source, where the source is to be specified by node number and process ID." Given this message class, it's possible to look for a message from one or more acceptable sources, rejecting all others to a queue for future retrieval. For example, we could request "the next message from (node 1, process 0)" or, equally well, "the next message from (node 1 or 2, process 0)." That we can discriminate on source implies that the message must include, however transparently, its source information: in this case, two integers. In *Zipcode*, we call this data the "PO Box."

The message class just discussed would *not* allow discrimination based on the particular aspect of the process that sent a message, nor on the particular contents of a message. These possible deficiencies can be handled in distinct ways. On one hand, we could define a more powerful class (denote 'B'), increasing the contents of its PO Box compared to the 'A' class: "messages chosen based on their source, plus an integer type." Given such a class, messages could be tagged appropriately by the sender to indicate their contents and/or intended use. If we really want to indicate the contents of the message by type, this is probably the most convenient approach. If, however, analogous parts of the communicating processes produce messages that they want to keep exclusively among themselves, addressing their messages in a narrow sense, it is more convenient to define a message context. We call the integer that specifies context the "zipcode" because it states conceptually "where" the message is to go within its destination process(es), but not in detail.

A message context is like a message type, but stronger – knowing the context implies knowing who can participate in the transmission process. So, for example, we could pose receipt selectivity as "Accept a class 'B' message from (node 0, process 0)

in context 6 (or zipcode 6)," where "6" indicates the specific phase of the computation for which the message is intended (such as a linear-algebra subroutine operating on a set of related matrices using processes in a particular logical configuration). So far, context is just an extra integer added for greater flexibility. However, it leads immediately to further interesting capabilities. As stated, being part of a message context implies knowing the participants: in the simplest instance, an explicit list of the participating processes. A message class can specify indentifying information in PO Boxes in a number of ways, and we could imagine altering the semantics of the PO Box to exploit this extra information. First, we could assign an abstract name to each process in the process list. A class 'A' message could be changed to have its receipt selectivity be "messages specified by their context and position (index) of the source process in that context's process list." A request could be "accept an 'A' class message in context 6 from abstract process name 30." Once we abstract the basis of receipt selectivity, context and class information together uniquely identify the message(s) we want to accept; each is insufficient alone.

We need a terse, flexible notation, and message structure to permit multiple contexts and classes to work together. Figure 3.1. illustrates the structure of a *Zipcode* "letter" – a message, plus enabling information: the variable-length envelope/cover including its zipcode, PO Box, and other needed structural data. The postal analogy in *Zipcode* carries quite far because a process creates and mails a letter, first by grabbing and filling out a blank message, then by addressing its envelope, and finally, by posting the entire object. Starting from a list of addressees, a class, and a zipcode context, a canonical data object, a *mailer*, is constructed by *Zipcode* calls. A mailer is the object used when creating, receiving, or posting letters within the system. From it, further contexts of communication may be created via inheritance routines (basically, by correctly deriving a communicating subset of processes and making a new process list for them).

3.2.2 "No Class" Systems

Typical node operating systems are "no class" systems. Specifically, they are systems where the only explicit *class* is "messages identified by a single integer," and *types* are instantiations of that integer. Types are most often bound at compile time by applications, and diverse applications usually attach distinct semantic connotations to the same integer types, implying source-level conflicts. Furthermore, all messages are in the same context, so there is no way to distinguish messages intended for one phase of a process over another, to avoid such conflicts, except by the types themselves.

Broadcast and *combine* operations require extension of typing for their deterministic implementation. It's necessary to discriminate among messages based on their source. Consequently, typed message systems must include extra header information invisible to the user, at least in those messages destined to participate in a global operation – multiple classes, though invisible and inaccessible, play a role even in these systems.

3.2.3 *Reactive Kernel* Primitives

For the purpose of this discussion, we need to define six of the *RK* primitives, fitting neatly into two categories: message-generating (*i.e.*, allocate, receive) and message-consuming (*i.e.*, free, send), as follows:

```
char *msg;
int length, node, pid;
int count, *proc_list;
```

Message-Generating Primitives:

```
msg = xmalloc(length);
msg = xrecv();     /* unblocked */
msg = xrecvb();    /* blocked */
```

Message-Consuming Primitives:

```
xsend(msg, node, pid);
xmsend(msg, count, proc_list);
xfree(msg);
```

Basically, messages are created by `xmalloc()`, sent via `xsend()` or `xmsend()` (multiple destinations), and received via `xrecv()` (unblocked) or `xrecvb()` (blocked). Sending a message is equivalent to an `xfree()` with the side-effect that the message is mailed to the specified destination(s). This represents the complete message-passing notation of *RK*.

3.2.4 *Zipcode* Class-Independent Calls

Zipcode maintains the same basic naming convention and style as *RK*. For all classes, the same calls are used for allocation, sending and receiving letters. Specific classes may define additional calls to increase the convenience of use (see G2-Class calls further below). Small-y calls require specification of the mailer relative to which a letter is to be created, sent or received. Big-Y calls depend on the current mailer context established by `Ypush()/Ypop()` calls. As such, they omit mailer arguments. The `[yY]mail()` calls transmit to all addressees of a mailer.

```
char *letter;
ZIP_MAILER *mailer;
```

Context-setting Primitives:

```
Ypush(mailer);
Ypop();
```

Letter-Generating Primitives:

```
letter = ymalloc(mailer, length);
letter = yrecv(mailer);    /* unblocked */
letter = yrecvb(mailer);   /* blocked */

letter = Ymalloc(length);
```

```
letter = Yrecv();       /* unblocked */
letter = Yrecvb();      /* blocked */
```

Letter-Consuming Primitives:

```
ysend(mailer, letter, node, pid);
ymsend(mailer, letter, count, proc_list);
yfree(letter);

Ysend(letter, node, pid);
Ymsend(letter, count, proc_list);
Yfree(letter);
```

Abstraction to process-list addressees:

```
ymail(mailer, letter);
Ymail(letter);
```

Variations of the basic `[yY]send()` and `[yY]mail()` macros are provided for determining the disposition of the letter's PO Box information. The three versions alternatively use a default value for the PO Box, accept an argument as a pointer to the contents of the PO Box to be used, or assume the PO Box is preset correctly in the letter's envelope. `[yY]mail()` applied to appropriately inherited child mailers, allows specification of arbitrary, user-defined subsets of recipients of the original mailer's addressees. Similarly, variations of the `[yY]recv[b]()` calls are provided to control receipt selectivity based on a default PO Box, a PO Box specified as an extra argument, or obviating PO-Box-based selectivity entirely, accepting the next message in the correct zipcode context. Lower-level calls allow replacement of the receipt-selectivity routine (the delivery function), in order to allow flexible user-defined interpretations of the PO Box data in an incoming letter. The latter feature can be used to define "wildcard" message receipt selectivity, for example.

Any host/node data-format conversions to the cover information are automatically performed without any user intervention. This feature causes additional load only in the host process.

3.2.5 Mailer Creation

Mailers are created through a loose synchronization between the members of the proposed mailer's process list. A single process creates the process list, placing itself first in the list, and initiates the "mailer-open" call with this process information; it's called the "Postmaster" for the mailer, as initiator. The other participants receive the process list as part of the synchronization procedure. A special reactive process, "The Postmaster General," maintains and distributes zipcodes as mailers are opened; essentially the zipcode count is a single location of shared memory.

Class-independent mailer creation:

```
ZIP_MAILER *mailer; /* mailer pointer */
ZIP_CLASS *class;   /* class spec. */
ZIP_ADDRESSEES *addr; /* addressee list */
ZIP_MAILER *parent; /* parent, if any */
void *extra;        /* class extra data */
int *copyflg;       /* copying flags */
short int *zipcode; /* zipcode, if known */
ZIP_MAILER *(**inherit)();  /* overrides for inheritance */

mailer = yopen(class, addr, extra, parent, copyflg, zipcode, inherit);
```

Typical call:

```
mailer = yopen(class, addr, NULL, NULL, NULL, NULL, NULL);
```

3.2.6 Pre-Defined Letter Classes

Y-Class mail is used mainly for *Zipcode* internal mechanisms. The PO Box information is a single short integer type. Global operations cannot be implemented for this class, because of its intentional simplicity.

Z-Class mail is a general purpose class. Process names are abstracted to a single integer (based on position in the process list); receipt-selectivity is based on that

source name. Global operations are implemented for this class, with analogous calling sequences to the G2-Class 2D-grid global operations noted below.

G1-Class mail is a 1D-grid-abstraction class, similar to Z-Class mail.

G2-Class mail is a 2D-grid-abstraction class. A $P \times Q$ grid naming abstraction is attached to the process list; each process is specified by a (p, q) pair (*e.g.*, in the PO Box). Through inheritance, row and column mailers are defined in each process as the appropriate subsets of the 2D grid. This class has received the most extensive use because of the natural application to linear algebra and related computations (see chapters 2, 5).

Class-specific primitives for G2-Class mail have been defined for both higher efficiency and better abstraction. Small-g calls require mailer specification while big-G calls do not, analogous to the y- and Y-type calls defined generically above.

```
int p, q; /* source or destination */
```

Letter-Generating Primitives:

```
letter = g2Recv(mailer,p, q); /* unblocked */
letter = g2Recvb(mailer, p, q); /* blocked */

letter = G2Recv(p, q);    /* unblocked */
letter = G2Recvb(p, q);   /* blocked */
```

Letter-Consuming Primitives:

```
g2Send(mailer, letter, p, q);
G2Send(letter, p, q);
```

Global operations *combine* and *broadcast* (fanout) are defined and have been highly tuned for this class (see schematics of these operations in Figures 2.3, 2.5, 2.6, 2.7). Combines are over arbitrary associative-commutative operators specified by

`(*comb_fn)()`. Broadcasts share data of arbitrary length, assuming all participants know the source:

```
void (*comb_fn)(); /* operation */
void *buffer;      /* data/result */
int size, items;   /* data specifications */

g2_combine(mailer, buffer, comb_fn, size, items);
G2_combine(buffer, comb_fn, size, items);

void *data;        /* data/result */
int length;        /* length of data */
int orig_p, orig_q; /* origin */

g2_fanout(mailer, &data, &length, orig_p, orig_q);
G2_fanout(&data, &length, orig_p, orig_q);
```

G2-Grid mailer creation:

```
int P, Q; /* grid shape */

mailer = g2_grid_open(P, Q, addr, zipcode);
```

A much more general version, `_g2_grid_open()`, (analogous to `yopen()`) is also available. See also appendix B.

G3-Class mail is a 3D-grid-abstraction class. A $P \times Q \times R$ grid naming abstraction is attached to the process list, analogously to the G2-Class 2D-grid primitives. This class should prove very useful in defining operations such as matrix-matrix multiplications in an unrestrictive setting.

3.2.7 The *Zipcode* Queue

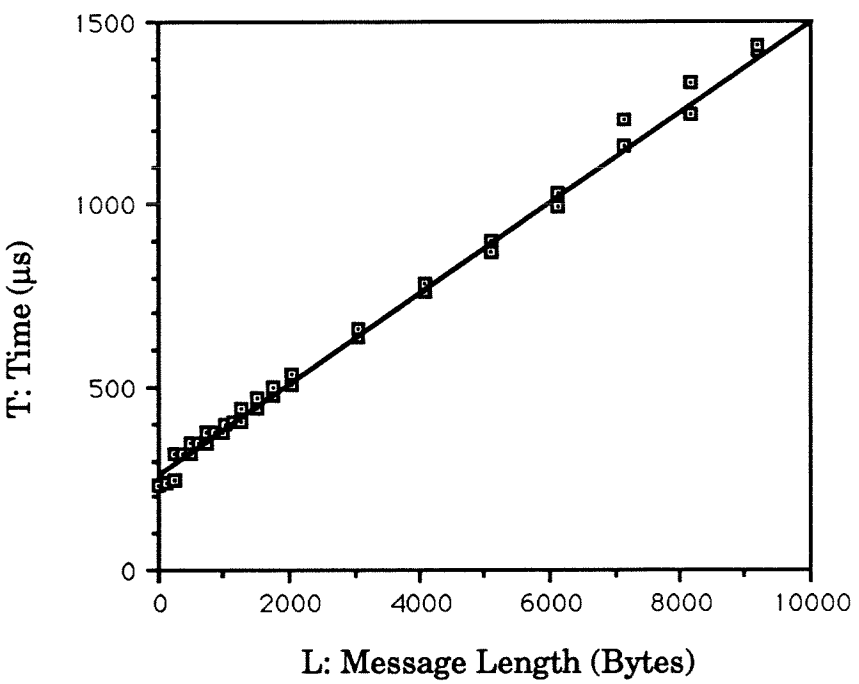
Message selectivity implies that some messages will have to be stored on a queue that the *Zipcode* layer must maintain; there is no push-back mechanism in *RK*. In our experience, multicomputer codes do not accumulate very many messages on the

queue; typically not more than five. We have therefore chosen the simplest possible queueing arrangement: a linked list with linear access from oldest to newest. Hashing by zipcode and/or class could also be implemented, but thus far appears superfluous.

3.3 Performance

We quantify performance in three categories: single transmission timings, broadcast operations, and combine operations, which we consider in turn. For each case, we have restricted our attention to lengths that are even, to avoid severe penalities from data copying (*i.e.*, `bcopy()`) operations that are incurred for odd-length messages.

Figure 3.2. Graph of 2D-Grid Primitive Transmission Timings on a 16-node Symult s2010



A fit yields: $T = 260.25 + 0.12660L\mu s$, where T is time in μs, and L is the message length in bytes. An underlying RK transmission costs approximately $T = 220.0 + 0.1L\mu s$, unoptimized (*vs.* $T = 200.0 + 0.1L\mu s$ optimized).

3.3.1 Single Transmissions

Single-transmission performance is measured using a quiescent ensemble, through which a single *Zipcode* letter is passed around many, many times among random destinations. The performance illustrated in Figure 3.2. is for a 16-node machine, where G2-Class 2D-grid primitives were employed. There is a stair-stepping cost increase as a function of length. This is expected because of 256-byte pages used by *RK* to pass messages. Based on a least-squares fit of the data, we conclude that a reasonably conservative measure for the startup cost of G2-Class primitives is $260.25\mu s$ compared to about $220\mu s$ for the bare *RK* primitives. With optimized compilation, *RK* startup time drops to about $200\mu s$; this savings would be reflected directly in reduced *Zipcode* startup time. Furthermore, no optimizations, either by register keyword usage or optimized compilations, have yet been employed on the *Zipcode* layer. Such optimizations are expected further to improve performance, perhaps as much as $10\mu s$ for the Symult implementation.

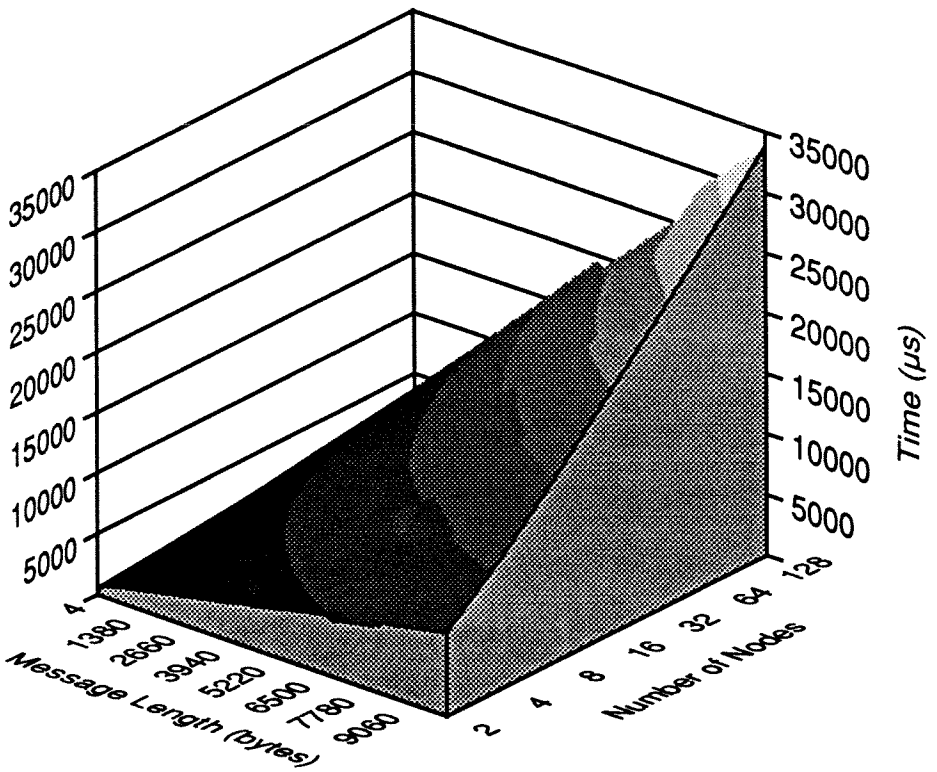
From this performance, we can estimate the systemic granularity of the Symult s2010, at the application level, an important determining factor in the top-down design of concurrent algorithms. For the granularity design equation, $\gamma = 260.25\mu s$, and $\delta \approx .126\mu s$; τ_{+*} is the highly optimized time for the double-precision floating point operation $a = a + b * c$ (*vs.* $13.785\mu s$, unoptimized). See also Seitz *et al.* [41].

3.3.2 Global Operations

There are two global operations *broadcast* (fanout) and *combine* (recursive doubling) [51]. They are extensible to all classes whose receipt selectivity includes source information.

Broadcast is a one-to-all concurrent fanout operation. This has been implemented so that the originating process sends $\lceil \log_2 N \rceil$ letters, for N participants; completion

Figure 3.3. Graph of 2D-Grid Broadcast Primitive Timings on a Symult s2010



is in $\lceil \log_2 N \rceil$ time (see Figure 2.5). Other tree approaches are possible, and have potential merit in load-balancing situations (see Figures 2.6, 2.7). The key feature of *broadcast* is its lesser performance penalty for non-powers of two vis-à-vis *combine*, so it should be used wherever possible. Figures 3.3., 3.4. illustrate performance for the G2-Class 2D-grid primitives. They are only slightly cheaper than the *combine* primitives (Figure 3.5.) for powers of two. For non-powers of two, the difference is more dramatic (see appendix B for further discussion). A least-squares fit of the timing data for lengths from $4 \ldots 10,084$, representative of performance for all node counts from $N = 2 \ldots 128$, is

$$T = (4.1926 \times 10^2 + 4.0138 \times 10^{-1} L)\log_2 N \quad (3.1)$$

$$+ (3.5611 \times 10^2 + 1.4140 \times 10^{-1} L)\mu s \quad (3.2)$$

where T is the time in μs, and L is the length in bytes. Finally, a linear transmission regime for small N has been implemented but is not reflected here. It produces lower overhead when $N \leq 4$.

Combine is the usual associative-commutative global operation, completed in logarithmic time in the number of processes. Figure 3.5. illustrates performance for powers of two, for the G2-Class 2D-grid primitive case while Figure 2.3. illustrates the operation qualitatively for eight participants. Non-powers of two are substantially more expensive; in the worst case, roughly twice the cost of *combine* for the next highest power of two. A least-squares fit of the timing data for lengths from $4 \ldots 10,084$, valid for power-of-two nodes $N = 2 \ldots 128$, is

$$T = (6.0766 \times 10^2 + 4.3976 \times 10^{-1} L)\log_2 N$$

$$+ (2.9994 \times 10^2 + 2.7555 \times 10^{-1} L)\mu s \quad (3.3)$$

where, again, T is the time in μs, and L is the length in bytes.

Both the *broadcast* and *combine* primitives exemplify the high-frequency stepping characteristic, which results from the 256-byte pages used for message transmission by *RK*. At each page boundary, a small additional startup cost is incurred. Furthermore, both operations illustrate a "trough" of improved performance, beginning at lengths somewhat beyond 5,000 bytes, and ending at roughly 8,192 bytes. This trough is thought to be a memory-allocation effect within *RK*; memory pages are managed and dispensed at the lowest level in 8,192 byte (8K) pages.

3.4 "Virtual Distributed Memory"

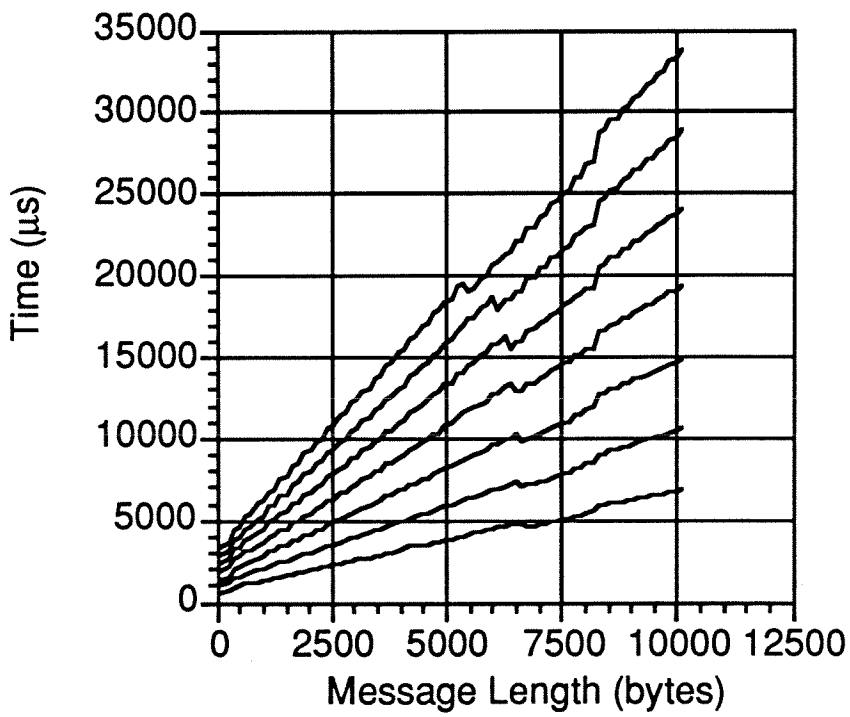
In some circles, it's popular to try to hide distributed memory characteristics by introducing a notion of "global virtual shared memory" that constructs, in principle, a shared-memory paradigm for multicomputing. This follows the tacit assumption that multicomputers are hard to program, while multiprocessors are easy to program, and that shared-memory ideas should be spread to the multicomputer regime insofar as possible, thereby reducing the effort inherent in multicomputer computation. Lacking evidence to suggest efficient realizations of this scheme are possible, we suggest the diametric opposite – "virtual distributed memory." We consider the distributed-memory paradigm to be the more practical model for concurrent computation on medium-grain multiple-instruction, multiple-data multicomputers and multiprocessors alike. We define uniform message-passing primitives for multicomputers and multiprocessors, and achieve portability and high performance for both classes of machines, encapsulating any special features of the memory hierarchy in higher-level data distributions. Data distribution is handled at the application-level, rather than directly and unportably in the communications layer. Applications are written for correctness independent of data distribution, with performance depending heavily on the appropriate data-distribution(s) (*e.g.*, scatter distribution *vs.* linear distribution

in multicomputer linear algebra computations). The effects of locality of data are still left as tuning parameters for the application programmer, but systematically so.

We contend that this approach not only promotes portability, but also rationalizes medium-grain multiprocessor programming, while promoting modular, object-oriented algorithms. Instead of hiding bottlenecks and unscalabilities in the form of shared-memory hotspots and critical sections, the "virtual distributed memory" approach – multiprocessor support for communicating sequential processes – makes explicit the synchronizations, and data dependencies that render multiprocessor code quite challenging to debug or extend to many processors, if not to develop at the outset.

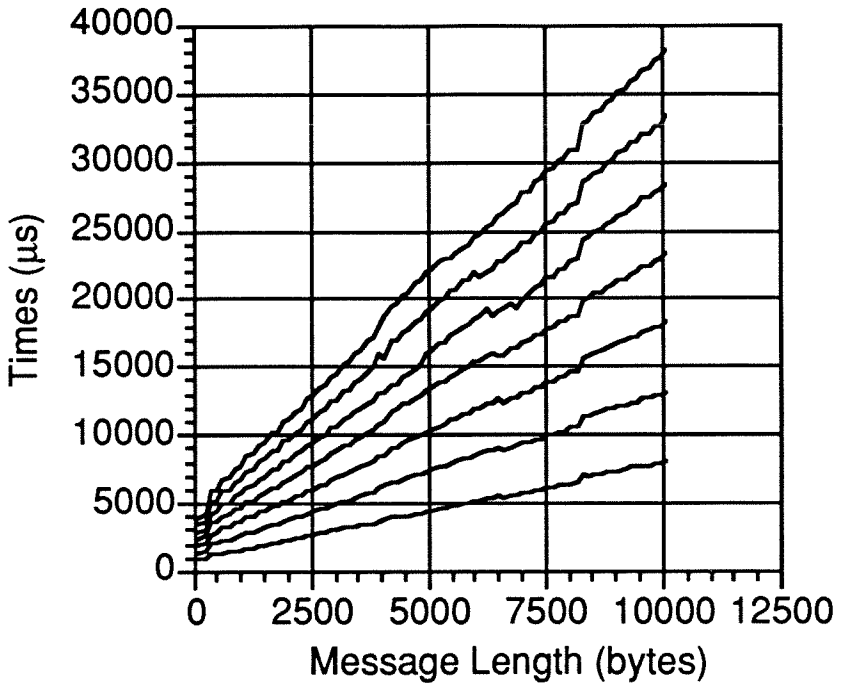
RK ports readily to multiprocessor environments or can be emulated. We are aware of a six-processor Sequent Symmetry implementation by Hamrén and Mattisson, achieving message-startup times of $250\mu s$, competitive with the Symult s2010 multicomputer at roughly $200-220\mu s$ [33,41]; they indicate no explicit per-byte message transmission costs because global memory pointers are used to emulate message passing. This performance results with one process per processor, with much lower performance evident with multiple processes per processor. The Sequent *RK* implementation is based on Unix System V shared-memory primitives and should itself port to other archetypical multiprocessors (*e.g.*, BBN Butterflies, multi-headed Crays). Given the *RK* underpinning, *Zipcode* and the whole body of *Zipcode*-compatible codes port immediately to such multiprocessor environments also. A full discussion of this class of implementations with ported *RK* / *Zipcode* performance will be addressed in a future paper.

Figure 3.4. Graph of 2D-Grid Broadcast Primitive Timings on a Symult s2010



Times quoted for 2, 4, 8, 16, 32, 64, and 128 node configurations. Linear-linear graph exemplifies low- and high-frequency behavior.

Figure 3.5. Graph of 2D-Grid Combine Primitive Timings on a Symult s2010



Times quoted for 2, 4, 8, 16, 32, 64, and 128 node configurations.

3.5 Conclusions, Future Work

In typical multicomputer programs, a layer of communication primitives is constructed above those provided by the operating system. Early point-to-point node operating systems, such as Intel's NX, pre-defined the style and abstraction of message typing. (The decision that messages are typed per se is already a strong assumption.) Consequently, application programs were forced either to conform to the pre-defined style, or to ignore the typing feature, and add additional typing overhead of their own. The Caltech *Reactive Kernel* (*RK*) was designed with this experience in mind, and overcomes the design flaw simply by omitting low-level typing altogether. *RK* consequently presents a set of message primitives that must be augmented for any non-trivial application. Application programs define ad hoc extensions to pattern message passing according to their needs, yet such layers often imply incompatibility between any two application programs or subroutine libraries. The key design principle underlying *Zipcode* is that a single, extensible layer above *RK* is suitable for the vast majority of multicomputer applications, thereby avoiding fundamental incompatibilities before they arise, and also eliminating duplication of effort in application-level message-passing design.

We foresee *RK* as the low-level portability standard for multicomputers and multiprocessors in the 1990's, much as Unix is projected to become the operating system standard of 1990's personal computers, workstations and supercomputers alike. The flexible features of *Zipcode* make it a suitable basis for many application codes and libraries, promoting both portability, and codes of complexity wherever *RK* is implemented or emulated. *Zipcode*, as a portability pedestal for multicomputer applications, encapsulates the interprocessor hardware characteristics, while encouraging the development of codes whose correctness is independent of data distribution. Data distributions can subsequently be used to tune for high performance in a hardware- and application-conscious way.

The key features of *Zipcode* are its design for extensibility, allowing the definition of many-classes of communication and hence message receipt selectivity; support for abstraction of process lists into convenient working groups for communication; the ability to define many non-interfering communication contexts based on process lists with instantiation at runtime rather than compile-time; and the derivation of additional communication contexts through inheritance. Use of *Zipcode* implies acceptable overhead compared to the pervasive one-shot message-passing layers of most multicomputer applications. We asserted at the outset of this work that message-passing generality could be achieved with very little additional overhead compared to one-shot layers. This has subsequently been achieved in *Zipcode*.

For the future, we foresee several classes of improvements and a wider range of implementations, both for new and extant multicomputers, and for medium-grain multiprocessors, as noted above. We foresee the creation of a slightly more extensive pool of general-purpose message classes, based on user feedback. We expect to extend grid-based primitives to provide grid-to-grid data transformations. In the area of debugging, we intend more dramatic growth. We expect to introduce more sophisticated macros and function calls to allow for automated detection of many communication-related errors, as well as better monitoring of the *Zipcode* queue. We do not plan to replace the queueing mechanism at present, but we do expect to make small definitional changes to allow the queueing mechanism to be application re-defined.

Experience with *Zipcode* suggests ways to extend *RK* for overall higher performance of the application. In particular, implementation of *broadcast* and *combine* by *RK* can be posed in a completely general way, consistent with its unrestrictive philosophy; however, such implementations could take advantage of important hardware optimizations and produce much faster primitives overall. The extant *Zipcode* calls would layer transparently above such new primitives (see appendix B).

A numerical toolbox consisting of *Zipcode*-based applications is under construction and refinement. The advantages of the *Zipcode* basis will include portability and compatibility between a number of numerical libraries from several sources, working primarily, at present, with G2-Class 2D-grid primitives and variants. This will be the subject of a future paper.

Chapter 4

Concurrent Data Distributions

Abstract

Mappings between global and local coefficient names are fundamental to the control of data locality in multicomputation. Prior to this work, the prevalent forms of data distribution were the simple linear and scatter distributions, which we relegate to appendix C. Here, we have generalized the linear and scatter distributions by introducing parameters into the closed-form $O(1)$-time, $O(1)$-memory distributions. Now, we can explore the degree of coefficient blocking and scattering incrementally, and test strategies for improvement of concurrent algorithms with these new adjustable parameters. In chapter 5, we demonstrate the effectiveness of these new distributions. Derivations are reserved for appendix C.

4.1 Introduction

We introduce new closed-form $O(1)$-time, $O(1)$-memory data distributions useful not only for concurrent linear algebra and overlying problems that use it, but also generally for concurrent computations. We quantify evaluation costs in Table 4.1.

Every concurrent data structure in our grid-oriented computations is associated with a logical process grid at creation (*cf.*, Figure 4.1., chapters 2 and 3). Vectors are either row- or column-distributed within a two-dimensional process grid. Row-distributed vectors are *replicated* in each process column, and distributed in the

Table 4.1. Data-Distribution Function and Inverse Costs

Distribution:	$\mu(I,P,M)$	$\mu^{-1}(p,i,P,M)$
One-Parameter (ζ)	$5.5554 \times 10^1 \pm 5 \times 10^{-3}$	$4.0024 \times 10^1 \pm 7 \times 10^{-3}$
Two-Parameter (ξ)	$6.1710 \times 10^1 \pm 1 \times 10^{-2}$	$4.2370 \times 10^1 \pm 8 \times 10^{-3}$
Block-Linear (λ)	$5.4254 \times 10^1 \pm 7 \times 10^{-3}$	$3.5404 \times 10^1 \pm 5 \times 10^{-3}$

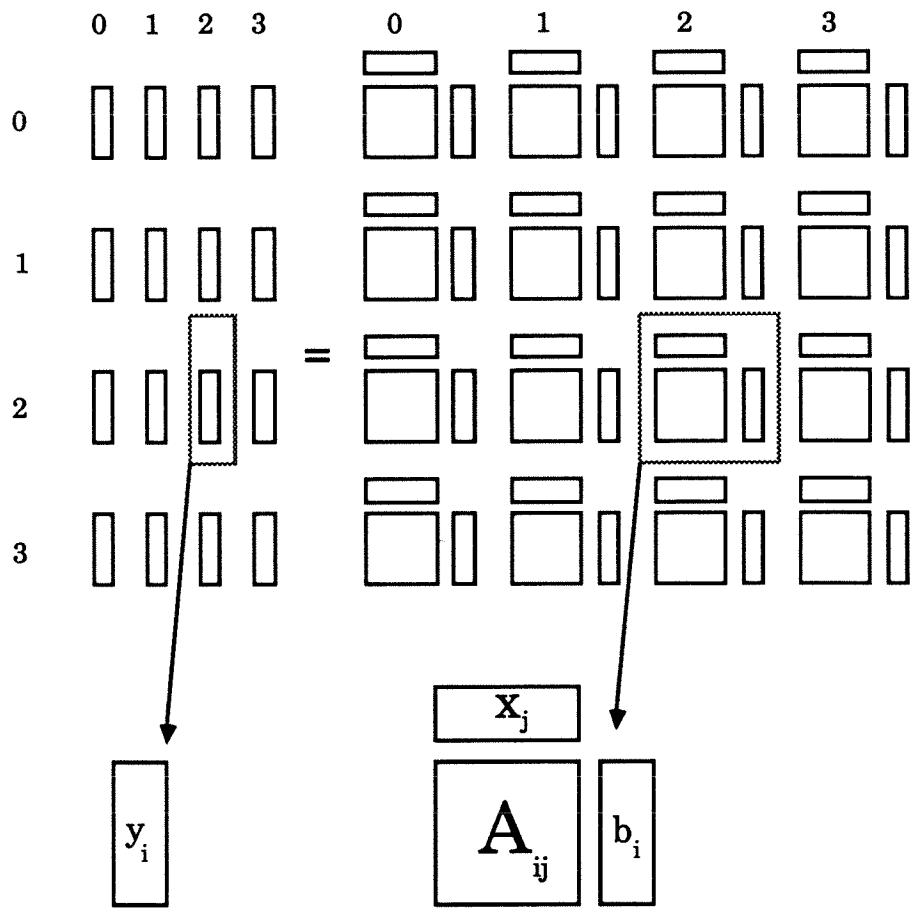
For the data distributions and inverses described here, evaluation time in μs is quoted for the Symult s2010 multicomputer. Cardinality function calls are inexpensive, and fall within lower-order work anyway – their timing is hence omitted. The cheapest distribution function (scatter) costs $\approx 15\mu$s by way of comparison.

process rows. Conversely, column-distributed vectors are replicated in each process row, and distributed in the process columns. Matrices are distributed both in rows and columns, so that a single process owns a subset of matrix rows and columns. This partitioning follows the ideas proposed by Fox *et al.* [20] and others. Within the process grid, coefficients of vectors and matrices are distributed according to one of several data distributions. Data distributions are chosen to compromise between load-balancing requirements and constraints on where information can be calculated in the ensemble.

4.2 New Data Distributions

Definition 4.1 (Data-Distribution Function) *A data-distribution function μ maps three integers $\mu(I,P,M) \mapsto (p,i)$ where I, $0 \leq I < M$, is the global name of a coefficient, P is the number of processes among which all coefficients are to be partitioned, and M is the total number of coefficients. The pair (p,i) represents the process p $(0 \leq p < P)$ and local (process-p) name i of the coefficient $(0 \leq i < \mu^\sharp(p,P,M))$. The inverse distribution function $\mu^{-1}(p,i,P,M) \mapsto I$ transforms the local name i back to the global coefficient name I.*

Figure 4.1. Process Grid Data Distribution of $Ax = b$



Representation of a concurrent matrix, and distributed-replicated concurrent vectors on a 4x4 logical process grid. The solution of $Ax = b$ first appears in x, a column-distributed vector, and then is normally "transposed" (or "converted") via a global *combine* to the row-distributed vector y. See appendix A.

The formal requirements for a data distribution function are as follows. Let $\mathcal{I}^p$ *be the set of global coefficient names associated with process* p, $0 \leq p < P$, *defined implicitly by a data distribution function* $\mu(\bullet, P, M)$. *The following set properties must hold:*

$$\mathcal{I}^{p_1} \cap \mathcal{I}^{p_2} = \emptyset, \ \forall\, p_1 \neq p_2, \quad 0 \leq p_1, p_2 < P \tag{4.1}$$

$$\bigcup_{p=0}^{P-1} \mathcal{I}^p = \{0, \ldots, M-1\} \equiv \mathcal{I}_M. \tag{4.2}$$

Figure 4.2. Example of Process-Grid Data Distribution

$$\begin{pmatrix} A^{0,0} & A^{0,1} & A^{0,2} & A^{0,3} \\ A^{1,0} & A^{1,1} & A^{1,2} & A^{1,3} \\ A^{2,0} & A^{2,1} & A^{2,2} & A^{2,3} \\ A^{3,0} & A^{3,1} & A^{3,2} & A^{3,3} \end{pmatrix}_{\mathcal{G}} = \left(\begin{array}{cccccccccc} a_{0,1} & a_{0,5} & a_{0,2} & a_{0,6} & a_{0,3} & a_{0,7} & a_{0,0} & a_{0,4} & a_{0,8} \\ a_{1,1} & a_{1,5} & a_{1,2} & a_{1,6} & a_{1,3} & a_{1,7} & a_{1,0} & a_{1,4} & a_{1,8} \\ \hline a_{2,1} & a_{2,5} & a_{2,2} & a_{2,6} & a_{2,3} & a_{2,7} & a_{2,0} & a_{2,4} & a_{2,8} \\ a_{3,1} & a_{3,5} & a_{3,2} & a_{3,6} & a_{3,3} & a_{3,7} & a_{3,0} & a_{3,4} & a_{3,8} \\ \hline a_{4,1} & a_{4,5} & a_{4,2} & a_{4,6} & a_{4,3} & a_{4,7} & a_{4,0} & a_{4,4} & a_{4,8} \\ a_{5,1} & a_{5,5} & a_{5,2} & a_{5,6} & a_{5,3} & a_{5,7} & a_{5,0} & a_{5,4} & a_{5,8} \\ a_{6,1} & a_{6,5} & a_{6,2} & a_{6,6} & a_{6,3} & a_{6,7} & a_{6,0} & a_{6,4} & a_{6,8} \\ a_{7,1} & a_{7,5} & a_{7,2} & a_{7,6} & a_{7,3} & a_{7,7} & a_{7,0} & a_{7,4} & a_{7,8} \\ \hline a_{8,1} & a_{8,5} & a_{8,2} & a_{8,6} & a_{8,3} & a_{8,7} & a_{8,0} & a_{8,4} & a_{8,8} \\ a_{9,1} & a_{9,5} & a_{9,2} & a_{9,6} & a_{9,3} & a_{9,7} & a_{9,0} & a_{9,4} & a_{9,8} \\ a_{10,1} & a_{10,5} & a_{10,2} & a_{10,6} & a_{10,3} & a_{10,7} & a_{10,0} & a_{10,4} & a_{10,8} \end{array}\right)$$

An 11x9 array with block-linear rows ($B = 2$) and scattered columns on a 4x4 logical process grid. Local arrays are denoted at left by $A^{p,q}$ where (p,q) is the grid position of the process on $\mathcal{G} \equiv \left(\left\{(\lambda_2, \lambda_2^{-1}, \lambda_2^{\sharp}); P = 4, M = 11\right\}, \left\{(\sigma_1, \sigma_1^{-1}, \sigma_1^{\sharp}); Q = 4, N = 9\right\}\right)$. Coefficient Subscripts (*i.e.*, $a_{I,J}$) are the global (I,J) indices.

The cardinality of the set $\mathcal{I}^p$, is given by $\mu^{\sharp}(p,P,M)$. We attach an implicit ordering to the coefficient sets as follows:

$$\mathcal{I}^p = (I_0^p, I_1^p, \ldots, I_{\mu^{\sharp}(p,P,M)-1}^p), \tag{4.3}$$

where

$$I_i^p \equiv \mu^{-1}(p,i,P,M). \tag{4.4}$$

Generically, we will denote distribution functions that work on matrix columns and row-distributed vectors by μ, and matrix rows and column-distributed vectors by ν.

The linear and scatter data-distribution functions are most often defined. We generalize these functions (by blocking and scattering parameters) to incorporate practically important degrees of freedom. These generalized distribution functions yield optimal static load balance as do the ungeneralized functions described in [56]

(see also appendix C) for unit block size, but differ in coefficient placement. This distinction is technical, but necessary for efficient implementations.

Definition 4.2 (Generalized Block-Linear) *The definitions for the generalized block-linear distribution function, inverse, and cardinality function are*

$$\begin{aligned}
\lambda_B(I,P,M) &\mapsto (p,i), \\
p &\equiv P-1-\max\left(\left\lfloor\frac{I_B^{\text{rev}}}{l+1}\right\rfloor,\left\lfloor\frac{I_B^{\text{rev}}-r}{l}\right\rfloor\right), && (4.5)\\
i &\equiv I-B\left(pl+\Theta^1\left(p-(P-r)\right)\right), && (4.6)
\end{aligned}$$

while

$$\begin{aligned}
\lambda_B^{-1}(p,i,P,M) &\equiv i+B\left(\left(pl+\Theta^1\left(p-(P-r)\right)\right)\right), && (4.7)\\
\lambda_B^{\sharp}(p,P,M) &\equiv B\left(\left\lfloor\frac{b+p}{P}\right\rfloor-\theta\right)+(M \bmod B)\theta, && (4.8)
\end{aligned}$$

where B denotes the coefficient block size,

$$b = \begin{cases} \frac{M}{B} & \text{if } M \bmod B = 0 \\ \left\lfloor\frac{M}{B}\right\rfloor+1 & \text{otherwise,} \end{cases} \qquad (4.9)$$

$$I_B = \left\lfloor\frac{I}{B}\right\rfloor,\quad I_B^{\text{rev}} = b-1-I_B\ , \qquad (4.10)$$

$$l = \left\lfloor\frac{b}{P}\right\rfloor,\quad r = b \bmod P\ , \qquad (4.11)$$

$$\Theta^k(t) \equiv \begin{cases} 0 & t \le 0 \\ t^k & t>0,\ k>0 \\ 1 & t>0,\ k=0 \end{cases}\ , \qquad (4.12)$$

$$\theta = \left\lfloor\frac{p+1}{P}\right\rfloor\Theta^0(M \bmod B)\ , \qquad (4.13)$$

and where $b \geq P$.

For $B = 1$, *a load-balance-equivalent variant of the common linear data-distribution function is recovered. The general block-linear distribution function divides coefficients among the* P *processes* $p = 0, \ldots, P-1$ *so that each* $\mathcal{I}^p$ *is a set of coefficients with contiguous global names, while optimally load-balancing the* b *blocks among the* P *sets. Coefficient boundaries between processes are on multiples of* B. *The maximum possible coefficient imbalance between processes is* B. *If* $M \bmod B \neq 0$, *the last block in process* $P-1$ *will be foreshortened.*

Definition 4.3 (Generalized Block-Scatter) *The generalized block-scatter distribution, inverse and coefficient-cardinality function are, in turn, as follows:*

$$\sigma_B(I, P, M) \equiv (P - 1 - (I_B^{\text{rev}} \bmod P), \tag{4.14}$$

$$B\left(\left\lfloor \frac{b+p}{P} \right\rfloor - 1 - \left\lfloor \frac{I_B^{\text{rev}}}{P} \right\rfloor\right) + I \bmod B\Bigg) \mapsto (p, i), \tag{4.15}$$

$$\sigma_B{}^{-1}(p, i, P, M) \equiv B\left(((b+p) \bmod P) + P\left\lfloor \frac{i}{B} \right\rfloor\right) + (i \bmod B) \mapsto I, \tag{4.16}$$

$$\sigma_B^{\sharp}(p, P, M) \equiv \lambda_B^{\sharp}(p, P, M), \tag{4.17}$$

where B, I_B, b *and so forth are as defined above.*

For $B = 1$, *a load-balance-equivalent variant of the common scatter distribution is recovered, and the divisibility condition defining* b *becomes redundant. The generalized block-scatter distribution function divides coefficients into* B*-sized blocks with contiguous global names, and scatters such blocks among the* P *processes* $p = 0, \ldots, P-1$, *while optimally load-balancing the* b *blocks among the* P *processes. If* $M \bmod B \neq 0$, *the last block in process* $P-1$ *will be foreshortened.*

Definition 4.4 (Parametric Functions) *To allow greater freedom in the distribution of coefficients among processes, we define a new, two-parameter distribution function family, ξ. The B blocking parameter (just introduced in the block-linear and block-scatter functions) is mainly suited to the clustering of coefficients that must not be separated by an interprocess boundary. Increasing B worsens the static load balance. Adding a second scaling parameter S (of no impact on the static load balance) allows the distribution to scatter coefficients to a greater or lesser degree, directly as a function of this one parameter. The two-parameter distribution function, inverse and cardinality function are defined below. The one-parameter distribution function family, ζ, occurs as the special case $B = 1$, also as noted below:*

$$\xi_{B,S}(I,P,M) \;\mapsto\; (p,i) \;\equiv\; \begin{cases} (p_0,i_0) & \Lambda_0 \geq l_S \\ (p_1,i_1) & \Lambda_0 < l_S \end{cases}, \tag{4.18}$$

where

$$l_S \;\equiv\; \left\lfloor \frac{l}{S} \right\rfloor, \quad \Lambda_0 \;\equiv\; \left\lfloor \frac{i_0}{BS} \right\rfloor, \tag{4.19}$$

$$(p_0,i_0) \;\leftarrow\; \lambda_B(I,P,M), \tag{4.20}$$

$$I_{BS} \;=\; p_0 l_S + \Lambda_0, \tag{4.21}$$

$$p_1 \;\equiv\; I_{BS} \bmod P\ , \tag{4.22}$$

$$i_1 \;\equiv\; BS \left\lfloor \frac{I_{BS}}{P} \right\rfloor + (i_0 \bmod BS), \tag{4.23}$$

with

$$\zeta_{B,S}(I,P,M) \;\equiv\; \xi_{1,S}(I,P,M), \tag{4.24}$$

$$\xi^{\sharp}_{B,S}(p,P,M) \;\equiv\; \lambda^{\sharp}_{B}(p,P,M), \tag{4.25}$$

$$\zeta^{\sharp}_{S}(p,P,M) \;\equiv\; \lambda^{\sharp}_{1}(p,P,M), \tag{4.26}$$

and where r, b, and so forth are as defined above. The inverse distribution function ξ^{-1} *is defined as follows:*

$$\xi_{B,S}^{-1}(p,i,P,M) \;\mapsto\; I \;=\; \lambda_B^{-1}(p^*,i^*,P,M), \tag{4.27}$$

$$(p^*,i^*) \;\equiv\; \begin{cases} (p,i) & \Lambda \geq l_S \\ (p_2,i_2) & \Lambda < l_S\ , \end{cases} \tag{4.28}$$

$$\Lambda \;\equiv\; \left\lfloor \frac{i}{BS} \right\rfloor, \quad I_{BS}^* \;=\; p + \Lambda P\ , \tag{4.29}$$

$$p_2 \;\equiv\; \left\lfloor \frac{I_{BS}^*}{l_S} \right\rfloor, \tag{4.30}$$

$$i_2 \;\equiv\; BS\,(I_{BS}^* \bmod l_S) + (i \bmod BS), \tag{4.31}$$

with

$$\zeta_S^{-1}(p,i,P,M) \;\equiv\; \xi_{1,S}^{-1}(p,i,P,M). \tag{4.32}$$

For $S = 1$*, a variant block-scatter distribution results (but not* σ*, in general), while for* $S \geq S_{crit} \equiv \lfloor l/2 \rfloor + 1$*, the generalized block-linear distribution function is recovered. See also appendix C.*

Definition 4.5 (Data Distributions) *Given a data-distribution function family* $(\mu, \mu^{-1}, \mu^\sharp)$ *(*$(\nu, \nu^{-1}, \nu^\sharp)$*), a process list with P (Q) participants, M (N) as the number of coefficients, and a row (respectively, column) orientation, a row (column) data distribution* $\mathcal{G}^{row}$ *(*$\mathcal{G}^{col}$*) is defined as:*

$$\mathcal{G}^{row} \equiv \left\{(\mu, \mu^{-1}, \mu^\sharp); P, M\right\},$$

respectively,

$$\mathcal{G}^{col} \equiv \left\{(\nu, \nu^{-1}, \nu^\sharp); Q, N\right\}.$$

A two-dimensional data distribution may be identified as consisting of a row and column distribution defined over a two-dimensional process grid of $P \times Q$ processes, as $\mathcal{G} \equiv (\mathcal{G}^{row}, \mathcal{G}^{col})$.

Further discussion and detailed comparisons on data-distribution functions are offered in appendix B. Figure 4.2. illustrates the effects of linear and scatter data-distribution functions on a small rectangular array of coefficients.

Chapter 5

Concurrent Sparse Linear Algebra

Abstract

Efficient sparse linear algebra *cannot* be achieved as a straightforward extension of the dense case, even for concurrent implementations. This chapter details a new, general-purpose unsymmetric sparse LU factorization code built on the philosophy of Harwell's *MA28*, with variations. We apply this code in the framework of Jacobian-matrix factorizations, arising from Newton iterations in the solution of nonlinear systems of equations (see chapter 6). Serious attention has been paid to the data-structure requirements, complexity issues and communication features of the algorithm. Key results include reduced communication pivoting for both the "analyze" A-mode and repeated B-mode factorizations, and effective application of the general-purpose data distributions introduced in chapter 4, which prove useful in the incremental trade-off of process-column load balance in LU factorization against triangular solve performance. Future planned efforts in concurrent sparse linear algebra are cited in conclusion.

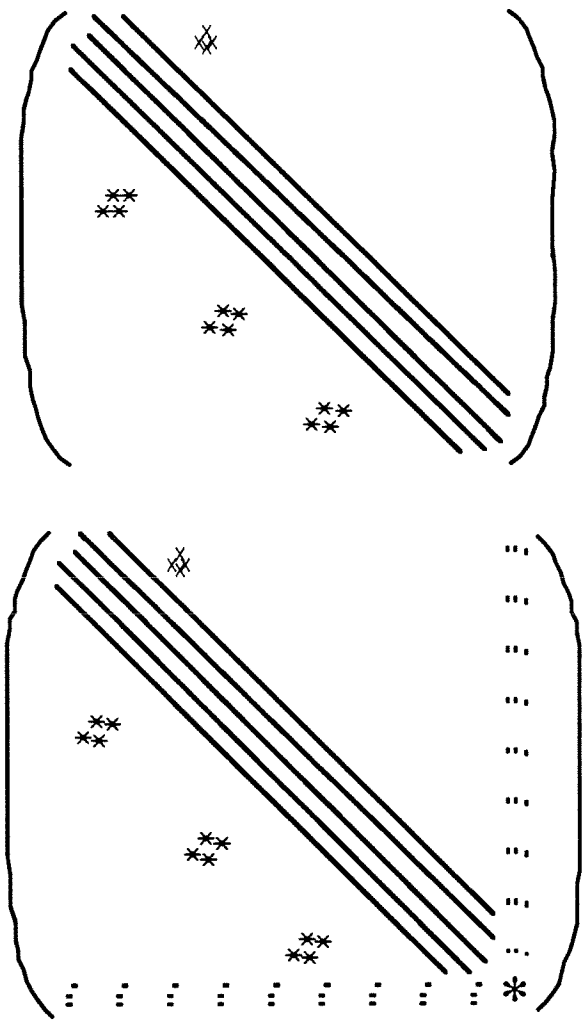
5.1 Introduction

The topic of this chapter is the implementation and concurrent performance of sparse, unsymmetric LU factorization for medium-grain multicomputers. Our target hardware is distributed-memory, message-passing concurrent computers such as

the Symult s2010 and Intel iPSC/2 systems. For both of these systems, efficient cut-through *wormhole* routing technology provides pair-wise communication performance essentially independent of the spatial location of the computers in the ensemble [6]. Message-passing performance, portability and related issues impacting multicomputer algorithms have already been detailed in earlier chapters.

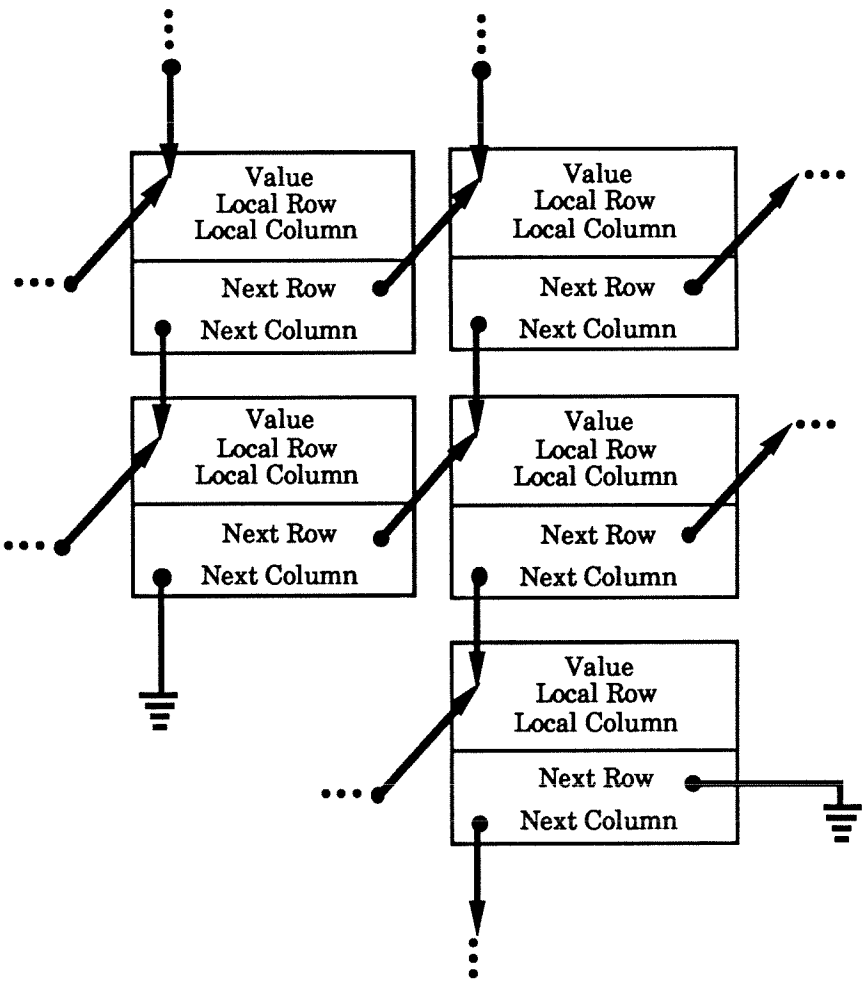
Questions of linear-algebra performance are pervasive throughout scientific and engineering computation. The need for high-quality, high-performance linear algebra algorithms (and libraries) for multicomputer systems therefore requires no attempt at justification. The motivation for the work described here has a specific origin, however. Our main higher-level research goal is the concurrent dynamic simulation of systems modelled by ordinary differential and algebraic equations; specifically, dynamic flowsheet simulation of chemical plants (*e.g.*, coupled distillation columns); see chapter 6. Efficient sequential integration algorithms solve staticized nonlinear equations at each time point via modified Newton iteration (*cf.*, [9, Chapter 5]). Consequently, a sequence of structurally identical (or nearly identical) linear systems must be solved; the matrices are finite-difference approximations to Jacobians of the staticized system of ordinary differential-algebraic equations. These Jacobians are large, sparse and unsymmetric for our application area. In general, they possess both band and significant off-band structure. Generic structures are depicted in Figure 5.1. This work should also bear relevance to electric power network/grid dynamic simulation where sparse, unsymmetric Jacobians also arise, and also elsewhere.

Figure 5.1. Example Jacobian Matrix Structures



In chemical-engineering process flowsheets, Jacobians with main band structure, and lower-triangular structure (feedforwards), upper-triangular structure (feedbacks), and borders (global or artificially restructured feedforwards and/or feedbacks) are common.

Figure 5.2. Schematic of Linked-list Entry Structure for Sparse Matrices



A single entry consists of a double-precision value (8 bytes), the local row (i) and column (j) index (2 bytes each), a "Next Column Pointer" indicating the next current column entry (fixed j), and a "Next Row Pointer" indicating the next current row entry (fixed i), at 4 bytes each. Total: 24 bytes per entry. There are additional storage overheads as well, as detailed in appendix D.

5.2 Design Overview

We solve the problem $Ax = b$ where A is large, and includes many zero entries. We assume that A is unsymmetric both in sparsity pattern and in numerical values. In general, the matrix A will be computed in a distributed fashion, so we will inherit a distribution of the coefficients of A (see Figures 4.1, 4.2). Following the style of Harwell's *MA28* code for unsymmetric sparse matrices, we use a two-phase approach to this solution. There is a first LU factorization called A-mode or "analyze," which builds data structures dynamically, and uses a user-defined pivoting function. The repeated B-mode factorization uses the existing data structures statically to factor a new, similarly structured matrix, with the previous pivotal sequence ("preset pivoting"). B-mode monitors stability with a simple growth factor estimate (see appendix D). In practice, A-mode is repeated whenever instability is detected. The two key contributions of this sparse concurrent solver are reduced communication pivoting, and effective application of new data distributions for better overall performance.

Following Van de Velde [57], we consider the LU factorization of a real matrix A, $A \in \Re^{N \times N}$. It is well known (*e.g.*, [22, pages 117-118]), that for any such matrix A, an LU factorization of the form

$$\mathsf{P}_R A \mathsf{P}_C^T = \hat{L}\hat{U} \tag{5.1}$$

exists, where $\mathsf{P}_R, \mathsf{P}_C$ are square, (orthogonal) permutation matrices, and $\hat{L}, \hat{U}$ are the unit lower-triangular, and upper-triangular factors, respectively. Whereas the pivot sequence is stored (two N-length integer vectors), the permutation matrices are not stored or computed with explicitly. Rearranging, based on the orthogonality of the permutation matrices,

$$A = \mathsf{P}_R^T \hat{L}\hat{U} \mathsf{P}_C. \tag{5.2}$$

We factor A with implicit pivoting (no rows or columns are exchanged explicitly as a result of pivoting). Therefore, we do not store $\hat{L}, \hat{U}$ directly, but instead:

$$L = \mathsf{P}_R^T \hat{L} \mathsf{P}_C, \tag{5.3}$$

$$U = \mathsf{P}_R^T \hat{U} \mathsf{P}_C. \tag{5.4}$$

Consequently,

$$\hat{L} = \mathsf{P}_R L \mathsf{P}_C^T, \tag{5.5}$$

$$\hat{U} = \mathsf{P}_R U \mathsf{P}_C^T, \tag{5.6}$$

and

$$A = L(\mathsf{P}_C^T \mathsf{P}_R)U. \tag{5.7}$$

The "unravelling" of the permutation matrices is accomplished readily (without implication of additional interprocess communication) during the triangular solves (see appendix D).

For the sparse case, performance is more difficult to quantify than for the dense case, but, for example, banded matrices with bandwidth β can be factored with $O(\beta^2 N)$ work; we expect sub-cubic time complexity in N for reasonably sparse matrices, and strive for sub-quadratic time complexity for very sparse matrices. The triangular solves can be accomplished in work proportional to the number of entries in the respective triangular matrix L or U. The pivoting strategy is treated as a parameter of the algorithm and is not pre-determined. We can consequently treat the pivoting function as an application-dependent function, and sometimes tailor it to special problem structures (see [53, Section 7]) for higher performance. As for all sparse solvers, we also seek sub-quadratic space complexity in N, attained by storing matrix entries in linked-list fashion, as illustrated in Figure 5.2.

For further discussion of LU factorizations and sparse matrices, see [22,16].

5.3 Reduced-Communication Pivoting

At each stage of the concurrent LU factorization, the pivot element is chosen by the user-defined pivot function. Then, the pivot row (new row of U) must be broadcast, and pivot column (new column of L) must be computed and subsequently broadcast on the logical process grid (*cf.*, Figure 4.1.), vertically and horizontally, respectively. Note that these are interchangeable operations. We use this degree-of-freedom to reduce the communication complexity of particular pivoting strategies, while impacting the effort of the LU factorization itself negligibly. First we describe the methodology, then we explain its advantages qualitatively in the following section.

5.3.1 Formalism

We define two "correctness modes" of pivoting functions: "first row fanout" and "first column fanout." Each of these modes has specific exit requirements for the user-defined pivot function. After defining these correctness modes, we indicate how partial row pivoting, partial column pivoting and preset pivoting satisfy these correctness modes. Before delving into this discussion, we defined the terms to be used.

Definition 5.1 (Terms) *We consider a process grid of shape $P \times Q$. Then, we denote by $\hat{p}$ ($\hat{q}$) the process row (resp., column) containing the matrix row (resp., column) with the current pivot element. By $\hat{\imath}$, we denote the local matrix row index in process row $\hat{p}$, and by $\hat{\jmath}$, the local matrix column in process column $\hat{q}$. The quadruple $\hat{p}, \hat{q}, \hat{\imath}, \hat{\jmath}$ uniquely specifies the location of the current pivot element in the ensemble, and in the matrix itself. By k, we denote the current iteration of the LU factorization procedure in the following commentary.*

Lemma 5.1 (First Row Fanout) *If the "first row fanout" correctness mode exit conditions for the pivot function are defined as follows:*

- All processes have the correct $\hat{p}$,
- The pivot process row (row $\hat{p}$) has the correct $\hat{q}$ and $\hat{\imath}$,
- The pivot process has the correct pivot value and $\hat{\jmath}$,

then, the LU Factorization with row broadcast preceding column broadcast is correct (apart from pivoting-strategy induced pathologies[1]), provided that the local values for $\hat{p}$, $\hat{q}$, $\hat{\imath}$, $\hat{\jmath}$, and the pivot value are also broadcast whenever a row or column broadcast is performed.

Proof For "first row fanout," the universal knowledge of $\hat{p}$ and knowledge of the pivot matrix row $\hat{\imath}$ by the pivot process row, allow the correct vertical broadcast of this new row of U. In addition, the broadcast extends the correct value of $\hat{q}$ to all process columns, and the correct value of $\hat{\jmath}$ and the pivot value are extended to the entire process column $\hat{q}$. Since all processes now have the correct $\hat{q}$, the column broadcast becomes feasible, once the multiplier column has been computed in process column $\hat{q}$. Since the process column $\hat{q}$ has both the correct value of $\hat{\jmath}$, and the correct pivot value, the multiplier (L) column may be correctly formed and then broadcast among the process columns. The incidental broadcast of the pivot value with the multiplier column broadcast extends the correct pivot value to all processes. Each process now contains the correct pivot-index quadruple, and the correct pivot value, and hence can proceed with the local elimination step for iteration k. ■

Lemma 5.2 (Partial Column Pivoting Correctness) *Partial column pivoting can be set up to satisfy "first row fanout" correctness, assuming a "strong" row data distribution function, as described exclusively in chapter 4. See also appendices C, D.*

Proof For partial column pivoting, the kth row is eliminated at the kth step of the factorization. From this fact, each process can derive the process row $\hat{p}$ and $\hat{p}$-local

[1]In general, specialized pivoting strategies may not find a stable pivot if they fail to perform complete pivoting. This is always at issue in LU factorization, and unrelated to the reduced-communication strategies in our present discussion.

matrix row $\hat{\imath}$ using the row data distribution function. Having identified themselves, the pivot-row processes can look for the largest element in local matrix row $\hat{\imath}$ and choose the pivot element globally among themselves via a *combine*. The remaining processes perform no pivoting-related communication. At completion, this places $\hat{q}$, $\hat{\jmath}$ and the pivot value in the entire pivot process row. This completes the requirements for the "first row fanout" correctness mode. ■

Lemma 5.3 (Preset Pivoting Correctness) *Preset pivoting can be set up to satisfy "first row fanout" correctness, assuming that the pivot quadruple for the kth iteration is pre-stored in each process. For large problems, storing all the pivot quadruples in each process causes an unacceptable memory unscalability. However, this unscalability can be removed cheaply with occasional broadcast communications ("pivot windowing") as described in appendices C, D.*

Proof For preset pivoting, the kth elimination row and column are known, stored in each process as $\hat{p}, \hat{\imath}, \hat{q}, \hat{\jmath}$. Furthermore, the pivot process contains the correct pivot value. Hence, preset pivoting satisfies the requirements of the "first row fanout" correctness mode. ■

Lemma 5.4 (First Column Fanout) *If the "first column fanout" correctness mode exit conditions for the pivot function are defined as follows:*

- All processes must know $\hat{q}$,
- The entire pivot process column must know $\hat{\jmath}$, the pivot value, and $\hat{p}$,
- The pivot process must, in addition, know $\hat{\imath}$,

then, the LU Factorization with column broadcast preceding row broadcast is correct (apart from pivoting-strategy induced pathologies), provided that the local values for $\hat{p}$, $\hat{q}$, $\hat{\imath}$, $\hat{\jmath}$, and the pivot value are also broadcast whenever a row or column broadcast is performed.

Proof - For "first column fanout," the entire pivot process column knows the correct pivot value, and local column of the pivot. Hence, the multiplier column may be computed by dividing the pivot matrix column by the pivot value. This column of L may then be broadcast horizontally, including the pivot value, $\hat{p}$ and $\hat{\imath}$ as additional salient information. After this step, the entire ensemble has the correct pivot value, and $\hat{p}$; in addition, the pivot process row has the correct $\hat{\imath}$. Hence, the pivot matrix row may be identified and broadcast by the pivot process row. This second broadcast completes the needed information in each process for effecting the kth elimination step. ■

Lemma 5.5 (Partial Row Pivoting Correctness) *Partial row pivoting can be set up to satisfy "first column fanout" correctness, assuming a "strong" column data distribution function, as described exclusively in chapter 4. See also appendices C, D.*

Proof For partial row pivoting, the kth column is eliminated at the kth step of the factorization. From this fact, each process can derive the process column $\hat{q}$ and $\hat{q}$-local matrix row $\hat{\jmath}$ using the column data distribution function. Having identified themselves, the pivot-column processes can individually select their largest element in the local matrix column $\hat{\jmath}$ and then choose (among themselves) the globally largest pivot element via a *combine*. The remaining processes perform no pivot-related communication. At completion, this places the correct $\hat{p}$, $\hat{\imath}$ and pivot value in the entire pivot process column. This completes the requirements for the "first column fanout" correctness mode. ■

Hence, when using partial row or partial column pivoting, only local *combines* of the pivot process column (respectively, row) are needed. The other processes don't participate in the *combine*, as they must without this methodology. Preset pivoting implies no pivoting communication, except very occasionally (*e.g.*, 1 in 5000 times) as detailed in appendix D to remove memory unscalabilities. This pivoting approach

is a direct savings, gained at a negligible additional broadcast overhead. See both of the appendices mentioned above.

5.3.2 Advantages

For A-mode factorization, communication costs are reduced if we utilize either partial row or partial column pivoting. Specifically, we eliminate the *combine* over all grid processes in favor of a combine only on the pivot column (resp., row). The other processes in the grid overlap part or all of fixed costs of entering the broadcast function in "receive mode." Furthermore, we no longer synchronize all the processes, so we don't necessarily pay the full performance penalty of imbalances. The less synchronous algorithm may allow some imbalances to wash out. Since the only *combine* operation is now over a subgrid, the use of non-powers of two for the grid shape can also be contemplated reasonably.

For B-mode factorization, communication costs are always reduced dramatically. Preset pivoting implies no generic pivot-induced communication, as just described in the previous section (see clarifications in appendix D). The only iteration-by-iteration communication steps are the broadcasts of the rows and columns of U and L, respectively. It is even more reasonable to consider non-powers-of-two for the grid shapes for B-mode, in view of the absence of *combines*. Actually, two *combines* are used, one before the B-mode factorization, and one after it; they implement the growth-factor test. This is, however, a secondary overhead to the algorithm.

Some applications will use A-mode more frequently than others; some applications may conceivably use B-mode only seldom or not at all. It's important to note that we have made contributions to improved performance for both the more expensive A-mode factorization procedure, and for the cheaper repeated B-mode factorization, that assumes we are factoring a matrix of the known structure built-up in A-mode.

One reason we care about non-power-of-two grid shapes is that they allow finer

optimization of the overall performance, both when considering the work involved in calculating the matrix in the ensemble, and also considering the triangular solves. So, not only does the reduction in pivotal communication speedup both A-mode and B-mode factorizations, it also implicitly allows tuning for better triangular solve performance, since we can select grid shapes more finely. Our main contribution to improved triangular solves comes, however, from the use of compromise data distributions and this is discussed next.

5.4 Performance *vs.* Scattering

Consider a fixed logical process grid of R processes, with $P \times Q = R$. For the sake of argument, assume partial row pivoting during LU factorization for the retention of numerical stability. Then, for the LU factorization, it is well known that a scatter distribution is "good" for the matrix rows, and optimal were there no off-diagonal pivots chosen. Furthermore, the optimal column distribution is also scatter, because columns are chosen in order for partial row pivoting. Compatibly, a scatter distribution of matrix rows is also "good" for the triangular solves. However, for triangular solves, the best column distribution is linear, because this implies less intercolumn communication, as we detail below. In short, the optimal configurations conflict, and because explicit redistribution is expensive, a static compromise must be chosen. We address this need to compromise through the one-parameter distribution function ζ described in the previous chapter, offering a variable degree of scattering via the S-parameter. To first order, changing S does not affect the cost of computing the Jacobian (assuming columnwise finite-difference computation), because each process column works independently (also, see comments in the next chapter).

It's important to note that triangular solves derive no benefit from $Q > 1$. The standard column-oriented solves keep one process column active at any given time. For any column distribution, the updated right-hand-side vectors are retransmitted W

times (process column-to-process column) during the triangular solve – whenever the active process column changes. There are at least $W_{min} \equiv Q - 1$ such transmissions (linear distribution), and at most $W_{max} \equiv N - 1$ transmissions (scatter distribution). The complexity of this retransmission is $O(WN/P)$, representing quadratic work in N for $W \sim N$.

The time complexity for a sparse triangular solve is proportional to the number of elements in the triangular matrix, with a low leading coefficient. Often, there are $O(N^{1.x})$ with $x < 1$ elements in the triangular matrices, including fill. This operation is then $O(N^{1.x}/P)$, which is less than quadratic in N. Consequently, for large W, the retransmission step is likely of greater cost than the original calculation. This retransmission effect constrains the amount of scattering and size of Q in order to have any chance of concurrent speedup in the triangular solves.

Using the one-parameter distribution with $S \geq 1$ implies that $W \approx N/S$, so that the retransmission complexity becomes $O(N^2/SP)$. Consequently, we can bound the amount of retransmission work by picking S sufficiently large. Clearly, $S = S_{crit}$ is a hard upper bound, because we reach the linear distribution limit at that value of the parameter (see chapter 4). We suggest picking $S \approx 10$ as a first guess, and $S \sim \sqrt{N}$, more optimistically. The former choice basically reduces retransmission effort by an order of magnitude. Both examples in the following section illustrate the effectiveness of choosing S by these heuristics.

The two-parameter ξ distribution can be used on the matrix rows to trade-off load balance in the factorizations and triangular solves against the amount of (communication) effort needed to compute the Jacobian. In particular, a greater degree of scattering can dramatically increase the time required for a Jacobian computation (depending heavily on the underlying equation structure and problem), but importantly reduce load imbalance during the linear algebra steps. The communication overhead caused by multiple process rows suggests shifting toward smaller P and

larger Q (a squatter grid), in which case greater concurrency is attained in the Jacobian computation, and the additional communication previously induced is then somewhat mitigated. The one-parameter distribution used on the matrix columns then proves effective in controlling the cost of the triangular solves by choosing the minimally allowable amount of column scattering.

Let's make explicit the performance objectives we consider when tuning S, and, more generally, when tuning the grid shape $P \times Q = R$. In the modified Newton iteration, for instance, a Jacobian factorization is reused until convergence slows unacceptably. An "LU Factorization + Backsolve" step is followed by η "Forward + Backsolves," with $\eta \sim O(1)$ typically (and varying dynamically throughout the calculation). Assuming an averaged η, say η^* (perhaps as large as five [9]), then our first-level performance goal is a heuristic minimization of

$$T_{LU} + (\eta^* + 1)T_{Back} + \eta^* T_{Forward} \tag{5.8}$$

over S for fixed P, Q. $\eta^* > 1$ more heavily weights the reduction of triangular solve costs *vs.* B-mode factorization than we might at first have assumed, placing a greater potential gain on the one-parameter distribution for higher overall performance. We generally want heuristically to optimize

$$T_{Jac} + T_{LU} + (\eta^* + 1)T_{Back} + \eta^* T_{Forward} \tag{5.9}$$

over S, P, Q, R. Then, the possibility of fine-tuning row and column distributions is important, as is the use of non-power-of-two grid shapes.

5.5 Performance

5.5.1 Order 13040 Example

We consider an order 13040 banded matrix with a bandwidth of 326 under partial row pivoting. For this example, we have compiled timing results for a 16x12 process grid with random matrices (entries have range 0–10,000) using different values of S on the column distribution (see Table 5.1). We indicate timing for A-mode, B-mode, Backsolves and Forward- and Backsolves together ("Solve" heading). For this example, $S = 30$ saves 76% of the triangular solve cost compared to $S = 1$, or approximately 186 seconds, roughly 6 seconds above the linear optimal. Simultaneously, we incur about 17 seconds additional cost in B-mode, while saving about 93 seconds in the Backsolve. Assuming $\eta^* = 1$ ($\eta^* = 0$), in the first above-mentioned objective function, we save about 262 (respectively, 76) seconds. Based on this example, and other experience, we conclude that this is a successful practical technique for improving overall sparse linear algebra performance. The following example further bolsters this conclusion.

5.5.2 Order 2500 Example

Now, we turn to a timing example of an order 2500 sparse, random matrix. The matrix has a random diagonal, plus two-percent random fill of the off-diagonals; entries have a dynamic range of 0-10,000. Normally, data is averaged over random matrices for each grid shape (as noted), and over four repetitive runs for each random matrix. Partial row pivoting was used exclusively. Table 5.2. compiles timings for various grid shapes of row-scatter/column-scatter, and row-scatter/column-($S = 10$) distributions, for as few as six nodes and as many as 128. Memory limitations set the lower bound on the number of nodes.

This example demonstrates that speedups are possible for this reasonably small

Table 5.1. Order 13040 Band Matrix Performance

Distribution		(time in seconds)			
Row	Column	A-Mode	B-Mode	Back-Solve	Solve
Scatter	S=1	1.140×10^3	1.603×10^2	1.196×10^2	2.426×10^2
	S=10	1.148×10^3	1.696×10^2	3.294×10^1	6.912×10^1
	S=25	1.091×10^3	1.670×10^2	2.713×10^1	5.752×10^1
	S=30	1.095×10^3	1.769×10^2	2.653×10^1	5.631×10^1
	S=40	1.116×10^3	2.157×10^2	2.573×10^1	5.472×10^1
	S=50	1.127×10^3	2.157×10^2	2.764×10^1	5.743×10^1
	S=100	1.279×10^3	4.764×10^2	2.520×10^1	5.367×10^1
	Linear	2.247×10^3	1.161×10^3	2.333×10^1	4.993×10^1

The above timing data, for the 16x12 grid configuration with scattered rows, indicates the importance of the one-parameter distribution with $S > 1$ for balancing factorization cost *vs.* triangular-solve cost. The random matrices, of order 13040, have an upper bandwidth of 164 and a lower bandwidth of 162. "Best" performance occurs in the range $S \approx 25 \ldots 40$.

sparse example with this general-purpose solver, and that the one-parameter distribution is key to achieving overall better performance even for this random, essentially unstructured example. Without the one-parameter distribution, triangular solver performance is poor, except in grid configurations where the factorization is itself degraded (*e.g.*, 2x16). Furthermore, the choice of $S = 10$ is universally reasonable for the $Q > 1$ grid shapes illustrated here, so the distribution proves easy to tune for this type of matrix. We are able to maintain an almost constant speed for the triangular solves while increasing speed for both the A-mode and B-mode factorizations. We presume, based on experience, that triangular solve times are comparable to but somewhat worse than the sequential solution times – further study is needed in this area to see if and how performance can be improved. The consistent A-mode to B-mode ratio of approximately two is attributed primarily to reduced communi-

cation costs in B-mode, realized through the elimination of essentially all *combine* operations in B-mode.

While triangular-solve performance exemplifies sequentialism in the algorithm, it should be noted that we do achieve significant overall performance improvements between 6 (6x1) nodes and 96 (16x6 grid) nodes, and that the repeatedly used B-mode factorization remains dominant compared to the triangular solves even for 128 nodes. Consequently, efforts aimed further to increase performance of the B-mode factorization (at the expense of additional A-mode work) are interesting to consider. For the factorizations, we also expect that we are achieving non-trivial speedups relative to one node, but we are unable to quantify this at present because of the memory limitations alluded to above.

5.6 Future Work, Conclusions

There are several classes of future work to be considered. First, we need to take the A-mode "analyze" phase to its logical completion, by including pivot-order sorting of the L/U pointer structures to improve performance for systems that should demonstrate sub-quadratic sequential complexity. This will require minor modifications to B-mode (that already takes advantage of column-traversing elimination), to reduce testing for inactive rows as the elimination progresses. We already realize optimal computation work in the triangular solves, and we mitigate the effect of $Q > 1$ quadratic communication work using the one-parameter distribution.

Second, we need to exploit "timelike" concurrency in linear algebra – multiple pivots. This has been addressed by Alaghband for shared-memory implementations of *MA28* with $O(N)$-complexity heuristics [1,2]. These efforts must be reconsidered in the multicomputer setting and effective variations must be devised. This approach should prove an important source of additional speedup for many chemical engineering applications, because of the tendency towards extreme sparsity, with mainly band

and/or block-diagonal structure.

Third, we could exploit new communication strategies and data redistribution. Within a process grid, we could incrementally redistribute L/U by utilizing the inherent broadcasts of L columns and U rows to improve load balance in the triangular solves at the expense of slightly more factorization computational overhead and significantly more memory overhead (nearly a factor of two). Memory overhead could be reduced at the expense of further communication if explicit pivoting were used concommitantly.

Fourth, we can develop adaptive broadcast algorithms that track the known load imbalance in the B-mode factorization, and shift greater communication emphasis to nodes with less computational work remaining. For example, the pivot column is naturally a "hot spot" because the multiplier column (L column) must be computed before broadcast to the awaiting process columns. Allowing the non-pivot columns to handle the majority of the communication could be beneficial, even though this implies additional overall communication. Similarly, we might likewise apply this to the pivot row broadcast, and especially for the pivot process, because it must participate in two broadcast operations (see Figures 2.6, 2.7).

We could utilize two process grids. When rows (columns) of U (L) are broadcast, extra broadcasts to a secondary process grid could reasonably be included. The secondary process grid could work on redistribution L/U to an efficient process grid shape and size for triangular solves while the factorization continues on the primary grid. This overlapping of communication and computation could also be used to reduce or nearly mask the cost of transposing the solution vector from column-distributed to row-distributed, which normally follows the triangular solves.

Further, we may wish to consider the formulation of local elimination steps in a way suitable for nodal vector computation on future systems. This will primarily be of interest for very large sparse matrices, and is mainly a sequential optimization of

the nodāl code, despite its potential importance.

The sparse solver supports arbitrary user-defined pivoting strategies. We have considered but not fully explored issues of fill-reduction *vs.* minimum time; in particular we have implemented a Markowitz-count fill-reduction strategy [16]. Study of the usefulness of partial column pivoting and other strategies is also needed. We will report on this in the future.

Reduced-communication pivoting and parametric distributions can be applied immediately to concurrent dense solvers with definite improvements in performance. While triangular solves remain lower-order work in the dense case, and may sensibly admit less tuning in S, the reduction of pivot communication is certain to improve performance. A new dense solver exploiting these ideas is under construction at present. Some of the notions speculatively referred to above will also be considered in the dense case.

In closing, we suggest that the algorithms generating the sequences of sparse matrices must themselves be reconsidered in the concurrent setting. Changes that introduce multiple right-hand sides could help to amortize linear algebra cost over multiple timelike steps of the higher-level algorithm. Because of inevitable load imbalance, idle processor time is essentially free – algorithms that find ways to use this time by asking for more speculative (partial) solutions appear of merit toward higher performance.

Table 5.2. Order 2500 Matrix Performance

Shape	Column Distrib.	(time in seconds) A-Mode	B-Mode	Back-Solve	Solve	Avgs
6x1	Scatter	4.859×10^2	2.145×10^2	3.025×10^0	6.696×10^0	3
3x3	Scatter	3.567×10^2	1.783×10^2	1.997×10^1	4.115×10^1	1
3x4	Scatter	3.101×10^2	1.303×10^2	2.149×10^1	4.452×10^1	1
4x3	Scatter	2.778×10^2	1.526×10^2	1.728×10^1	3.537×10^1	1
2x16	Scatter	4.500×10^2	3.350×10^2	3.175×10^0	1.101×10^1	1
12x1	Scatter	2.636×10^2	1.206×10^2	4.0188×10^0	8.340×10^0	3
16x1	Scatter	2.085×10^2	1.000×10^2	4.856×10^0	9.8744×10^0	3
8x2	Scatter	2.013×10^2	9.41×10^1	1.127×10^1	2.295×10^1	3
	$S = 10$	1.997×10^2	9.63×10^1	4.508×10^0	9.399×10^0	3
4x4	Scatter	2.371×10^2	1.056×10^2	1.225×10^1	3.549×10^1	3
	$S = 10$	2.329×10^2	1.104×10^2	4.192×10^0	9.406×10^0	3
4x6	Scatter	1.456×10^2	7.72×10^1	1.723×10^1	3.528×10^1	3
	$S = 10$	1.684×10^2	8.85×10^1	4.206×10^0	9.303×10^0	3
12x2	Scatter	1.490×10^2	6.95×10^1	9.08×10^0	1.851×10^1	3
	$S = 10$	1.425×10^2	6.54×10^1	4.557×10^0	9.439×10^0	3
12x3	Scatter	1.0429×10^2	5.39×10^1	9.34×10^0	1.898×10^1	3
	$S = 10$	1.0382×10^2	5.42×10^1	4.539×10^0	9.390×10^0	3
8x8	Scatter	1.154×10^2	6.16×10^1	1.1082×10^1	2.2906×10^1	3
	$S = 10$	1.145×10^2	6.64×10^1	4.4600×10^0	9.651×10^0	3
12x6	Scatter	6.470×10^1	3.527×10^1	9.410×10^0	1.9141×10^1	3
	$S = 10$	6.265×10^1	3.417×10^1	4.555×10^0	9.495×10^0	3
16x6	Scatter	5.014×10^1	2.744×10^1	9.085×10^0	1.8327×10^1	3
	$S = 10$	4.984×10^1	2.905×10^1	5.2811×10^0	1.0740×10^1	3
16x8	Scatter	7.046×10^1	3.879×10^1	8.9535×10^0	1.8243×10^1	3
	$S = 10$	6.70×10^1	3.854×10^1	5.239×10^0	1.0816×10^1	3

Performance as a function of grid shape and size, and S-parameter; all cases have scattered rows. "Best" performance is for the 16x6 grid with $S = 10$.

Chapter 6

Concurrent DASSL

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 chapter, 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.

6.1 Introduction

In this chapter, 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 [48]. 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 [38,9], and based on a loosely synchronous model of communicating sequential processes [27]. *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, for example, amortizing the cost of linear algebra over more candidate solutions in the algorithm.

6.2 Mathematical Formulation

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

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

$$\mathbf{F}(\mathbf{Z}, \dot{\mathbf{Z}}, \mathbf{u}; t) = \mathbf{0}, \quad t \in [T_0, T_1], \tag{6.1}$$

$$\mathbf{Z}(t = T_0) \equiv \mathbf{Z}_0, \quad \dot{\mathbf{Z}}(t = T_0) \equiv \dot{\mathbf{Z}}_0, \tag{6.2}$$

with unknown state vector $\mathbf{Z}(t) \in \Re^N$, known external inputs $\mathbf{u}(t) \in \Re^U$, where $\mathbf{F}(\bullet; t) \mapsto \Re^N$ and $\mathbf{Z}_0, \dot{\mathbf{Z}}_0 \in \Re^N$ are the given initial-value and derivative vectors, respectively. We will refer to Equation 6.1'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) [9, Chapter 5], where the index is the minimum number of times that part or all of Equation 6.1 must be differentiated with respect to t in order to express $\dot{\mathbf{Z}}$ as a continuous function of $\mathbf{Z}$ and t [9, page 17].

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

$$\mathbf{F}_{\mathcal{D}}(\mathbf{Z}_i; \tau_i) \equiv \mathbf{F}(\mathbf{Z}_i, \mathcal{D}_i\mathbf{Z}_i, \mathbf{u}_i; t = \tau_i) = \mathbf{0}, \tag{6.3}$$

a set of (in general) nonlinear *staticized* equations. A sequence of Equation 6.3'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*,

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

the variable step-size integration algorithm picks the τ_i's as the integration progresses, based in part 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 6.3, is via the familiar Newton-Raphson iterative method (yielding $\mathbf{Z}_i \equiv \mathbf{Z}_i^{\infty}$):

$$\begin{aligned} \mathbf{Z}_i^{k+1} &= \mathbf{Z}_i^k - c\{\nabla_{\mathbf{Z}}\mathbf{F}_{\mathcal{D}}(\mathbf{Z}_i^{m_k};\tau_i)\}^{-1}\mathbf{F}_{\mathcal{D}}(\mathbf{Z}_i^k;\tau_i), \\ & \qquad k = 0, 1, \ldots \end{aligned} \tag{6.4}$$

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.* [9, 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

$$\nabla_{\mathbf{Z}}\mathbf{F}_{\alpha,\beta}(\mathbf{Z};\tau_i) \equiv \nabla_{\mathbf{Z}}\mathbf{F}(\mathbf{Z}, \alpha\mathbf{Z}+\beta, \mathbf{u}; t) = \frac{\partial \mathbf{F}}{\partial \mathbf{Z}} + \alpha \left.\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{Z}}}\right|_{\dot{\mathbf{Z}}=\alpha\mathbf{Z}+\beta}, \tag{6.5}$$

where, 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:

$$\mathbf{F}_{\alpha,\beta}(\mathbf{Z}+\delta_j;\tau_i) - \mathbf{F}_{\alpha,\beta}(\mathbf{Z};\tau_i) = \left\{\nabla_{\mathbf{Z}}\mathbf{F}_{\alpha,\beta}(\mathbf{Z};\tau_i)\right\}\delta_j + O\left(\|\delta_j\|_2^2\right) \tag{6.6}$$

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 6.6 yields a first-order-accurate approximation in d_j of the jth column of the Jacobian matrix:

$$\frac{\mathbf{F}_{\alpha,\beta}(\mathbf{Z}+\delta_j;\tau_i) - \mathbf{F}_{\alpha,\beta}(\mathbf{Z};\tau_i)}{d_j} = \left\{\nabla_{\mathbf{Z}}\mathbf{F}_{\alpha,\beta}(\mathbf{Z};\tau_i)\right\}\mathbf{e}_j + O(d_j), \quad j = 1,\ldots,N \tag{6.7}$$

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 [9,16]. 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 chapter 5, although future versions of *Concurrent DASSL* will support the iterative linear algebra approaches by Ashby, Lee, Brown, Hindmarsh *et al.* [5,10]. For the sparse 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 [9]. The useful lifetime of a single Jacobian evidently depends somewhat strongly on details of the integration procedure [9].

6.3 *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 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 (chapter 7). This is akin to the *ASCEND II* methodology utilized by Kuru and many others [29]. 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 run-time 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 new kinds of templates requires, however, 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 [61,14], 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* [4,38,15]). 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.

6.3.1 Template Structure

A template is a prototype for a sequence of DAE's which can be used repeatedly in multiple, distinct 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 can 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 $\mathbf{Z}$, 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 the concurrent formulation section 6.4.

6.3.2 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 [16]. 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 tridiagonal Jacobian structure with off-block-tridiagonal 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 $P \times Q = R$. The partitioning of the equations forms, in main part, the so-called "concurrent database." This grid structure is illustrated in 4.1. 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 that 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

[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.

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 it 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. See appendix C.

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 chapter 5, 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 chapter 5. The effects of row and column degree-of-scattering are described in chapter 5 with attention to linear algebra performance.

6.4 Concurrent Formulation

6.4.1 Overview

Next, we turn to Equation 6.1'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 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 6.1.

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 their respective databases are mailed to them. 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 6.3 – 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.

6.4.2 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 [9, Chapter 5] and outlined in [38]. *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 $P \times Q = R$ 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.*, [58]), 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 (chapter 3). 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 synchronization 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*) see Stone [51], and chapters 2, 3. 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 result repetitively, or a single Q-striding norm, that takes advantage of the column-wise 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. (The granularity design equation defined in chapter 2 is useful in making judgements in this connection.)

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 [16,29], which is directly a function of the sparsity structure of the model equations, Equation 6.1. 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 $T_{Jac,P\times Q=R}$ the time for this step, $S_{Jac,P\times Q=R}$ 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:

$$T_{Jac,P\times Q=R} \approx \lceil N/Q \rceil \times T_{res,P}, \tag{6.8}$$

$$S_{Jac,P\times Q=R} \approx \frac{N}{\lceil N/Q \rceil} \times S_{res,P}, \tag{6.9}$$

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 chapter 5). The first is called "bandlike" Jacobian evaluation. For a banded Jacobian matrix of bandwidth β, only β 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 (chapter 5). 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 [16, 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 process column-by-process 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 in chapter 5. 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 [2], 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 chapter 5. 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 chapter 5. 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 chapter 5 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.

6.5 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 (See chapters 3, 5).

In this simulation, we consider seven distillation columns arranged in a tree-sequence, 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 6.1. 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 tridiagonal 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 tridiagonal 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 "$1 + \epsilon$" 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 chapter 5.

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

Table 6.1. 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 a seven-column simulation of eight alcohols, with a total of 1,001 trays. See chapter 5 for more on data distributions.

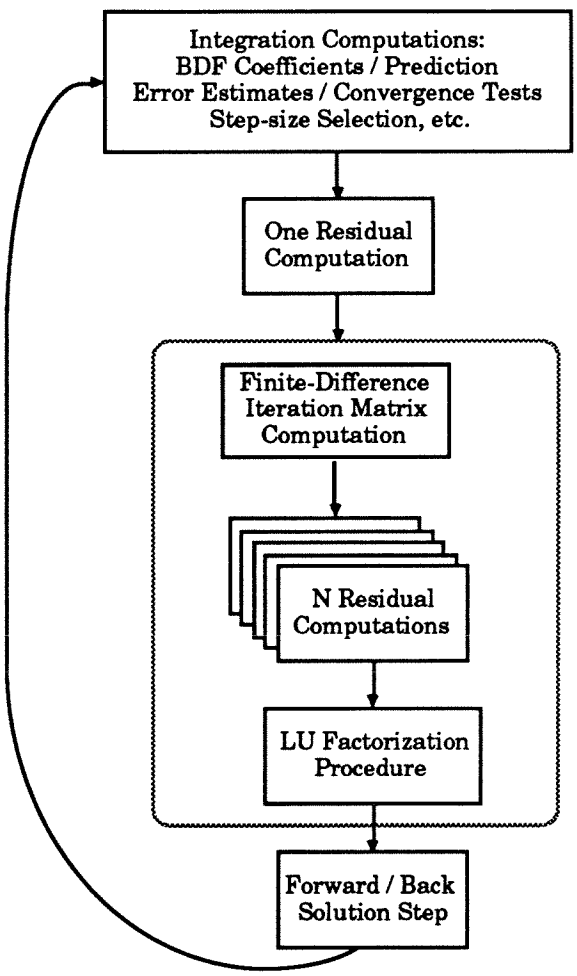
for overall speedup compared to the sequential case. Mainly, we plan to consider the Waveform-Relaxation approach more heavily, and also to consider new classes of dynamic distillation simulations with *Concurrent DASSL*.

6.6 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 timelike 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 (see chapter 7).

Figure 6.1. 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.

Chapter 7

Waveform Relaxation for Distillation Simulation

Abstract

Our goal of *orders-of-magnitude* speedup strongly motivates the use of novel algorithmic approaches for large-scale simulation. As discussed in chapter 2, the sequential fraction, communication overheads, and load-imbalance effects may drastically limit the performance potential of parallelized sequential algorithms. In this chapter, we indicate the formulation of a specific form of dynamic distillation simulation – binary distillation with constant relative volatility thermodynamics – within the Waveform Relaxation (Picard-Lindelöf) paradigm, a method whose modern origin is in electrical engineering circuit simulation [30,62,39,36,46,32,17,37]. The key points of this chapter are as follows: first, that it's possible to formulate simple DAE-based chemical engineering models within an existing Waveform Relaxation software framework, second, there is evidence to suggest that the method should be effective at least for stiff ODE systems, based on electrical-engineering experience, and, third, that there is reasonable hope to apply Waveform Relaxation to more general low-index DAE's arising in chemical engineering, though still much study and practical experience are clearly required to achieve this goal.

Initially, we are developing a simplified binary distillation simulation using Waveform Relaxation. This numerical technique has proven successful for the concurrent simulation of very large-scale integrated (VLSI) circuits [30,62,39,32,17,37] and is

therefore a potentially promising approach. Rather than an end in itself, however, we expect that results of this research effort will prove relevant to more general concurrent dynamic simulation including rigorous multicomponent distillation and chemical process flowsheeting. We describe the implementation effort (which generalizes the pre-existing *CONCISE* VLSI circuit simulator, [32,37]), the simplified distillation model, design issues and current status including a sketch of the underlying algorithm. Appendix E discusses existing convergence theory for Waveform Relaxation, and looks at distillation models in more detail within the context of proving convergence. A motivating example appeared in chapter 2.

7.1 Introduction

Cost-effective, high-speed computing is essential in many aspects of chemical-engineering practice, notably for the simulation of large-scale dynamic systems. The arrival of powerful, highly concurrent message-passing multicomputers potentially offers such economical large-scale computing capability [6,20]. Development of appropriate, efficient algorithms which realize this potential must therefore become an important area of ongoing research and development in chemical engineering. We seek algorithms suitable for use with *many* processes (that is, one hundred to one thousand or more) rather than a handful. Only such algorithms can deliver the *orders-of-magnitude* speedup needed to expedite large-scale chemical engineering computations. Finally, one should note that the approach followed here does not exclude the possible advantages of vectorization on a process-by-process basis; this additional source of speedup remains a higher-order effect peculiar to individual processes that need not be considered initially (see, however, [59]).

To motivate our choice of application, we note that the distillation column and its variants are arguably the most important class of unit operations in chemical plants; we have consequently chosen dynamic distillation simulation as a primary target for

concurrent simulation. Notably, dynamic models most often consist of differential-algebraic systems with stiff, nonlinear ordinary differential equations modeling fluid flow, mass and energy balances, as well as nonlinear algebraic equations modeling the vapor-liquid equilibrium. In simplified models, fluid flow dynamics are neglected, increasing the number of algebraic equations. A multicomponent system with a practical number of columns involves thousands of equations, as we have seen in chapter 6. Furthermore, accurate physical property calculations for vapor-liquid equilibrium can be outstandingly arduous. In short, detailed dynamic models normally have huge computing requirements [28]. Presently, we work with a highly simplified binary distillation model with simple vapor-liquid equilibrium (constant relative volatility). A set of stiff, coupled nonlinear ordinary differential equations result if we exclude network feedbacks; a differential-algebraic system results otherwise. Importantly, even the simpler of these two models still captures three of the central simulation issues: problem stiffness, large scale, as well as complications resulting from coupling between columns in multiple-column simulations.

It is natural to seek highly concurrent algorithms for distillation simulation in order to reduce the time needed to obtain a simulation. However, to make effective use of many communicating sequential processes in a multicomputer, it remains essential (for a fixed problem or problem *size*) to avoid sequential bottlenecks (as indicated by Amdahl's law; [19,3,32] and chapter 2) as well as gross imbalance of the computational load among the processes. From the outset, we want to re-emphasize the importance of algorithms which can work effectively with one hundred to one thousand processes or more in order to yield *orders-of-magnitude* speedup. Finding such solution procedures is much more difficult than that of developing algorithms for four or eight-headed machines wherein a significantly large sequential fraction of computation remains permissible. These factors strongly motivate the investigation of new computational algorithms for the simulation of large-scale systems. In the last

chapter, we obtained limited speedup for our large-scale distillation problem using a carefully parallelized sequential method, though this work is amenable to further performance improvements.

In this connection, we have embraced the *Waveform Relaxation* methodology, a subject of intensive research in the area of concurrent VLSI circuit simulation where it has proven effective [32,37]. Efficient implementations of Waveform Relaxation have minor inherent sequentialism and fall, consequently, in the loose category of "efficient concurrent algorithms" – see chapter 2. Temporal latency of subsystems can, furthermore, be exploited naturally by multirate integration [46,32] incorporated therein, implying reduced computational effort for latent subsystems and, correspondingly, the opportunity for greater overall speedup in the total simulation. Because of the particular success of Mattisson's *CONCISE* simulator for VLSI circuit simulation via Waveform Relaxation [32,37], we have formulated the binary distillation model within this program. By building upon and generalizing an extant system, we also avoid the redundancy of effort inherent in creating an entirely new concurrent simulation program. The important and difficult questions of dynamic load balancing are reserved for future consideration.

7.2 'Idea' of Waveform Relaxation

Recall the initial-value problem **IVP** defined in the previous chapter, Equation 6.1. For the computation of $\mathbf{Z}_i$, we could, alternatively, formulate the inner iteration as independent, scalar, Newton-Raphson iterations, given an initial approximation $\mathbf{Z}^0$ (*e.g.*, $\mathbf{Z}^0 \equiv \mathbf{Z}_{i-1}$) where $\mathbf{Z} \equiv (Z_1, Z_2, \ldots, Z_N)^T$:

$$Z_j^{k+1} = Z_j^k - \left(\frac{\partial F_{\mathcal{D}j}(Z_j^{m_k}, Z_{l\neq j}^0; \tau_i)}{\partial Z_j} \right)^{-1} \times F_{\mathcal{D}j}(Z_j^k, z_{l\neq j}^0; \tau_i), \qquad (7.1)$$
$$j = 1, \ldots, N, \quad k = 0, 1, \ldots$$

yielding $f Z_j^\infty$, $j = 1, \ldots, N$. In each of these N independent scalar iteration processes, all $Z^0_{l \neq j, (l=1,\ldots,N)}$ remain at the constant values provided initially. After the jth scalar iteration process converges, it broadcasts its new value Z_j^∞ to the other $\leq N-1$ processes that need it, and which reserve it for future use. When all N processes have converged and the data broadcasts are also completed, each process has available a new approximation

$$\mathbf{Z}^0 \leftarrow \mathbf{Z}^\infty \equiv (Z_1^\infty, Z_2^\infty, \ldots, Z_N^\infty)^T. \tag{7.2}$$

The inner iterations are repeated as necessary (restarting with $k = 0$) until a global stopping criterion is satisfied. When (if) convergence is achieved, $\mathbf{Z}_i \leftarrow \mathbf{Z}^\infty$. In short, assuming convergence with reasonable speed (topics beyond our current scope), we have uncovered useful concurrency with this iterative approach while each iterative process is reduced to a straightforward scalar procedure [32].

In a Waveform Relaxation scheme, we apply this same sort of idea to *waveforms* of values (elements of a function space defined on subintervals of $[T_0, T_1]$) in each process instead of to single real values as just illustrated. It is worthwhile to note that many types of Waveform Relaxation algorithms are possible; for example, the approach developed in [39] differs markedly from that used in *CONCISE* and which which we outline next.

In the following, we define waveforms by underlined symbols such as $\underline{Z}_j$, which specifically connotes the entire function $Z_j(t)$ for $t \in [t_a, t_b] \subseteq [T_0, T_1]$, a subinterval of interest; $\underline{\mathbf{Z}}$ is defined as the vector of waveforms $(\underline{Z}_1, \underline{Z}_2, \ldots, \underline{Z}_N)^T$. Numerically, waveforms are stored as discrete, ordered sets of time-value pairs; this data is interpolated to provide waveform values at any particular time in the interval $[t_a, t_b]$.

The task of the jth inner iteration is the construction of $\underline{Z}_j^K$ given $\underline{\mathbf{Z}}^{K-1}$ by solving the following scalar differential or algebraic equation on $[t_a, t_b]$:

$$F_j(\ (\underline{Z}_1^{K-1}, \ldots, \underline{Z}_{j-1}^{K-1}, \underline{Z}_j^K, \underline{Z}_{j+1}^{K-1}, \ldots, \underline{Z}_N^{K-1})^T,$$

$$(\dot{\underline{Z}}_1^{K-1}, \ldots, \dot{\underline{Z}}_{j-1}^{K-1}, \dot{\underline{Z}}_j^{K}, \dot{\underline{Z}}_{j+1}^{K-1}, \ldots, \dot{\underline{Z}}_N^{K-1})^T, \underline{\mathbf{u}}; t) = 0 \qquad (7.3)$$

yielding $\underline{Z}_j^K$, where F_j is defined to be the jth component of the original (undiscretized) differential-algebraic system, Equation 6.1. It should be emphatically clear from the notation that the waveforms not local to process j are held fixed: only the jth waveform is updated in process j. Once the jth process completes its waveform, it broadcasts this waveform to other processes that need it, according to the structure of the DAE's. Furthermore, upon completion of all N waveform integrations as well as their transmissions, each process has the part of $\underline{\mathbf{Z}}^K$ available that it needs, and can repeat the inner iteration process (as necessary) to obtain $\underline{\mathbf{Z}}^{K+1}$ and so forth, until a stopping criterion is satisfied.

The specific details of the integration procedure are not unimportant to the waveform iteration: it is of central importance both for VLSI circuit and distillation simulation applications that the integrator work effectively for stiff systems (including systems with purely algebraic equations). In particular, *CONCISE* employs a backward-differentiation formula approach which implements a predictor-corrector algorithm for each point in the integration; for details see [45,32,8,9]. This variable step-order integration algorithm attempts to minimize the number of steps required to perform the waveform integration consistent with error control based on a measure of the local truncation error. In this way, latent processes compute waveforms with less time-value points than more active processes. Correspondingly, there is a reduction in the amount of computational effort associated with latent states. Furthermore, the local truncation error tolerances are reduced with increasing K-iteration to 'encourage' more accuracy in later iterations while allowing early iterations to be less accurate.

The foregoing discussion captures the very basic idea of Waveform Relaxation, while ignoring many of the practically important issues dealt with in *CONCISE* (in

order to enhance convergence/performance) including:

- *Efficient scheduler with small sequential fraction.*
- *Works over subintervals of waveforms.*
- *Dynamic waveform splitting strategy.*
- *'Breakpoint' Strategy following input discontinuities.*
- *Ability to cluster tightly coupled states.*
- *Optional direct solution of clustered states.*
- *Provision for several outer iteration schemes.*

It is beyond the scope of this discussion to describe *CONCISE* in complete detail although further structural details are mentioned below in conjunction with distillation simulation. We refer you to the theses by Mattisson and Peterson which provides more elaborate discussions [32,37].

At this juncture, we turn to a discussion of the binary distillation model.

7.3 The Binary Distillation Model

The following assumptions are made with respect to the binary distillation model. It is worthwhile to reiterate that the model chosen here is not intended to reflect state-of-the-art distillation simulation. As discussed above, we chose this simplified model to illustrate important issues in concurrent distillation simulation that will carry over to more complicated and correspondingly realistic models. Here, we follow the standard simplified modelling assumptions utilized in [50]:

- *Constant molar flows:* L, V.
- *No vapor holdup on trays.*
- *Immediate vapor response.*

- *Constant liquid holdup on trays:* M.
- *Immediate liquid response.*
- *Negligible flow dynamics between trays.*
- *Vapor-liquid equilibrium attained on each tray.*
- *Perfect mixing attained on each tray.*
- *Perfect level control in condensers and reboilers.*
- *Isobaric operation.*
- *Constant relative volatility vapor-liquid equilibrium.*

A distillation column is constructed of vertically connected trays (stages) upon which mass transfer (physical separation) is accomplished between a contacted vapor and liquid phase; countercurrent vapor and liquid flow streams link column trays. Side feed (draw) streams add (remove) liquid and/or vapor to (from) column trays providing bypasses and feedbacks as well as linking multiple columns. Figure 7.1. presents an abstract model of the *TRAY template*, the format in which the tray equations outlined below are realized in *CONCISE*; Figure 7.2. illustrates a single-feed column utilizing TRAY templates set up in various configurations. These differential-algebraic equations have six states:

$x[i]$ *Light component liquid phase composition.*

$y[i]$ *Light component vapor phase composition.*

$L[i]$ *Liquid flow rate out bottom of tray.*

$V[i]$ *Vapor flow rate out top of tray.*

$M[i]$ *The tray's constant liquid holdup.*

$zd[i]$ *Flashed draw composition variable.*

In what follows, the constant $\bar{\alpha}$ is the relative volatility parameter; Table 7.1. presents a summary of nomenclature for these equations as well as for Figures 7.1., 7.2

Table 7.1. Nomenclature for the TRAY Template
(comments appropriate for Model #3)

	Holdup	
M[i]	Ext. Input	Holdup for TRAY i
M[i]	Non-State	Output of Holdup
	Liquid Input from TRAY i+1	
L[i+1]	State	Flow rate
x[i+1]	State	Light component composition
	Vapor Input from TRAY i-1	
V[i+1]	State	Flow rate
y[i+1]	Non-State	Light component composition
	Normal Vapor Feed Inputs	
Vf[i]	—	Flow rate
yf[i]	—	Light component composition
	Normal Vapor Feed Inputs	
Lf[i]	—	Flow rate
xf[i]	—	Light component composition
	Flashed Feed	
F[i]	—	Flow rate
zf[i]	—	Light liquid composition
qf[i]	—	Quality of the feed
	Normal Draw (External Inputs)	
Vd[i]	—	Vapor Flow rate
Ld[i]	—	Liquid Flow rate
	Normal Draw (Outputs)	
Vd[i]	—	Vapor Flow rate
y[i]	—	Light component vapor composition
Ld[i]	—	Liquid Flow rate
x[i]	—	Light component liquid composition
	Flashed Draw (External Inputs)	
Fd[i]	—	Flow rate
qd[i]	—	Quality
	Flashed Draw (Outputs)	
Fd[i]	—	Flow rate
qd[i]	—	Quality
zd[i]	State	Composition
	Vapor Output to TRAY i+1	
V[i]	State	Flow rate
y[i]	Non-State	Light component composition
	Liquid Output to TRAY i-1	
L[i]	State	Flow rate
x[i]	State	Light component composition

(for brevity, we omit various technical details concerning flash mode operation of the side streams).

Mole (or mass) balance of light component:

$$\begin{aligned} M[i]\dot{x}[i] &= (L[i+1]\,x[i+1]) + (V[i-1]\,y[i-1]) \\ &\quad - ((L[i]\,x[i]) + (V[i]\,y[i])) \end{aligned} \tag{7.4}$$

Vapor-liquid equilibrium (constant relative volatility formulation):

$$y[i] = \frac{\bar{\alpha}\,x[i]}{1 + (\bar{\alpha} - 1)x[i]} \tag{7.5}$$

Immediate liquid, vapor response, respectively:

$$L[i] = L[i+1] + Lf[i] - Ld[i] \tag{7.6}$$

$$V[i] = V[i-1] + Vf[i] - Vd[i] \tag{7.7}$$

Constant liquid holdup:

$$M[i] \triangleq M[i] \tag{7.8}$$

Flashed draw composition variable:

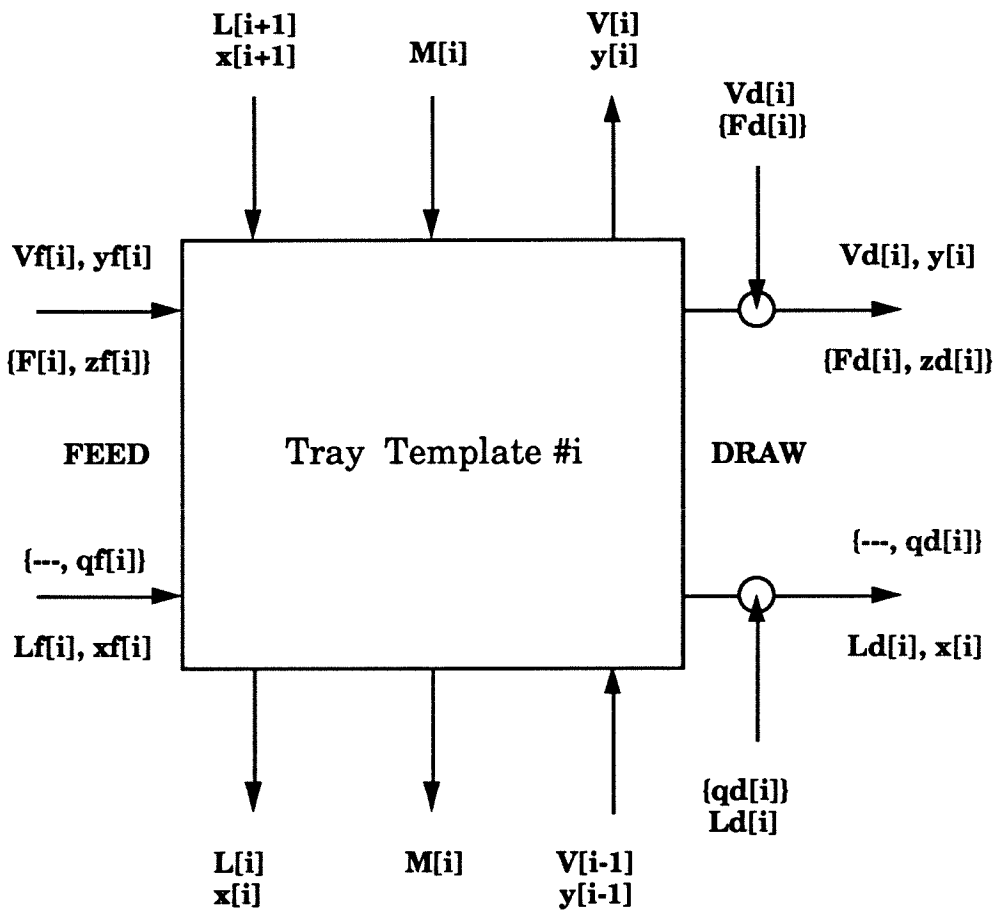
$$zd[i] = qd[i]\,x[i] + (1 - qd[i])y[i] \tag{7.9}$$

By the notation "$M[i] \triangleq M[i]$," we mean that the holdup variable $M[i]$ assumes whatever value is externally set for it.

7.4 The TRAY Template

CONCISE implements circuit simulation via an *extended nodal formulation* where

Figure 7.1. The *CONCISE* TRAY Template



each ungrounded nexus of circuit device terminals establishes a node [32]. For circuit simulation, the equation $\mathbf{F}(\mathbf{Z}, \dot{\mathbf{Z}}, \mathbf{u}; t) = \mathbf{0}$ implies the usual statement of Kirchoff's current law (KCL) for an electrical network; namely, *The algebraic sum of currents leaving any node is zero.* In that context, the state variables (nodes) $\mathbf{Z}$ represent voltages; it is therefore voltage waveforms which are iterated in *CONCISE* when applied to circuit simulation.

The nonlinear mathematical model associated with any particular circuit element (*e.g.*, capacitor, MOS transistor) is represented in *CONCISE* as a *device.* A device is realized in the simulator with a series of functions and subroutines that implement the mathematical model and also manage associated information/database requirements. At present, we restrict our attention solely to issues involving the mathematical model. As stated in the previous section, the equations of binary distillation are housed as the "seventeen-terminal" *TRAY template* depicted in Figure 7.1. A distillation tray is simulated by connecting a TRAY template to six nodes, which correspond directly to the states L, V, M, x, y, zd. Furthermore, in order to establish, for example, a vapor flow stream from tray $i-1$ to tray i of the same column, two designated terminals of TRAY template i associated with that vapor inflow are connected to states V, y of tray $i-1$. Other terminals of a TRAY template may be connected to ground (implying a zero value) or to external inputs. In this way, a general set of coupled, binary distillation columns can be realized using only the single TRAY template type. For example, Figure 7.2. captures the six different instantiations of the TRAY template needed to create the single-feed column illustrated there.

CONCISE evaluates a circuit on a node-by-node rather than a device-by-device basis. Consequently, the evaluation of a single node is an important primitive function. Each node maintains a list of device terminals connected to it; associated with those terminals are functions coded to return the derivative ($\dot{Z}$) and state (Z) contribution associated with the connection; there are, as expected, six such functions

associated with the TRAY template. As a result of the nodal formulation, a device routine requires only the state values of the node under evaluation as well as the states of the other nodes connected to the device (neighbor nodes) in order to arrive at its contributions to Z and $\dot{Z}$. Although each state of a distillation tray typically has three terminals tied to it in view of the nearest-neighbor connectivity of a column (ignoring side streams), one TRAY template provides the entire state and derivative contribution. The other connections are, in effect, *read-only.* Hence, the tray equations remain in their natural format (modulo a conventional minus sign for derivatives). Therefore, while the TRAY template conveniently houses the structured, nonlinear differential-algebraic equations, the modeling equations need not be contorted to fit within the *CONCISE* framework.

For the purpose of solving the distillation trays, the six states of a single tray will almost certainly be clustered, because they are tightly coupled. As such, they will be solved together using a stiff DAE solution approach (in turn using a direct method for the linear systems arising in its Newton iterations) as a single process. We pose several possible numerical structures for these six states.

Model #1 The clustered states fall into two structural categories: *Inputs:* L, V, M – if no feedbacks are envisaged, then the flows need not be integration states. *Differential-Algebraic States:* x and y and zd; *Non-States:* none.

Model #2 The clustered states fall into two structural categories: *Inputs:* M is non-dynamic and depends solely on external inputs and/or states of other trays. *Differential-Algebraic States:* L, V, x , y and zd; *Non-States:* none.

Model #3 The clustered states fall into three structural categories: *Inputs:* M is non-dynamic and depends solely on external inputs and/or states of other trays. *Differential-Algebraic States:* L, V, x and zd; *Non-States:* y. We "tear" the vapor

composition variable in this model (see also chapter 6).

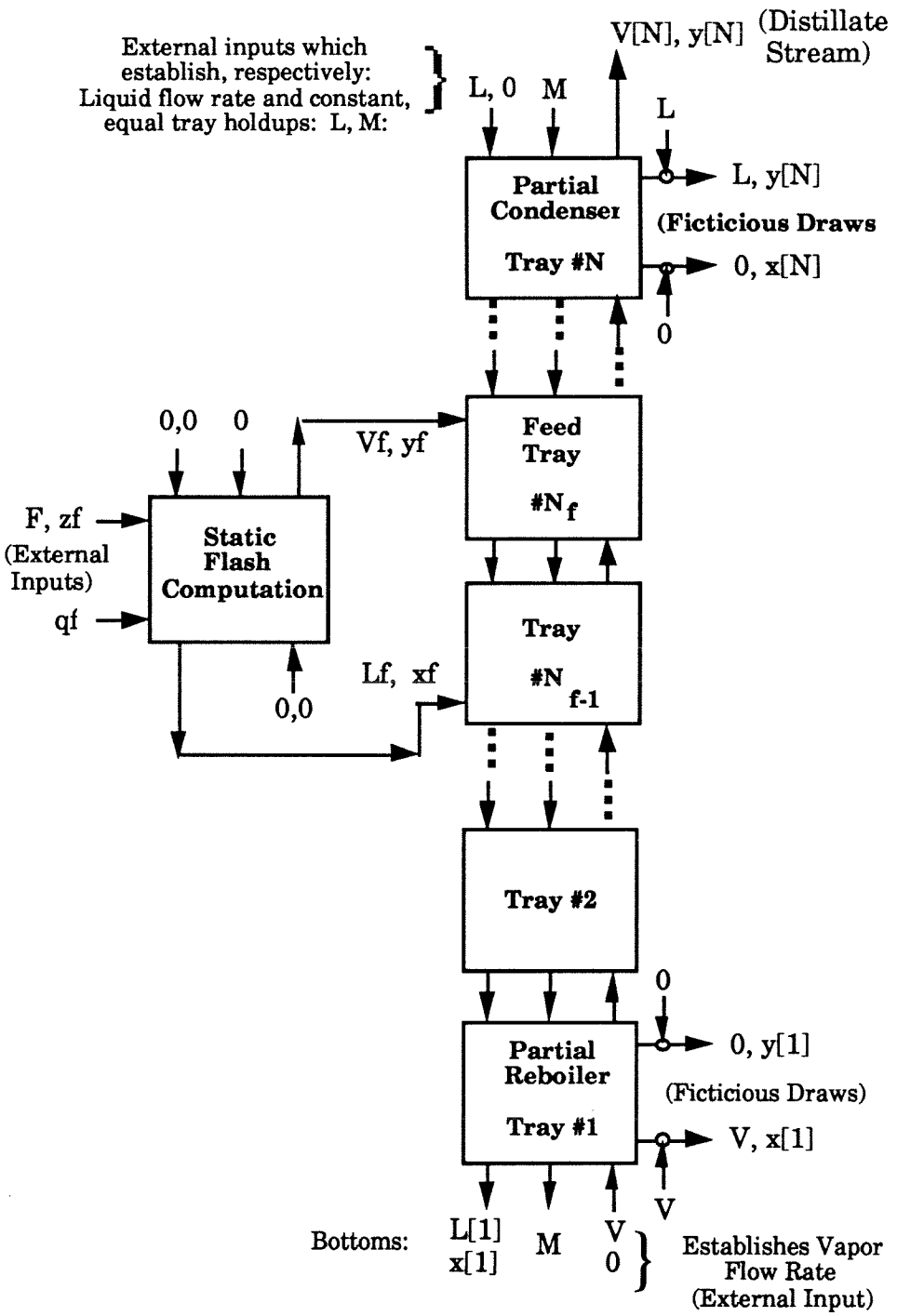
Model #4 The clustered variables fall into three structural categories: *Inputs:* M, L and V – no feedbacks assumed; *Differential-Algebraic States:* L, V, x and zd; *Non-state:* y. We "tear" the vapor composition variable in this model (see also chapter 6).

Though the backward-differentiation-formula approach to integration can accomodate algebraic equations per se, there remains a fundamentally different issue associated with the *Inputs*. They admit *no correction* in the integration algorithm since their waveforms are based entirely on node waveforms determined outside the cluster. For each time point computed in a waveform update, the differential-algebraic states (see various models) are handled together and their implicit integration governs at which time points the *Inputs* need evaluation. Thus, although the *Inputs* can be evaluated before the integration of $\mathbf{x}$ and $\mathbf{y}$ for any appropriate t in the waveform interval, the meaningful time points for such evaluations are not available a priori. In order to address these differential-algebraic issues properly within *CONCISE*, we introduced general three-phase integration logic that supports the three categories of variables just outlined.

7.5 Motivating Convergence

Issues of consistency, stability, convergence and performance are obviously central to the success of Waveform Relaxation for the dynamic simulation of distillation columns; these concepts are considered elsewhere [30,36,46] in general and for the VLSI-circuit context. Convergence is also movitated in [32,37]. Consistency implies that, as a supposed global stepsize $h \rightarrow 0$, the numerical formula recovers the original, undiscretized differential equations; performance implies that, in addition to convergence of the numerical scheme that the speed of convergence be acceptable.

Figure 7.2. A Single Feed Column in the *CONCISE* paradigm



At present, we merely point out that the same motivation offered in [32] for MOS circuits also applies at least to binary distillation Model #4 discussed here; namely, a sufficient condition for convergence is that each node be connected to a grounded capacitance (see also [30]). In our present context, this requirement only concerns the dynamic states – for Model #4 it is sufficient to consider the light liquid component x state; *Inputs* and *Non-States* as described above are not subject to integration as such.

So, we consider a version of Model #4 mentioned above wherein L, V, M are the non-dynamic input variables, x is dynamic, and y is a non-state, and we ignore zd. Define the sole differential equation describing x for the jth tray (using unemboldened, subscripted scalars instead of variables with bracketed indices henceforth for compactness of notation) by:

$$\begin{aligned} F_j \quad \equiv \quad & -M_j\dot{x}_j + L_{j+1}x_{j+1} + V_{j-1}y_{j-1} + Lf_jxf_j \\ & + Vf_jyf_j - (L_{j+1} + Lf_j)x_j - (V_{j-1} + Vf_j)y_j \equiv 0. \end{aligned} \tag{7.10}$$

Discretizing the time derivative using a backward differentiation formula yields an approximation of the form:

$$\dot{x}_j(t = \tau_i) \;\approx\; \frac{a_0}{h}x_j(\tau_i) - \mathcal{B}_j(\tau_{i-1}, \ldots, \tau_{i-k+1}), \tag{7.11}$$

where $\mathcal{B}_j(\ldots)$ is the part of the backward-differentiation expression independent of $x_j(\tau_i)$ while a_0/h is the "derivative operator" at $t = \tau_i$. For the backward Euler method, this approximation is simply:

$$\dot{x}_j(t = \tau_i) \;\approx\; \frac{x_j(\tau_i) - x_j(\tau_{i-1})}{\tau_i - \tau_{i-1}}. \tag{7.12}$$

Inserting the more general approximation, the **IVP** model and forming the relevant

(tridiagonal band) Jacobian matrix entries yields:

$$\frac{\partial F_{\mathcal{D}_j}}{\partial x_j} = -\frac{a_0}{h}M_j - (L_{j+1} + Lf_j) - (V_{j-1} + Vf_j) \times \frac{\partial y_j}{\partial x_j}, \tag{7.13}$$

$$\frac{\partial F_{\mathcal{D}_j}}{\partial x_{j-1}} = V_{j-1}\frac{\partial y_{j-1}}{\partial x_{j-1}}, \tag{7.14}$$

$$\frac{\partial F_{\mathcal{D}_j}}{\partial x_{j+1}} = L_{j+1}, \tag{7.15}$$

and

$$\frac{\partial y_j}{\partial x_j} = \frac{\bar{\alpha}}{(1 + (\bar{\alpha} - 1)x_j)^2}, \tag{7.16}$$

where only the diagonal entry of the jth row ($\forall j$) depends on h.

For h sufficiently small, a_0/h becomes large enough so that the Jacobian is diagonally dominant in view of the constant, non-zero holdups $M_j \, \forall j$, a quantity analogous to a grounded capacitance [30,32]. Off tridiagonal-band elements cannot depend on a_0/h explicitly because states do not directly couple to derivatives of other states (*cf*, nodal formulation *vs.* modified nodal formulation [32]). Hence, the same sufficient condition that leads to Waveform Relaxation convergence in [30,32] applies equally well to the binary distillation model. However, to be ultimately satisfactory, an in-depth analysis (not offered here) is clearly in order and we discuss this further in appendix E.

7.6 Summary, Discussion, Conclusions

The need for cost-effective high speed computing is essential in many areas of chemical engineering; notably, the simulation of large-scale dynamic systems is routinely computationally demanding. The advent of powerful, highly concurrent message-passing multicomputers offers real potential for such economical supercomputing provided

that concurrent algorithms can be developed that avoid both gross load imbalance and sequential bottlenecks. Noting that the problem of dynamic distillation simulation is both a demanding computational problem class and also of real importance in chemical engineering, we undertook the implementation of concurrent simulation for simplified, binary distillation. Even this modest framework retains the central aspects of stiffness, inter-column coupling as well as large-scale and is therefore a good starting point. Our target machines are general, multiple-instruction, multiple data multicomputers with asynchronous, point-to-point communication routines (*e.g.*, Intel iPSC/2, Symult s2010). We emphasize that the ultimate goal of this research is to develop efficient concurrent simulation techniques for relevant, demanding chemical engineering systems, rather than singularly for (binary) distillation.

In order to develop a simulator with a small sequential fraction and therefore the potential for thousand-process concurrency and *orders-of-magnitude* speedup, we followed the lead of VLSI-circuit simulation researchers where the *Waveform Relaxation* paradigm has proven successful [30,62,39,36,46,32,17,37]. In the foregoing, we describe the basic ideas behind this methodology and we also indicate several of the important heuristic aspects of the *CONCISE* VLSI-circuit simulator designed to improve convergence/performance; *CONCISE* is our starting point for concurrent dynamic simulation with Waveform Relaxation. The use of an existing, mature simulation system allows us to avoid the major replication of coding effort inherent in a fresh programming start.

We present the assumptions behind the binary distillation model and indicate how the nonlinear differential-algebraic equations are housed within *CONCISE*, in this instance within a *TRAY Template*, a seventeen-terminal circuit device analogue. We illustrate (with a single-feed column example) how the TRAY template covers the panoply of tray configuration requirements for the formulation of a distillation column network. This is important, because the implementation of multiple devices

requires significant one-time (formulation) effort as well as modest additional runtime overhead. Furthermore, we describe the important issues involved in the integration of the tray equations and the additional features that have had to be added to the implicit integration routines in order to deal with the three classes of state variables incorporated in these equations (pure inputs, coupled differential-algebraic states, non-states). We also motivated convergence of Waveform Relaxation for the binary distillation model in the same sense as done by Mattisson in [32].

Some of the remaining tasks not addressed in this chapter include, broadly:

- Verification of results against a sequential distillation simulator.
- Extension to more rigorous thermodynamics.
- Extension to multicomponent systems.
- Enhanced software framework that allows the easy incorporation of new templates (*i.e.*, problem classes) and will permit a variety of chemical engineering simulations to be evaluated with Waveform Relaxation.
- Extended software capability that allows competing Waveform Relaxation approaches (*e.g.*, [39]) to be compared fairly.
- Extension to systems modeled with partial differential equations, incorporation of multigrid methods and efficient implementation/extension of Waveform Relaxation ideas enunciated in [60,59] for PDE systems.

Chapter 8

Conclusions, Future Proposed Work and Recommendations

Abstract

This chapter draws together the seven preceeding chapters, as well as the appendices that follow, summarizing the progress implied by this thesis work. We indicate some of the research issues that should be tackled in the upcoming years to make concurrent computation an effective, widespread methodology for chemical engineering research and practice. We also indicate a number of practical problems that must be resolved. For example, we indicate the changes in academic and industrial attitudes towards computational algorithms research that are needed to permit new generations of chemical engineers to become literate, effective computational engineers as well. In this discussion, we indicate our recommendations not only for future work, but also for the study of concurrent computation on applications in chemical engineering other than our present focus, simulation of systems of differential-algebraic equations (dynamic flowsheet simulation).

8.1 Perspective and Summary

First, I wish to consider how far we've come in the six years constituting the span of this thesis research. In July 1984, when this work began, there were no commercial concurrent computers available and the experimental machines built at Caltech

were in their infancy. The prospective for per-node floating point performance was about 50,000 flops, with memory capacities of no more than 512K bytes. We expected to have machines with 32–128 nodes. Communication latency between near neighbors could be around 2,000 microseconds, and much more for far neighbors. Systems' Software-wise, we expected almost nothing – a C compiler, and low-abstraction, hypercube-oriented operating systems (no point-to-point communication primitives and no debugging support). Stability of the hardware and software were also questionable. Application software (like linear algebra codes) were non-existent.

In the intervening years, several generations of commercial machines have come into being, and concurrent supercomputers are in the offing, each node capable of producing many megaflops, and running a Mach (Unix-like) operating system kernel. Even now, per-node memories of 4M bytes are now common; message passing technology with application-level latency of 250 microseconds exist (with two orders of magnitude lower latency at the primitive hardware level). Scalar floating point from 100,000 to two million flops are commonplace. Machines of up to 2,048 nodes are expected shortly. Operating systems are much improved (including point-to-point communication); the standard C and Fortran languages are available and many vendors have stable compiler and operating-system products, though not all. There is some support for debugging. Until recently, it has been virtually impossible to find a vendor-independent application code of high-quality and portability that can plug into a new application or research program (though we have now created some). No convenient, general purpose, portable abstractions of message passing were available to improve programmability of complex applications until we addressed this need ourselves.

As far as chemical engineering, we had absolutely nothing in 1984 for multicomputers, and no experience for how to program them nor of what to expect. Now, we have experience with the parallelization of extant high-quality sequential algorithms,

and with the utilization of novel numerical techniques. We have high quality algorithms and their realizations in working codes for integration of systems of differential-algebraic equations, solution of linear systems of sparse, unsymmetric matrices, and many support libraries. We have a clear idea of achievable concurrent performance, of bottlenecks, and of the important features of a chemical-engineering problem class that dictate the degree of performance we might expect, for example the Jacobian structure, or the per-equation work in thermodynamic calculations. We have the ability to simulate distillation column networks with two simulation paradigms. Most importantly, with the ideas, algorithms and working production codes developed in this thesis work, we can begin to attack many more chemical-engineering problems in the future. We also expect to be ready to harness concurrent supercomputers when they arrive in the next two to three years. As such, our work has reacted to the existing multicomputer technology, and also anticipates future machines through portability factors, and careful design for data-distribution-independent correctness of the algorithms. In the next year, we will create a multicomputer toolbox to provide a portable basis for additional multicomputer research and development in chemical engineering, and beyond.

8.2 Recommendations for the Future

The most important fact to realize in multicomputer research is that there is a delicate balance between the computational science and the application area – both are needed, as we pointed out in the introduction to this work. However, there is also a delicate balance between algorithms research and implementation, both are also needed, though implementation has a "low-brow" reputation. In fact, to the uninitiated, the entire area of multicomputer algorithms research appears to be a mundane exercise in code implementation, or a "support activity." Undertaken correctly, this research area is far apart from those antiquated notions.

Chemical engineers need to know more about computation rather than just about Fortran programming. They need to know more about applied mathematics (like well-posedness), and about some of the fundamental limitations of computing (like complexity and uncomputability). Since supercomputers are meant to be the key mechanism for greater understanding of physical systems and models in the future, each engineer will need to understand concepts, theory and practice of computational science in order to be able to generate the research and complex tools needed to investigate important, complex chemical engineering systems. Some chemical engineers will have to specialize in this area, but all chemical engineers will have to become more literate. They will also have to develop better practices for their software realizations of research and applied work, because software and related data will increasingly become the fundamental representation of new knowledge and technology.

The antiquated notions will have to be overcome in academia first, otherwise undergraduate and graduate engineers will not be exposed to an open-minded attitude concerning computation. It will be the key role of the academic environment to create new computational knowledge for industry, because industry (at least U.S. industry) is too keenly aware of its quarterly bottom-line, and cannot invest the many years needed to see concurrent computation to its fruition. However, academia can produce the needed knowledge and technology transfer to industry, in order to overcome this barrier. First, however, computational science will have to be accepted as a legitimate research activity within the field. The profession does not need "parallel computing" as its next bandwagon, only to be dropped again after a few years. It needs acceptance, and concerted research effort. Many applications in chemical engineering are suitable for high performance computing; there is much new knowledge to be gained by investigating nature and systems with a high performance computational tool as well.

So, another balance must be struck in this work. We must be application driven on

the one hand. We have problems of importance, and we need to make them tractable. Or, we have problems we need to solve more effectively, in order to open new pathways to higher efficiency operations. On the other hand, and of equal importance, we must develop a basis of algorithmic understanding and understanding of concurrent supercomputation, and ask what problems we can attack given this knowledge. This latter approach is frowned upon as the "solution looking for a problem." Intellectually, both a top-down and a down-up approach are needed, however. One drives the other.

There are many theoretical problems to be solved in concurrent computation and modeling. Probably, the most important intellectual effort will go into the creation of new classes of concurrent algorithms that are most suitable for classes of concurrent machines. This research need will not be easily satisfied. Complementarily, we must solve a number of engineering-type problems: the creation of complex software systems (computer-aided design) for the purpose of doing high-performance computational science. We will need to learn how to utilize computer-aided design ideas from scratch, to some extent. We will need to do more than what computer scientists and engineers can themselves do for us — we will need to create "environments," tools and knowledge suitable for our own problems.

8.3 Specific Future Work

Apart from the above sincere statements of philosophy and needed changes, we plan extensions to the present research work. As indicated in chapters 6 and 7, novel numerical approaches are evidently key to high performance dynamic simulation of DAE systems. We propose to explore both "semi-classical" and completely new approaches. To do so, we will investigate ways to change the *DASSL* algorithm incrementally for higher concurrency. On the other hand, we will investigate the numerical properties and practical validity of Waveform Relaxation for interesting systems. We will explore higher performance concurrent sparse solvers, and the utilization of iterative linear

algebra as a further means to widen the applicability (and increase the performance) of *Concurrent DASSL*.

The algorithms we develop will be of minor interest if there are no means to generate and solve complex engineering problems with them. To do so, we must invest time and research plus development effort toward the creation of more powerful simulation layers (problem formulators). In doing so, we will engage ourselves in the research issues of problem formulation, equation ordering, model representation, and the fundamental tradeoffs between concurrency and convergence. We must look again at all the well-established flowsheeting ideas, and at the object-oriented systems. They must be rethought fundamentally now that concurrency is to be the new degree of freedom. Of course, our research will specialize to include the current ideas (sequential case) as a special case.

Finally, we need to import more knowledge from applied mathematics to make this all happen. We need to understand more subtle aspects of numerical algorithms if we plan to create novel algorithmic approaches. We also need to stay abreast of general-purpose methods (like new types of iterative linear algebra) that may be of real interest, but as yet unused in common chemical engineering practice. We need to develop a basis for communication with other engineering disciplines. Now, because of notation and practice in electrical engineering, for example, it is very difficult to discuss the similarities and differences of our respective applications under the Waveform Relaxation paradigm. We need to extend the formal notations of our respective areas to create better modelling notions, and promote better communication. We plan to continue collaboration with electrical engineers to learn more about their applications, and see if we can encompass our respective problem domains in a single concurrent simulation paradigm.

Finally, we need to make industry function better by putting key test systems into effect with industrial partners, proving the validity of this future work. This will

prove easiest in off-line applications; we believe that some big-return dynamic on-line simulation/optimization problems (like pipelines) will also be good testing grounds for our new research. After all, the goal of chemical engineering research is ultimately to improve practice.

Appendix A
More Concurrency Kernels

Abstract

We define additional concepts involving data locality and concurrency kernels – basic operations with interesting properties. For some of the kernels, we follow the lead of Van de Velde and others [54,55,20].

Except as noted, the data distributions mentioned here are assumed to be strong, such as those introduced in chapter 4 and at the beginning of appendix C. However, we extend parts of our discussion to weak data distributions with the help of further definitions and considerations from appendix C.

A.1 Definitions

To unify the discussion of kernels, we discuss data distribution projections first.

Definition A.1 (Data-Distribution Projection) *A data-distribution projection is a mapping of the form*

$$\Gamma_{\nu\circ\mu^{-1}}(p,i,P,M;Q,N) \equiv \nu\left(\mu^{-1}(p,i,P,M),Q,N\right) \mapsto (q,j) \tag{A.1}$$

which exists for all (p,i) if and only if $N \geq M$, where

$$\mathcal{G}^I \equiv \left\{(\mu,\mu^{-1},\mu^\sharp);P,M\right\}, \tag{A.2}$$

$$\mathcal{G}^{II} \equiv \left\{(\nu, \nu^{-1}, \nu^{\sharp}); Q, N\right\}, \tag{A.3}$$

are data distributions, and where (p,i) *(*(q,j)*) is a valid image in the data distribution* $\mathcal{G}^I$ *(resp.,* $\mathcal{G}^{II}$*). Equation A.1's "inverse" is defined as:*

$$\Gamma_{\mu o \nu^{-1}}(q,j,Q,N;P,M) \equiv \mu\left(\nu^{-1}(q,j,Q,N),P,M\right) \mapsto (p,i), \tag{A.4}$$

as expected, which exists for all (q,j) *if and only if* $M \geq N$*.*

We immediately restrict attention to the case $M = N$*. Then, Equation A.1 gives the process* q *and local offset* j *in distribution* $\mathcal{G}^{II}$ *of the coefficient* $I = \mu^{-1}(p,i,P,N)$*, where* (p,i) *is a process and local coefficient index in distribution* $\mathcal{G}^I$*. Equation A.4 provides the inverse mapping* $(q,j) \mapsto (p,i)$*.*

Data-distribution projections are used in the transformation and identification of invariant properties of coefficients. For example, the set of diagonals of a matrix in a process are deduced through a data-distribution projection. Furthermore, the conversion of a row-oriented concurrent vector to a column-oriented concurrent vector (or vice-versa), requires a data-distribution projection in each process of the underlying process grid; we will return to these points below.

In the practical applications just cited, a weaker form of data-distribution projection is used, as follows:

Definition A.2 (Local Projection) *Assuming* $M = N$*, the local projection of* $\mathcal{G}^I$ *onto* $\mathcal{G}^{II}$ *is defined in each process* p*,* $0 \leq p < P$*, as*

$$\Upsilon_{\nu o \mu^{-1}}(p,i,P,Q,N) \equiv \begin{cases} j & \text{for } q = p \\ -1 & \text{otherwise} \end{cases}, \tag{A.5}$$

and

$$\Upsilon_{\mu\circ\nu^{-1}}(q,j,Q,P,N) \equiv \begin{cases} i & \text{for } p = q \\ -1 & \text{otherwise} \end{cases}, \tag{A.6}$$

where -1 *connotes "none."*

Equation A.5 (A.6), gives the local coefficient j (resp., i) corresponding to local coefficient i (resp., j), or -1 if there is no such local correspondence. These projections indicate the intraprocess invariants of a change of data distribution. Clearly, each of these mappings can be computed once, and stored using memory in each process p, $0 \leq p < P$, proportional to $\mu^\sharp(p,P,N)$ (resp., $\nu^\sharp(p,Q,N)$).

Lemma A.1 (Local Projections & Weak Distributions) *The local projections are correct if weak data distributions* ω, υ *substitute for the strong data distributions* μ, ν *in Equations A.5, A.6. See appendix C for more details.*

Proof We prove correctness for Equation A.5. Equation A.6 follows analogously. By definition of a weak data distribution, inside a process p the global coefficient $I = \omega^{-1}(p,i,P,N)$ corresponding to the local coefficient i is well defined. Consequently, the first part of the local projection may be calculated. Secondly, given any global coefficient I, the weak distribution υ can identify the p-local coefficient corresponding to this global coefficient, or flag the absence (by -1) of such correspondence within the process p. Therefore, $\upsilon(I,Q,N)$ immediately completes the local projection,

$$\Upsilon_{\upsilon\circ\omega^{-1}}(p,i,P,Q,N), \tag{A.7}$$

as claimed. ■

Corollary A.1 (Mixtures of Strengths) *The local data-distribution projections below (section A.2) are correct if either strong data distribution* μ *or* ν *in Equations A.5, A.6 is replaced by a weak data distribution.*

Proof By inspection. ■

Corollary A.2 (Continued Correctness of Kernels) *Important consequences of the previous lemma and its first corollary are that concurrency kernels depending on local projections; transpose_row_to_column, transpose_column_to_row, and skew_inner_product (defined below) remain correct.*

Proof By inspection. ■

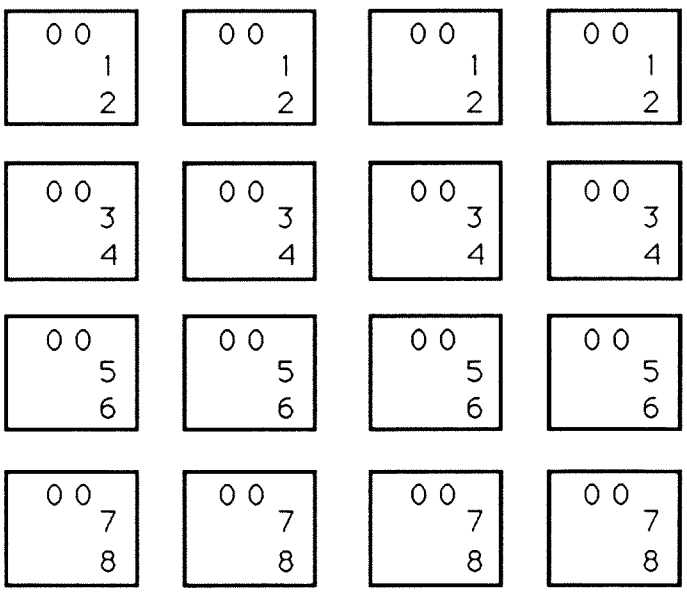
A.2 Kernels

In chapter 2, we introduced selected concurrent operations: *combine*, *broadcast*, weighted-vector sum, and matrix-vector product. The former two are common in all multicomputing; the latter two are prevalent in the implementation of linear algebra operations other than *LU* factorization (see [55]). We also had occasion to discuss norms of concurrent vectors in chapter 6, in conjunction with the discussion of *Concurrent DASSL*.

Here, we introduce three additional concurrency kernels of interest. The first two are closely related: *transpose_row_to_column* and *transpose_column_to_row*. The former converts a row-distributed concurrent vector arrayed on a two-dimensional logical process grid $\mathcal{G}$ to a column-oriented concurrent vector on the same grid. The second kernel does the opposite: a column-distributed vector is transformed into a row-distributed vector. The third kernel is a special version of inner product, "skew inner product." It forms the inner product of a row-distributed and a column-distributed vector; in connection with it, we also discuss the regular inner product operation, a slight generalization of an unweighted norm operation. Skew inner product illustrates the importance of pipelining operations in sequence in a multicomputer. The same operation can be accomplished first by "transposing" one vector into the same distribution as the second, and then by performing a standard inner product. However, the

latter requires two *combine* operations instead of one. Skew inner product is useful in the definition of a multicomputer conjugate gradient kernel, but we don't pursue this further here.

Figure A.1. Step #1 of the *transpose_row_to_column* operation



Preparation for converting a row-oriented vector $(1,2,3,4,5,6,7,8)^T$ to a column-oriented format on a 4 x 4 process grid. The destination column-oriented vector has all its elements set to zero initially.

A.2.1 Vector Transpose Operations

Sometimes, vectors must be "transposed" from a row-distributed format to column-distributed format, or vice-versa. A good example of this arises in the back-solve stage of LU factorization, where the right-hand-side vector b is posed in a row-distributed fashion, but the solution vector x emerges in a column-distributed fashion. Almost always, x must be "transposed" back to a row-distributed format.

Below, we refer to vectors defined on a two-dimensional process grid $\mathcal{G} \equiv$

$(\mathcal{G}^{row}, \mathcal{G}^{col})$, formed from row and column data distributions

$$\mathcal{G}^{row} \equiv \left\{(\mu, \mu^{-1}, \mu^{\sharp}); P, M\right\},$$

and

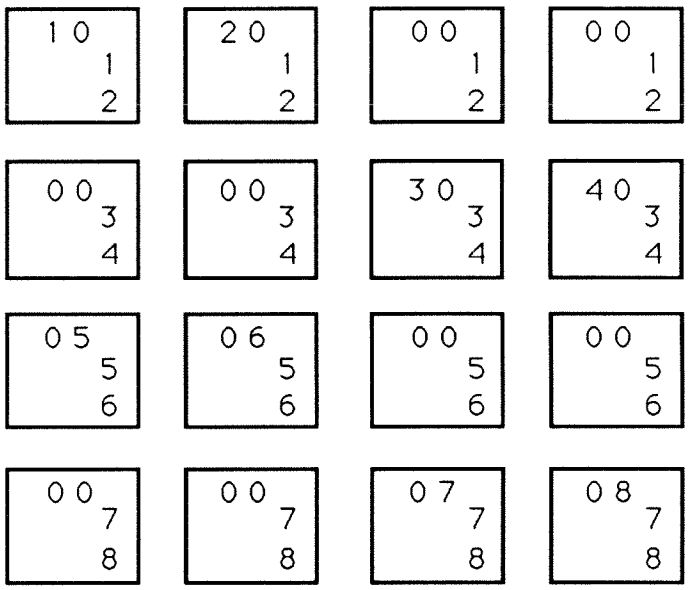
$$\mathcal{G}^{col} \equiv \left\{(\nu, \nu^{-1}, \nu^{\sharp}); Q, N\right\}.$$

See chapter 4 for further commentary.

Definition A.3 (*transpose_row_to_column*) *The originating row-distributed vector is denoted x, while the final column-distributed vector is denoted y. Both are defined on $\mathcal{G}$. In each process, all the elements of y are initialized to zero. Then, in each process p, $0 \leq p < P$, elements corresponding to the local projection coefficients of $\mathcal{G}^{row}$ onto $\mathcal{G}^{col}$ are copied from their locations in the local vector x to the local vector y. Remaining elements in each local y vector remain zero. Finally, and independently in each process column, the local y vectors are combined (vertically on the grid) using element-by-element addition as the associative-commutative combination operation. This has the effect of propagating the completed y vector to each process row, appropriately distributed in the process columns. Element-by-element addition can be used because we initialized all elements of the y vectors to zero before copying based on local projections.*

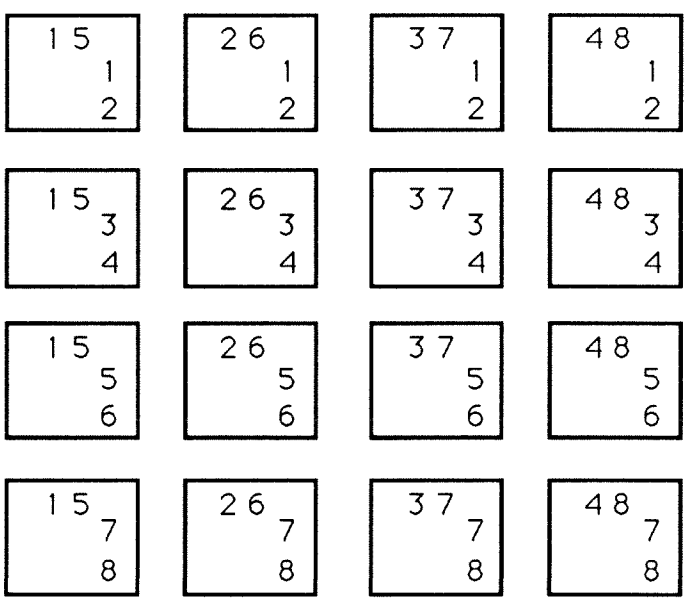
Figure A.1. illustrates the initial state for an eight-element, row-distributed vector $x = (1,2,3,4,5,6,7,8)^T$ on a 4×4 process grid. In the figure, the column-distributed vector has been initialized to zero. For this example, we choose a linear row distribution ($\mu = \hat{\lambda}$) and a scatter column distribution ($\nu = \hat{\sigma}$). In Figure A.2., copying of elements according to the local projections has been accomplished. The rest of the elements remain zero. In Figure A.3., the column-wise *combine* operations have been effected, completing the conversion.

Figure A.2. Step #2 of the *transpose_row_to_column* operation



According to the local projection of coefficients, appropriate elements are copied into the column-oriented vectors. The remaining elements in the column-oriented vector are untouched and remain zero.

Figure A.3. Step #3 of the *transpose_row_to_column* operation



Applying a *combine* to each process column converts the representation of Step #2 in Figure A.2 to the final form depicted here. The combination operation is a simple element-by-element addition.

Definition A.4 (*transpose_column_to_row*) *The operation is analogous to transpose_row_to_column. The originating column-distributed vector is denoted y, while the final row-distributed vector is denoted x. Both are defined on $\mathcal{G}$. Initially, in each process p, $0 \leq p < P$, all elements of the local vector x are set to zero. Then, corresponding to the local projection of $\mathcal{G}^{col}$ onto $\mathcal{G}^{row}$, selected elements from y are copied into x. Finally, and independently in each process row, the local x vectors are combined (horizontally on the grid), to complete the transformation. Again, element-by-element addition is used as the associative-commutative operation. Initialization of x to zero at the outset allows for the negligible "bookkeeping" inherent in this conversion procedure.*

A.2.2 Inner Products

First we define conventional inner products per, for example, [54,55]. Then we consider a new kernel, the skew inner product.

Definition A.5 (Inner Product) *The simplest inner product $x^T y$ of two concurrent vectors x and y involves a pair of compatibly arrayed row-distributed (or column-distributed) vectors. In the simplest form, each process column (resp., row) works independently to form the inner product of two row-distributed (resp., column-distributed) vectors. First, the local inner products are formed in each process, then a column-wise (resp., row-wise) combine is effected to complete the inner product in each process column (resp., row). For the row-oriented case, this procedure has a maximum speedup potential of P, the number of process rows, and communication complexity $O(\lceil \log_2 P \rceil)$, in view of the column-wise combines. For the column-oriented case, this procedure has a maximum speedup potential of Q, the number of process columns, and a communication complexity of $O(\lceil \log_2 Q \rceil)$, because of the row-wise combine operations.*

Alternatively, as suggested by Van de Velde, and as noted in chapter 6, we can elect further to reduce the computation effort in the inner product calculation at the expense of greater communication. We consider only the row-distributed case for brevity. If process column q, $0 \leq q < Q$, computes only one of every Q terms of its local inner product (Q-striding), and starts with local coefficient $j = q$, then the process columns produce no repetitive terms in the inner product. A combine over the whole process grid (more communication than before) sums the global result. The maximum potential speedup is now PQ, while the communication complexity becomes $O(\lceil \log_2 PQ \rceil)$.

Either of these approaches is arguably better for differing circumstances, grid shapes, and load-balance characteristics of operations preceeding and proceeding the inner product.

Definition A.6 (Skew Inner Product) *Skew inner product combines the ideas of transpose_row_to_column (or equivalently transpose_column_to_row) and the striding inner product. One valid definition is as follows.*

Given a row-distributed vector x and a column distributed vector y, we again wish to form $x^T y$, the inner product. Starting in each process with a local sum equal to zero, we form the inner product of x terms with compatible y terms according to the local projection of the row distribution onto the column distribution. Each process subsequently contains a unique part of the overall sum. A global combine over the whole grid completes the inner product calculation.

Using the alternative projection produces the same value for the inner product, but different partial sums in each process (in general) prior to the *combine*.

We note in closing that the outer product xy^T (a rank-1 matrix) can be formed without any communication, provided that x and y are row- and column-oriented vectors, respectively. If either vector is ill-oriented, it must first be "transposed" to

the appropriate orientation.

Appendix B

Zipcode Internals and Use

Abstract

This appendix provides a more detailed look at the design and structure of *Zipcode*; it supplements the discussion of chapter 3. The purpose of this supplemental discussion is to make the *Zipcode* mechanisms clearer, to promote further applications in the *Zipcode* notation, and to render existing *Zipcode*-based applications amenable to study. For the sake of brevity, this discussion remains far from exhaustive. We consider data structures and a selected subset of "internal" and "public" primitives. We also mention ideas for augmenting the existing *Reactive Kernel* specification to allow greater efficiency for *Zipcode* global functions.

The following discussion relies on an understanding of Figure 3.1. as well as the rest of chapter 3.

B.1 Conventions

There are two classes of primitives in *Zipcode*, "internal" and "public." Internal primitives, whether macros or function calls, begin with an underscore character. By convention, underscore-macros are subject change in structure, specification, and naming, and should not be used in applications. Nonetheless, their properties illustrate important features of the workings of this communication layer, and must be included. The second class, public primitives, whether realized as macros or func-

tion calls, begin with alphabetic letters, and are admitted for use by applications. Nominally, these calls are not expected to change. We try not to draw distinctions between macros and function calls, except where this is essential, because such details may change between versions and also depending on conditional-compilation flags set within the system. In general, macros are used whenever possible to reduce software overheads.

B.2 Data Structures

First, we review the process-local data structures used by *Zipcode*. Then, we consider the data structures that play a role within letters.

B.2.1 Local Structures

Local structures are used to defined *Zipcode* objects such as mailers, addressee lists, and classes of mail.

Class As discussed in chapter 3, *Zipcode* classes define the style of receipt selectivity of letters. Within a process, a *Zipcode* class is represented as follows:

```
typedef struct zip_class
{
    short stamp;        /* cost of this class of mail          */
    short class;        /* numerical value for this class      */

    union y_cover *(*dlvry)();
                        /* default "receipt selection" fn.     */

    void  (*po_ctoh)(); /* Node to Host conversion of PO Box   */
    void  (*po_htoc)(); /* Host to Node conversion of PO Box   */
    short l_pobox;      /* length of PO Box structure          */
    short po_count;     /* 2nd argument to conversion routines */

} ZIP_CLASS;
```

including a `stamp`, the length of the "cover" (the aligned preamble constructed from

the "envelope" information plus padding), a unique pre-defined `class` number, and a default style for receipt selectivity defined through a function pointer `(*dlvry)()`. The remaining tags are two function pointers and supplemental parameters for converting the PO Box area when a letter arrives at or exits the host process, which may have a different architecture (and, consequently, data formats) than the node processes. This is transparent to the applications, as noted in the main text.

Addressees are the main point of *Zipcode* – maintaining and utilizing a set of participants for letter transmissions. In the current version, all lists are explicit enumerations of {node, pid}-pairs. Furthermore, in each process, the name of the process is itself promoted to the beginning of the list, being permuted with whatever process is actually first in the list (the "postmaster" or "PM" process). This permutation, indicated by the `permute` tag, is useful for broadcasting operations that exclude the originating process. (The `permute` tag is equal to the offset of the local process in the original unpermuted list.)

```
typedef struct zip_addressees
{
    int   n_addressees; /* number of addressees */
    int   *addressees;  /* list of addressees */
    int   permute;      /* permutation for local rearrangement */

} ZIP_ADDRESSEES;
```

Mailer A mailer is the culmination of the previous data structures. Given a class structure `cl`, a `zipcode` (context number), and addressees `addr`, a basic mailer is realized. In addition, a mailer contains a default PO Box area denoted `pobox`, used by specific macros as the default data with which to mark letters before transmission or upon which to base receipt selectivity. Furthermore, a class-dependent `extra` area is supported that, as needed, may contain parameters associated with a given instantiation of a class (such as the grid shape $P \times Q \times R$ of a three-dimensional

grid). Finally, inheritance is indicated by linkages to parents and childen; linkages are determined at the time the mailer is created. Particular children are created by default, but others may be created by the application with no loss of generality; the desired properties of children are determined by their specializations of their parent's addressee list. Any number of children may be defined, but it was initially convenient to pose them in three categories for easier pointer access. While this choice is largely arbitrary, it proves convenient from an internal, implementation point-of-view.

```
typedef struct zip_mailer
{
    ZIP_CLASS cl;    /* mail class information          */

    short zipcode;    /* zipcode for the mailer          */
    void  *pobox;     /* "receipt selection" information */
    void  *extra;     /* extra mailer-specific data      */

    struct zip_mailer *rome;   /* ultimate parent        */
    struct zip_mailer *parent; /* immediate parent       */
    struct zip_mailer *schild; /* sinister     child[ren] */
    struct zip_mailer *dchild; /* dexterous    child[ren] */
    struct zip_mailer *ichild; /* illegitimate child[ren] */

    ZIP_ADDRESSEES addr;     /* this mailer's addressees: */

} ZIP_MAILER;
```

B.2.2 Letter Structures

Next, we consider a subset of the data structures that comprise letters – envelopes and covers. We present the Y-class envelope and cover as an example. Others are quite similar in structure. The envelope for Y-class mail is defined as follows:

```
typedef short Y_POBOX;
#define L_Y_POBOX sizeof(Y_POBOX)

typedef struct y_envlp /* Y-class mail Envelope.  */
{
    /* Primary data shared by all mail classes */
```

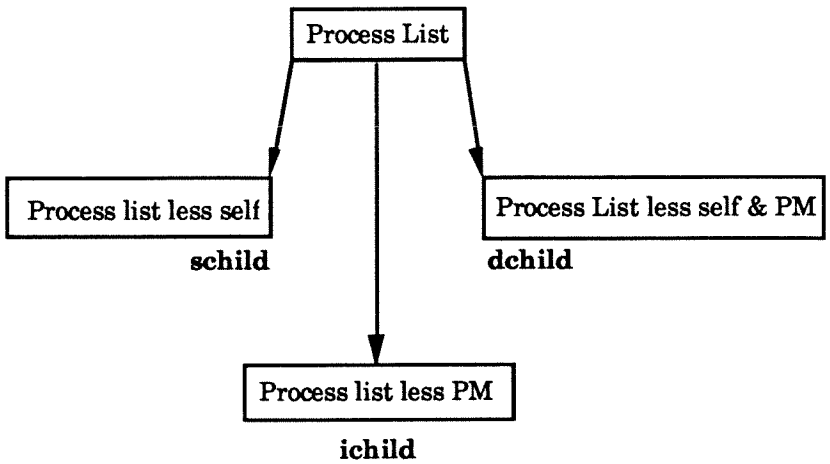
```
    short  zipcode; /* mailer identification       */
    short  class;   /* class  of mail              */
    short  stamp;   /* length of header            */
    short  unused;  /* makes mini alignment        */

    /* used in queueing of received letters: */
    union y_cover *next;

    /* Receipt selectivity for Y-class: */
    Y_POBOX  pobox;

} Y_ENVLP;
#define L_Y_ENVLP sizeof(Y_ENVLP)
#define Y_SHORTS 4 /* # of shorts leading all envelopes */
```

Other mail classes replace the PO Box structure **pobox**, but must retain the tags that preceed it. In the preceding, the **next** pointer is used to build linked lists of messages in the receive queues, thereby reducing overheads for queueing at the negligible expense of slightly longer transmissions.

The Y-class cover is created by aligning the envelope in a union, where L_ZIP_ALIGN is the appropriate alignment length (*e.g.,* 4 bytes), and where L_ZIP_SHORT ≡ sizeof(short) accounts for the "return postage" at the end of the cover, as follows:

```
#define Y_MODL ((L_Y_ENVLP + L_ZIP_SHORT) % L_ZIP_ALIGN)
#define Y_DIVL  ((L_Y_ENVLP + L_ZIP_SHORT) / L_ZIP_ALIGN)
#define Y_ALIGN (L_ZIP_ALIGN * (Y_DIVL + !(!Y_MODL)))

typedef union y_cover
{
    Y_ENVLP env;
    char align[Y_ALIGN]; /* alignment length */

} Y_COVER;
```

Figure B.1. *Zipcode* Generic Mailer Geneology



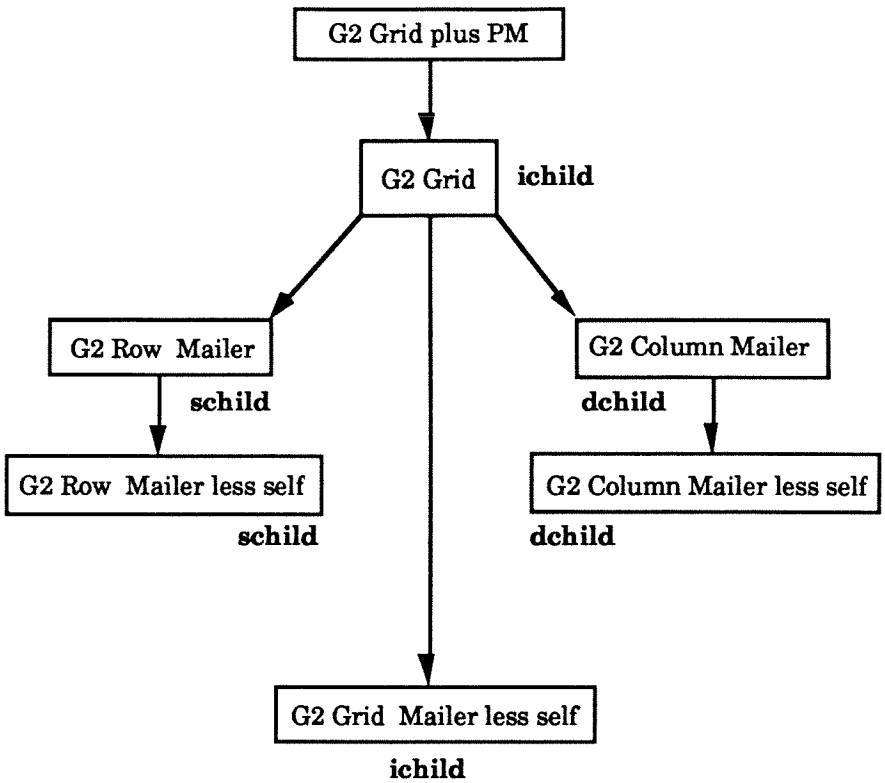
This figure illustrates the hierarchy of mailers created by `yopen()`, independent of class. In practice, the `yopen()` call is used thus far for the creation of Y- and Z-class mailers.

B.2.3 Illustrative Macros/Calls

Reflecting on the basic structure of *Zipcode* covers (padded envelopes), the following macros are used to transform between `letter` pointers, pointers used by the application level to refer to *Zipcode* letters, and `envlp` pointers, pointers referring to the start of letter envelopes, and manipulated exclusively by the *Zipcode* layer. The inverse mapping `letter` $\mapsto$ `envlp` is accomplished via the "return postage," a second copy of the cover's length. It's stored in the short word prior to the beginning of the letter space addressed by `letter`; an area at the end of the cover's alignment padding. This issue arises because different classes of mail have envelopes (and hence covers) of arbitrarily different lengths. The forward mapping `envlp` $\mapsto$ `letter` uses the `stamp` value stored at a constant offset from the beginning of the envelope for all classes of mail.

```
Letter-to-Envelope Conversion:

#define _y_lstamp(letter)  ((int))(((short *)(letter))[-1]))
#define _y_ltoe(letter)    ((((char *)(letter)) - _y_lstamp((letter)))
```

Figure B.2. *Zipcode* G2-Class 2D-Grid Mailer Geneology



This figure illustrates the hierarchy of mailers created by `_g2_grid_open()` and `g2_grid_open()`, the more commonly used call. "G2 Grid plus PM" differs from "G2 Grid" only if the initiating process (the postmaster or "PM") is not part of the logical grid. The mailer pointer returned by these functions is actually "G2 Grid." Row and Column children are also created, as noted.

and

```
Envelope-to-Letter Conversion:

#define _y_etol(envlp)  (((char *)(envlp))+(Y_ENVLP *)envlp)->stamp))
```

These conversions are used pervasively in the *Zipcode* system.

The following illustrates how letters are allocated and freed:

```
#define ymalloc(mailer, length)
        ((mailer) ? (_ymalloc((length), ((mailer)->cl.stamp))) : NULL)
#define yfree(letter)
        ((letter) ? Xfree(_y_ltoe((letter))) : ZIP_ERR)
```

where _ymalloc() is the low-level call that allocates letters of all classes using xmalloc(), while accounting for the need for *Zipcode* covers.

A number of class-independent transmission macros are supported, exemplified by the following:

Class-Independent Transmission Workhorse:

```
#define ypomsend(mailer, letter, c, d, PO)                    \
{                                                             \
   if ((mailer) && (letter) && (c > 0) && (d != NULL))        \
   {                                                          \
      Y_COVER *covr = (Y_COVER *)_y_ltoe((letter));           \
      covr -> env.zipcode = (mailer) -> zipcode;              \
      covr -> env.class   = (mailer) -> cl.class;             \
                                                              \
      if(PO)                                                  \
      {                                                       \
          bcopy((char *)(PO),                                 \
                (char *)&(covr -> env.pobox),                 \
                (mailer) -> cl.l_pobox);                      \
      }                                                       \
                                                              \
      Xmsend((char *)covr,(c),(d));                           \
   }                                                          \
   else if(letter) yfree(letter);                             \
                                                              \
   letter = NULL;                                             \
}
```

Finally, as noted in the text, yopen() is the class-independent means for creation of mailers. In Figure B.1. we note the hierarchy of mailers created by this call.

B.3 G2-Class Calls

Several G2-Class 2D-grid primitives were introduced in chapter 3. Here, we introduce two more. When entering a function that uses G2-Class mail, it is necessary to determine the grid shape $P \times Q$, as well as the current processes' location on the grid (p, q). Often this information is housed only in the mailer (though some appli-

cations may choose to duplicate this information). The following context-dependent (with respect to the mailer most recently stacked with `Ypush()`) calls provide simple access to these four quantities, while hiding the structure of the G2-Class `extra` area, something with which applications needn't concern themselves directly:

```
G2_PQ(P, Q); /* set variables (P,Q) to grid shape */
G2_pq(p, q); /* set variables (p,q) to grid position */
```

These are the preferred forms for accessing grid information from G2-Class mailers. The context-independent calls require more machinery, and are omitted here for brevity.

In passing, we indicate the hierarchy of G2-Class 2D-grid mailers created by `g2_grid_open()` and `_g2_grid_open()`. This is illustrated in Figure B.2. Note that this hierarchy is hard-wired by the former function, but can be bypassed by the latter for more elaborate applications.

B.4 Interstitial Layers

We note that it may prove interesting to interpose a communication protocol between the *Reactive Kernel* (*RK*) primitives and *Zipcode*. Possible uses of such "interstitial" layers are debugging support, and a mechanism to provide reactive multicast message support at a very low level [34], basically a broadcast mechanism, albeit of a narrow scope. That *Zipcode* does not preclude such added functionality is as important as the more specific comments that follow.

Interstitial layers are functionally invisible to the *Zipcode* system, but necessarily impact performance. They replace the *RK* functions `xsend()`, `xmsend()`, `xrecv()`, `xrecvb()`, `xmalloc()` and `xfree()` with functional look-alikes. The *Zipcode* header file defines macros `Xsend()`, `Xmsend()`, `Xrecv()`, `Xrecvb()`, `Xmalloc()` and `Xfree()` to facilitiate softer compile-time binding to the actual *RK* primitives; *Zipcode* macros and calls refer only to the capital-X macros. For example, by conditional compilation,

a debugging layer can be activated that provides diagnostics between the *RK* level and *Zipcode*-level calls; this is done by changing the definitions of the capital-X macros to refer to debugging primitives, which themselves call the *RK* primitives in turn. Currently, the debugging formulation does not alter the contents of *Zipcode* letters, but this could be changed in a more sophisticated debugging mechanism (*e.g.*, extended PO Boxes with diagnostic quantities such as checksums). Softening the runtime-binding to the *RK* primitives can also be considered through the use of function pointers, implying a small additional indirection overhead for each call.

A multicast layer would augment the *RK* message with a preamble, just as the *Zipcode* layer adds a preamble to form letters. The multicast mechanism would be handled through a low-level queueing mechanism below *Zipcode* that would selectively retransmit specially packaged "chain letters" after receiving them via conventional `xrecv[b]()` calls, and before (possibly) placing them on the local `Xrecv[b]()` queue for receipt by *Zipcode*-layer functions. The chain letter's preamble, in general, must contain delivery information, such as an explicit {node, pid}-list of recipients, and a checklist mechanism to indicate which recipients have already had the chain letter delivered. Any valid distribution mechanism may be chosen, independent of the *Zipcode* mailer and letter conventions, so long as the apparent *RK*-defined functionality (*e.g.*, the pairwise ordering property of messages between processes) is maintained. The payload of the chain letter might itself be a *Zipcode* letter of a particular class, or could be completely invisible to the *Zipcode* queues, depending on application needs.

B.5 Suggested Additions to the *Reactive Kernel*

Much higher performance for the entire *Zipcode* system would result if the system could be defined at the "reactive handler" level – the level upon which the public *RK* primitives are constructed. This would imply machine-dependent implementations

of *Zipcode*. Unfortunately, machine-level access will not be uniformly feasible across architectures, because of vendor restrictions and so forth. We propose a compromise: a few, extended *RK*-compatible primitives, that can be supported at an efficient hardware level by *RK* or by similar operating systems. This approach addresses the need for higher efficiency at two levels: first, efficient *RK* implementations seem likely for a number of machines and, second, other operating systems over time may upgrade to be *RK* functional look-alikes or upward compatible. Existing *Zipcode* global operations *combine* and *broadcast* would layer naturally atop these added primitives, requiring no change to programs written in the *Zipcode* notation.

Additions to *RK* of this nature are evidently planned, and we are hopeful that our suggestions will be considered seriously in the formation of its extended "user interface" for global concurrent operations.

Combine **operation** We propose the following user interface:

```
xcombine(proc_list_spec, buffer, comb_fn, size, nitems);
```

where `proc_list_spec` contains the following tags:

- The number of participating processes, N,
  and
- An array of processes specified by (node, pid)
  or
- A user-specified function pointer implementing a bijection that maps $0, \ldots, N-1 \mapsto$ {node,pid} pairs

where `buffer` is a storage block of length `items` $\times$ `size` bytes containing the initial local contribution to the result, and the result at termination of the *combine*. The function pointer `comb_fn` addresses a canonical associative-commutative combination operation, with the following calling format:

```
comb_fn(buffer1, buffer2, size, items)
void *buffer1, *buffer2; /* data to be 'combined'      */
int size, items;         /* size and number of objects */
```

We use many different `comb_fn`'s in practice. It is practically important not to be restricted to a few arbitrarily chosen operations. However, if one could provide a much faster version for a few key operations, this option would also be interesting, but which operations would one pick? The choices would have to be application-motivated, and would quickly become parochial; however, a reasonable set might be logical operations for integers, plus min/max, sum, difference, and product of integers and reals, including the possibility to identify the process that produced the extremal result for min/max. Independently, we are certain that fast synchronizations are a useful special case of *combine* that need to be defined:

```
xsync(proc_list_spec);
```

Messages transmitted by `xcombine()`, `xsync()` must of course be invisible to the `xrecv[b]()` calls.

Broadcast The *broadcast* (*fanout*) operation, as mentioned repeatedly in this thesis, starts with a single source process; all participants know the source. After a logarithmic number of steps in the number of participants, each participant has the result that originated in the source process. We propose the following calling sequence:

```
xfanout(proc_list_spec, source_proc_no, data_spec)
```

where `source_proc_no` is the number of the initiating process within the `proc_list_spec`, and `data_spec` is a data specification of the following type:

```
typedef struct fanout_data_spec
{
```

```
    void **data;            /* where the data will be stored   */
    int  *length;           /* how long the data is or can be  */
    void *(*malloc_fn)();   /* mechanism for allocating **data */

} FANOUT_DATA_SPEC;
```

Only the initiating process needs to pre-allocate the data. If `*data` is `NULL` in other processes, the data space will be allocated dynamically in `(*malloc_fn)()` space and `length` will be set to reflect the size of the data received. If `*data` is not `NULL`, then `*length` must contain the length of `*data`, and an error occurs if insufficient space is available, based on the length of the received data. `malloc_fn` is a `malloc`-like function pointer (*e.g.*, `malloc()`, `xmalloc()`). Again, this operation's messages must be independent of `xrecv[b]()` and `xcombine()` to ensure determinism, and to allow the high-level layers to be built on top of *RK* seamlessly.

Finally, it would be convenient to replace ad hoc process name conventions with the `proc_list` mechanism consistently. For instance,

`Creation/Destruction of Processes:`

```
xspawn("program_name", proc_list, "");
xmkill(proc_list);
```

and

`Generalized Multiple Sends:`

```
exmsend(proc_list_spec, msg);
exmsendb(proc_list_spec, msg);  /* blocked send */
char *msg;
```

Appendix C
Derivations of Data Distributions

Abstract

In this appendix, we cover additional data distributions functions. First, we cover the conventional data distributions functions linear and scatter, which antedate the generalized data distribution functions introduced in chapter 4. We also describe the block-linear and block-scatter data distributions, the simplest extensions of linear and scatter. We include derivations for selected data distributions presented in this thesis, as well as an example proof of correctness. To avoid confusion, all "ungeneralized" data distributions; namely, those that place load imbalances in lower-numbered processes (*e.g.*, linear and scatter), incorporate a hat as part of their name (*e.g.*, $\hat{\lambda}$ for linear *vs.* λ for generalized linear).

All of the closed-form $O(1)$-time, $O(1)$-memory data distribution functions are "strong" data distributions in the sense that each process has the entire mapping and inverse available to it without interprocess communication. We define "weak" data distributions (alluded to mainly in chapter 6). This formalism is used in appendices A, D to extend the usefulness of specific grid-oriented concurrent operations.

C.1 Definitions and Descriptions

In this section, we define the conventional distributions linear and scatter, and their simplest block forms. We follow [57, page 4] for the conventional linear and scatter

distribution formulas, though our notation differs slightly from the reference.

C.1.1 Conventional Functions

Definition C.1 (Linear) *The conventional linear, load-balanced distribution, inverse and coefficient cardinality functions are as follows:*

$$\begin{aligned}
\hat{\lambda}(I,P,M) &\mapsto (p,i), \\
p &\equiv \max\left(\left\lfloor \frac{I}{\hat{l}+1} \right\rfloor, \left\lfloor \frac{I-\hat{r}}{\hat{l}} \right\rfloor\right), && \text{(C.1)} \\
i &\equiv I - p\hat{l} - \min(p,\hat{r}), && \text{(C.2)}
\end{aligned}$$

while

$$\begin{aligned}
\hat{\lambda}^{-1}(p,i,P,M) &\equiv i + p\hat{l} + \min(p,\hat{r}) \mapsto I, && \text{(C.3)} \\
\hat{\lambda}^{\sharp}(p,P,M) &\equiv \left\lfloor \frac{M+P-1-p}{P} \right\rfloor, && \text{(C.4)}
\end{aligned}$$

where

$$\hat{l} = \left\lfloor \frac{M}{P} \right\rfloor, \quad \hat{r} = M \bmod P\ , \qquad \text{(C.5)}$$

and where we require $M \geq P$.

Description We defer the description to the block-linear case, which subsumes the linear, load-balanced distribution for the special case $B = 1$. See below. ■

Definition C.2 (Scatter) *The conventional scatter distribution, inverse and cardinality functions are as follows:*

$$\begin{aligned}
\hat{\sigma}(I,P,M) &\equiv \left(I \bmod P,\ \left\lfloor \frac{I}{P} \right\rfloor\right) \mapsto (p,i), && \text{(C.6)} \\
\hat{\sigma}^{-1}(p,i,P,M) &\equiv iP + p \mapsto I && \text{(C.7)}
\end{aligned}$$

and where

$$\hat{\sigma}^{\sharp}(p,P,M) \;=\; \hat{\lambda}^{\sharp}(p,P,M), \tag{C.8}$$

and where we require $M \geq P$.

Description We defer the description to the block-scatter case, which subsumes this distribution as the special case $B = 1$. See below. ■

C.1.2 Block Versions

For situations where $M = BP$, the following simple block data distributions are applicable:

Definition C.3 (Block-Linear) *The block-linear, load-balanced distribution, inverse and coefficient cardinality functions (with blocksize* $B \geq 1$*) are as follows:*

$$\begin{aligned}
\hat{\lambda}_B(I,P,M) &\mapsto (p,i), \\
p &\equiv \max\left(\left\lfloor \frac{I_B}{l+1} \right\rfloor, \left\lfloor \frac{I_B - r}{l} \right\rfloor\right), && \text{(C.9)} \\
i &\equiv I - B\left(pl + \min(p,r)\right), && \text{(C.10)}
\end{aligned}$$

while

$$\begin{aligned}
\hat{\lambda}_B^{-1}(p,i,P,M) &\equiv i + B\left(pl + \min(p,r)\right) \mapsto I, && \text{(C.11)} \\
\lambda_B^{\sharp}(p,P,M) &\equiv B\left\lfloor \frac{b+P-1-p}{P} \right\rfloor, && \text{(C.12)}
\end{aligned}$$

where b, l, r, I_B *are as defined in chapter 4 (in connection with the generalized distributions), but here* $M \bmod B = 0$ *and* $b \geq P$ *are both required for correctness. For* $B = 1$*, the linear load-balanced distribution is recovered.*

Description Global to Local: The quantity b is the total number of block elements of size B, l is the ideal number of block elements per process, while r is the number

of extra elements to be divided one each among the first r processes. I_B is the global block number within which the coefficient I resides. In the "head" regime, $0 \leq p < r$, we place $l+1$ blocks per process, while in the "tail" regime, $r \leq p < P$, we place l blocks per process; hence, to determine the process p within which block I_B resides, we must take the maximum of two terms:

$$\max\left(\left\lfloor \frac{I_B}{l+1} \right\rfloor, \left\lfloor \frac{I_B - r}{l} \right\rfloor\right). \tag{C.13}$$

Initially, the first term dominates, with the two terms becoming equal at $I_B = r(l+1)$; for $I_B > r(l+1)$, the second term dominates. Given p, pl constitutes the number of block elements in the first p processes assuming no imbalance, while $\min(p, r)$ accounts for extra block elements (but saturating at r). The sum of these terms multiplied by B comprises the total of coefficients used up by processes $0, \ldots, p-1$. Hence, subtracting this quantity from the global coordinate I gives the local coordinate i in process p.

Local to Global: The inverse is obtained by a simple rearrangement of the expression for i in terms of p and I.

Cardinality: There are b blocks to be divided among P processes. In the first r processes we choose to have $l+1$ block coefficients. The term $\lfloor (b+P-p-1)/P \rfloor$ is initially equal to $l+1$ (l) if $r > 0$ (resp., $r = 0$) and becomes equal to l exactly when $p = r$. By definition, $b = \lfloor M/B \rfloor$, so this is easy to see by recalling that

$$l = \left\lfloor \frac{b}{P} \right\rfloor, \quad r = b \bmod P\ , \tag{C.14}$$

and, consequently,

$$\left\lfloor \frac{b+\tilde{p}}{P} \right\rfloor = l+1 \quad \text{for } P - r \leq \tilde{p} \leq 2P - r - 1. \tag{C.15}$$

Once we have the proper number of block coefficients for process p, we scale up by the blocksize B to get the total number of coefficients. ■

Definition C.4 (Block-Scatter) *The block-scatter distribution, inverse and cardinality functions (with blocksize $B \geq 1$) are as follows:*

$$\hat{\sigma}(I,P,M) \equiv \left(I_B \bmod P,\ \left(B\left\lfloor\frac{I_B}{P}\right\rfloor + I \bmod B\right)\right) \mapsto (p,i), \quad \text{(C.16)}$$

$$\hat{\sigma}^{-1}(p,i,P,M) \equiv B\left(p + \left\lfloor\frac{i}{B}\right\rfloor P\right) + i \bmod B \ \mapsto\ I. \quad \text{(C.17)}$$

and where

$$\hat{\sigma}^{\sharp}_B(p,P,M) \equiv \hat{\lambda}^{\sharp}_B(p,P,M). \quad \text{(C.18)}$$

As for the block-linear distribution, $M \bmod B = 0$ and $b \geq P$ are required for correctness. For $B = 1$, the scatter distribution is recovered.

Description Global to Local: Again, I_B is the global block number containing coefficient I. Since we want to scatter blocks now, we use the expression $p = I_B \bmod P$ to move each successive block to the next highest process, modulo P. To find the local coefficient number i in process p, we notice that $\lfloor I_B/P \rfloor$ counts the number of completely filled-in scattered block layers in all processes by blocks $0,\ldots,I_B-1$. Hence, in process p, there are $\lfloor I_B/P \rfloor$ block coefficients already in place. There are consequently B times that many coefficients in place. Finally, the global coefficient I has an offset within the global block I_B equal to $I \bmod B$. This quantity must also be the offset within the scattered block, since we don't change the order of elements inside blocks. Therefore, the sum of $B\lfloor I_B/P \rfloor$ and $I \bmod B$ forms the local coefficient i, completing the construction.

Local to Global: We are given p and i. Now, $\lfloor i/B \rfloor$ is the number of block coefficients in process p filled-in before I_B. This quantity times P is the total number of complete block layers filled in before block I_B, and the extra offset p accounts for the incomplete

layer being filled in at the time I_B was scattered (that is, the block layer in which I_B resides). The sum of these quantities times B is the global coefficient of the zeroth element of block I_B. Again, relative positions in the global and local blocks are unchanged, so the additional offset $i \bmod B$ completes the back transformation to the global coefficient I.

Cardinality: The cardinality turns out equal to that for the linear distribution. As we scatter blocks, we build layers from low-numbered processes to high-numbered processes. We complete a total of lP whole layers of block coefficients. The last r blocks are scattered one each from process 0 through process $r-1$, creating the same cardinalities as for the block-linear distribution. ■

C.1.3 Generalized Families

Definition C.5 (Generalized Block-Linear) *The generalized block-linear distribution function family (λ) is defined in chapter 4. Here we present a description and derivation of important features.*

Description Global to Local: Global coefficient I resides in global block I_B. Counting from the end, the "reverse block" for I_B is defined as

$$I_B^{\text{rev}} \bmod P \equiv b - 1 - I_B.$$

We count blocks backwards, because we wish to place the coefficient imbalance in the r highest-numbered processes. By doing so, we place the last, possibly foreshortened block as the last block of coefficients in process $P-1$. That is, the "tail regime," $P - r \leq p \leq P - 1$, contains $l+1$ blocks, while the head regime, $0 \leq p \leq P - r - 1$, contains l blocks. This weighs down higher-numbered processes with the load imbalance (*cf.*, block-linear and block-scatter). To determine within which process p

block $I_{\bar{B}}$ resides, we must consequently consider the maximum of two terms:

$$\max\left(\left\lfloor\frac{I_B^{\text{rev}}}{l+1}\right\rfloor, \left\lfloor\frac{I_B^{\text{rev}}-r}{l}\right\rfloor\right). \tag{C.19}$$

Initially, the first term is dominant, with equality for $I_B^{\text{rev}} = (l+1)r$ (equivalently, $I_B = b-1-(l+1)r$). For larger values of I_B^{rev}, the second term dominates. Since we count blocks backwards, this quantity is subtracted from $P-1$ to give p. To get the local coefficient i in process p, we recognize that pl blocks are used up in processes $0,\ldots,p-1$, assuming no load imbalance. Furthermore, starting with process $P-r$, an additional block is used in each process to cover load imbalance. This is reflected in the "Heaviside" term $\Theta^1(p-(P-r))$, which is defined to be zero if its argument is non-positive and equal to its argument otherwise; equivalently, $\max(0, p-(P-r))$. Hence the sum of these two block terms scaled by the blocksize B, $B(pl+\Theta^1(p-(P-r)))$, is subtracted from I to give the local coefficient i. The possible shortness of the last block in process $P-1$ does not impact the calculation.

Local to Global: Since we know i as an explicit function of I and p, it's easy to rearrange the formula to express $I = i + B(pl+\Theta^1(p-(P-r)))$, as asserted.

Cardinality: By inspection, the cardinality function, $\lambda_B^\sharp(p,P,M)$ is equivalent to the following formula:

$$\lambda_B^\sharp(p,P,M) = \begin{cases} B\left\lfloor\frac{b+p}{P}\right\rfloor & \text{for } p < P-1 \\ B\left\lfloor\frac{b+P-1}{P}\right\rfloor & \text{for } p = P-1 \text{ and } M \bmod B = 0 \\ Bl + M \bmod B & \text{for } p = P-1 \text{ and } M \bmod B \neq 0 \end{cases}. \tag{C.20}$$

Initially (for $p=0$), the fraction $\lfloor(b+p)/P\rfloor$ is equal to l. When p reaches $P-r$, however, the fraction increases in value to $l+1$, as required. If the last block in $P-1$ is foreshortened, we account for this explicitly. Fortunately, because of its convenient location at the end, the foreshortened block, if present, does not impact

the calculation of the distribution function and its inverse in any explicit way. ■

Definition C.6 (Generalized Block-Scatter) *The generalized block-scatter distribution function family (σ) is defined in chapter 4. Here we present a description and derivation of important features.*

Description Global to Local: In order to place load imbalance in high-numbered processes, we utilize "back scattering" of coefficients, as compared to the common "forward scattering" used by the $\hat{\sigma}$ family. So, for block coefficient I_B, its mirror block is $I_B^{\text{rev}} = b - 1 - I_B$. We scatter based on this reverse block in order to find its process p, subtracting the scatter quantity $I_B^{\text{rev}} \bmod P$ from $P - 1$ to avoid reversing the order of blocks:

$$p = P - 1 - (I_B^{\text{rev}} \bmod P).$$

Importantly, this formula leaves the last global block of coefficients as the highest block in process $P - 1$. Hence, if $M \bmod B \neq 0$, then the foreshortened block ends up in process $P-1$. Since block layers are filled in top-down in this scheme, computing the local coefficient i also requires more effort. Given the process p, we have $\lfloor I_B^{\text{rev}}/P \rfloor$ complete blocks already in place above it. However, we need the block offset in process p from the beginning in order to figure the local coefficient i. From the cardinality function (the same as for the generalized block-linear function; see also below), we know that the number of block coefficients in p is

$$\left\lfloor \frac{b+p}{P} \right\rfloor, \tag{C.21}$$

and, therefore, the following is the number of unfilled-in blocks:

$$\left\lfloor \frac{b+p}{P} \right\rfloor - \left\lfloor \frac{I_B^{\text{rev}}}{P} \right\rfloor. \tag{C.22}$$

This quantity minus one is the local block index for I_B. Scaling up by the blocksize B gives the local coefficient of the zeroth element of block I_B, and adding the coefficient offset of I within the global block I_B completes the local coefficient i:

$$B\left(\left\lfloor\frac{b+p}{P}\right\rfloor - 1 - \left\lfloor\frac{I_B^{\text{rev}}}{P}\right\rfloor\right) + (I \bmod B). \tag{C.23}$$

Notice that the possibility that $M \bmod B \neq 0$ does not impact this derivation. Only the number of blocks in each process plays a role.

Local to Global: The inverse is derived as follows:

$$\left\lfloor\frac{i}{B}\right\rfloor = \left(\left\lfloor\frac{b+p}{P}\right\rfloor - 1 - \left\lfloor\frac{I_B^{\text{rev}}}{P}\right\rfloor\right), \tag{C.24}$$

$$\left\lfloor\frac{i}{B}\right\rfloor P + P = \left\lfloor\frac{b+p}{P}\right\rfloor P - \left\lfloor\frac{I_B^{\text{rev}}}{P}\right\rfloor P, \tag{C.25}$$

$$= (b+p) - (b+p) \bmod P - I_B^{\text{rev}} + I_B^{\text{rev}} \bmod P, \tag{C.26}$$

$$= p + 1 + I_B - (b+p) \bmod P + I_B^{\text{rev}} \bmod P. \tag{C.27}$$

Rearranging,

$$I_B = \left\lfloor\frac{i}{B}\right\rfloor P + (b+p) \bmod P + (P - 1 - I_B^{\text{rev}} \bmod P - p) \tag{C.28}$$

$$= \left\lfloor\frac{i}{B}\right\rfloor P + (b+p) \bmod P, \tag{C.29}$$

since $p = P - 1 - I_B^{\text{rev}} \bmod P$. Scaling up by the blocksize B, and adding the offset within the block $i \bmod B$, we obtain

$$I = B\left(\left\lfloor\frac{i}{B}\right\rfloor P + (b+p) \bmod P\right) + (i \bmod B), \tag{C.30}$$

as asserted.

Cardinality: The cardinality function $\sigma^\sharp$ is identical to $\lambda^\sharp$, by construction. From back scattering, the global blocks are layed out in reverse order, beginning with the last block and process $P-1$. Each subsequent block is placed in the next lowest-numbered process plus P, modulo P. Consequently, the foreshortened block, if any, is placed as the highest-numbered block in process $P-1$, and each process has at least l block coefficients. By virtue of the scattering order, the last r processes get $l+1$ block coefficients. Except for process $P-1$, these "tail regime" processes receive $B(l+1)$ total coefficients. Process $P-1$ gets $Bl+M \bmod B$ coefficients for $M \bmod B \neq 0$; otherwise it receives as many coefficients as the other processes of the tail regime. Processes in the "head regime" (the first $P-r$ processes) receive exactly Bl coefficients. ■

Definition C.7 (Parametric Families) *The parametric distribution function families (ξ and ζ) are defined in chapter 4. Here we present a description and derivation of important features. The one-parameter family ζ is recovered from ξ for the special case $B=1$, and consequently requires no separate discussion.*

Description Global to Local: The two-parameter distribution ξ incorporates a scattering parameter S as well as the blocking parameter B apparent in the block distribution functions described above and in chapter 4. In analogy to the generalized block-linear distribution, the two-parameter distribution divides the processes into a head and tail regime, again loading the tail regime with imbalance. The compatibility with the generalized block-linear distribution is inherent in the definition of the two-parameter distribution, as we shall see. This distribution also divides the coefficients of each process into two regimes: scattering and non-scattering. BS-size blocks are scattered by the two-parameter distribution.

First, we differentiate each process into scattering and non-scattering regimes. The parameter $l_S = \lfloor l/S \rfloor$ is the number of complete BS-size blocks in each process. Only coefficients will local indices less than l_S are candidates for scattering. We

must ask how we can know the local coefficient of a global coefficient I before the transformation is itself complete. Well, first we compute the generalized block-linear data distribution of I to obtain (p_0, i_0), I's image in a purely linear distribution. For $i_0 \geq l_S$, this is the final mapping for the coefficient; no scattering of it can occur. We quantify this breakpoint by $\Lambda_0 \equiv \lfloor i_0/BS \rfloor$. In the regime $\Lambda_0 \geq l_S$, the two-parameter distribution (and its inverse) are identical to the generalized block-linear distribution, by definition.

So, the interesting case is evidently $\Lambda_0 < l_S$. The total number of BS-blocks in each processes' scattering regime number l_S. In processes $0, \ldots, p_0 - 1$, exactly $p_0 l_S$ BS-blocks are used up. In process p_0, exactly Λ_0 blocks have been used up to reach coefficient I. The global BS-block number of coefficient I is consequently $I_{BS} \equiv p_0 l_S + \Lambda_0$. (Note that this counting completely ignores non-scattering coefficients in each process.) Now that we have the global BS-block number of coefficient I, we can forward scatter this block. We want successively higher blocks to go to successively higher processes, modulo P:

$$p_1 \equiv I_{BS} \bmod P. \tag{C.31}$$

Furthermore, by the time we have scattered I_{BS} BS-blocks, there are $\lfloor I_{BS}/P \rfloor$ complete BS-block layers already in place (below for forward scattering) in each process. This quantity scaled up by the blocksize BS gives the local coefficient number of the zeroth coefficient of the block. We don't change the relative order of coefficients within BS-blocks, so the additional offset is exactly $i_0 \bmod BS$. Therefore, we arrive at the local coefficient for I inside process p_1:

$$i_1 \equiv BS \lfloor I_{BS}/P \rfloor + (i_0 \bmod BS) \tag{C.32}$$

as asserted. The two-parameter distribution is just a composition of generalized block-linear and simple block-scatter distributions applied in a particular way, with

special rules added. For this reason, sufficiently large S causes a degeneration to the generalized block-linear distribution (we quantify when this happens in a bit). For $S = 1$, we are scattering the first l B-block coefficients in each process. This is not exactly how the generalized block-scatter distribution would partition the coefficients, but represents an equivalently useful scattering mechanism.

At what point does scattering cease? Trivially, for $S > l$, $l_S = 0$, and the scattering regime becomes vacuous in each process. A tighter bound is

$$S = \left\lfloor \frac{l}{2} \right\rfloor + 1, \tag{C.33}$$

for which there are a total of P BS-blocks, divided one each between the processes. The simple forward-scattering function doesn't scatter any of these blocks. So, we define $S_{crit} \equiv \lfloor l/2 \rfloor + 1$, the point at which all scattering ceases in the two-parameter distribution.

Local to Global: Our first goal is to see if the coefficient i is in the scattering or non-scattering regime. Simply, $\Lambda \equiv \lfloor i/BS \rfloor$ can be compared with l_S, as we compared $\Lambda_0 \equiv \lfloor i_0/BS \rfloor$ to l_S above. If $\Lambda \geq l_S$, we are in the non-scattering regime, and the global coefficient is given by the inverse generalized block-linear distribution $I = \lambda_B^{-1}(p, i, P, M)$.

Consequently, the interesting case again occurs in the scattering regime; namely, $\Lambda < l_S$. From p, i, we must reconstruct the global BS-block number of coefficient I, I_{BS}. From that, we can work back to the coefficients (p_0, i_0) formed by the original block-linear distribution and

$$(p_0, i_0) \overset{\lambda_B^{-1}}{\mapsto} I. \tag{C.34}$$

By definition of i in the scattering regime,

$$\Lambda \equiv \left\lfloor \frac{i}{BS} \right\rfloor = \left\lfloor \frac{I_{BS}}{P} \right\rfloor, \tag{C.35}$$

$$\Lambda P \;=\; I_{BS} - I_{BS} \bmod P \;=\; I_{BS} - p, \tag{C.36}$$

and, rearranging,

$$I_{BS} = \Lambda P + p, \tag{C.37}$$

which we denoted by I^*_{BS} in chapter 4, for clarity. Having recovered I_{BS}, the original process p_0 is simply $\lfloor I_{BS}/l_S \rfloor$. The original local index i_0 is the sum of $BS(I_{BS} \bmod l_S)$ and the BS-block offset $i \bmod BS$. The pair (p_0, i_0) can be back-transformed (as for coefficients in the non-scattering regime) via the generalized block-linear inverse function λ_B^{-1} to give I. (We called this coefficient pair (p_2, i_2) when stating the inverse in chapter 4, for clarity.) This completes the construction of the inverse.

Cardinality: The cardinality function is the same as for the generalized block-linear distribution λ_B. This is true because we implicitly rely on λ_B to generate the first step of the distribution. Then, the scattering effect merely rearranges BS-blocks between processes without changing the static load balance. ■

C.2 Selected Proofs

We constructively prove the correctness of the scatter and block-scatter data distributions. To prove correctness, we construct the inverse, we demonstrate the consistency of the cardinality function (*i.e.*, that it sums to exactly M coefficients), and that the distribution functions are one-to-one and onto.

Identity C.1 (Division with Remainder) *The following trivial identity holds for integers x, y, and is applied (often implicitly) throughout the derivations, descriptions and proofs:*

$$x = \left\lfloor \frac{x}{y} \right\rfloor y + x \bmod y \quad \forall\, y \neq 0. \tag{C.38}$$

Lemma C.1 (Coefficient Cardinality) *The linear, scatter, block-linear distribution and block-scatter distributions' cardinality functions defined above are correct, provided that* $M \bmod B = 0$.

Proof The total number of coefficients is

$$\tilde{M} \equiv \sum_{p=0}^{P-1} \hat{\lambda}_B^{\sharp}(p,P,M) = B \sum_{p=0}^{P-1} \left\lfloor \frac{b+P-p-1}{P} \right\rfloor \tag{C.39}$$

$$= B\left(r(l+1)+(P-r)l\right) \tag{C.40}$$

$$= B(Pl+r) \equiv B\left(P\left\lfloor \frac{b}{P} \right\rfloor + b \bmod P\right) \tag{C.41}$$

$$\equiv Bb \equiv B\left\lfloor \frac{M}{B} \right\rfloor = M - M \bmod B, \tag{C.42}$$

applying the definitions of l, r and b. Since $M \bmod B = 0$ by assertion, the total of coefficients is $\tilde{M} = M$, as required. The case $B = 1$ proves the correctness for the linear and scatter cardinality functions (which are the same). ■

Lemma C.2 (Scatter, Block-Scatter Correctness) *The block-scatter and scatter distribution functions are correct assuming* $M \bmod B = 0$. *The scatter-case proof is recovered in the following by setting* $B = 1$.

Proof First we construct the inverse function. By definition of i in Equation C.16,

$$\left\lfloor \frac{i}{B} \right\rfloor = \left\lfloor \frac{I_B}{P} \right\rfloor. \tag{C.43}$$

Therefore by Equation C.38,

$$\left\lfloor \frac{i}{B} \right\rfloor P = I_B - (I_B \bmod P), \tag{C.44}$$

$$\left\lfloor \frac{i}{B} \right\rfloor P + p = I_B, \tag{C.45}$$

$$B\left(\left\lfloor \frac{i}{B} \right\rfloor P + p\right) = BI_B = I - (I \bmod B), \tag{C.46}$$

$$= I - (i \bmod B), \tag{C.47}$$

and using the definition $p = I_B \bmod P$ from Equation C.16. Therefore,

$$I = B\left(\left\lfloor \frac{i}{B} \right\rfloor P + p\right) + (i \bmod B) \tag{C.48}$$

as asserted. Inserting $B = 1$ in the above arguments, we derive the scatter distribution inverse.

Next, given any (p, i) such that $0 \leq p < P$ and $0 \leq i < \hat{\lambda}_B^\sharp(p, P, M)$, we demonstrate that its pre-image I satisfies $0 \leq I \leq M-1$. By definition, $I \geq 0$ trivially and, furthermore:

$$I \leq \tilde{I} \equiv B\left(\left\lfloor \frac{\hat{\sigma}_B^\sharp(p, P, M) - 1}{B} \right\rfloor P + p\right) + ((\hat{\sigma}_B^\sharp(p, P, M) - 1) \bmod B) \tag{C.49}$$

Letting $s = \hat{\sigma}_B^\sharp(p, P, M)$,

$$\begin{aligned}
\tilde{I} &= (s - 1 - (s-1) \bmod P)P + Bp + (s-1) \bmod B \\
&= (s - 1 - (B-1))P + Bp + (B-1) \;=\; sP + B(p + 1 - P) - 1 \\
&= B\left(\left\lfloor \frac{b + P - p - 1}{P} \right\rfloor P\right) + B(p + 1 - P) - 1 \\
&= B(b + P - p - 1) + B(p + 1 - P) - 1 - B((b + P - p - 1) \bmod P) \\
&= (Bb - 1) - B((b + P - p - 1) \bmod P) \\
&= (M - 1) - B((b + P - p - 1) \bmod P) \;\leq\; M - 1
\end{aligned} \tag{C.50}$$

since

$$(\hat{\sigma}_B^\sharp(p, P, M) - 1) \bmod B \;=\; B - 1 \tag{C.51}$$

and since $Bb = M$ and $(b + P - p - 1) \bmod P \geq 0$, $I \leq \tilde{I} \leq M - 1$. By construction of the distribution function, the image of the pre-image I is (p, i), which satisfies the requirements on p and i above.

Next, for any two coefficients I^1, I^2 such that $0 \leq I^1, I^2 \leq M - 1$ with $I^1 \neq I^2$, their respective images (p^1, i^1), (p^2, i^2) are not identical (shows one-to-oneness). We assume $p^1 = p^2$ and $i^1 = i^2$, and demonstrate a contradiction. The contradiction arises trivially by applying the definition of the inverse distribution function (a nonsingular transformation of the distribution function) to the pairs (p^1, i^1) and (p^2, i^2), implying that $I^1 = I^2$. Therefore, we have a contradiction, and hence the images of I^1 and I^2 cannot be identical.

Finally, we have to show that the process of scattering actually produces the cardinalities we defined. This is easy to see. Starting with $p = 0$, blocks are scattered to successively higher processes, modulo P. This places at least l blocks in each process, and exactly $l + 1$ blocks in the first r processes. Scaling up by the blocksize B, we immediately see that each process has the appropriate number of coefficients alloted to it. ■

C.3 Weak Data Distributions

As stated at the outset of this appendix, and elsewhere in this thesis, it is important to consider the implications of data distributions that are not of a closed form, but which still require (substantially) less than $O(M)$ space and time complexity for M coefficients. To accomplish this general goal, we weaken the information content of data distribution functions in each process, per the following definition, and consider the implications on correctness of such weakened distributions. Weak data distributions offer the potential for much more control over the locality of data in the multicomputer ensemble, and are consequently necessary in general simulation programs. See chapter 6 for the connection of weak data distributions to concurrent

dynamic simulation problems in chemical engineering.

Definition C.8 (Weak Data Distribution) *A weak data distribution function, inverse function, and coefficient cardinality are defined local to each process* $\tilde{p}$, $0 \leq \tilde{p} < P$, *as follows:*

$$\omega(I,P,M) \mapsto \begin{cases} \tilde{\Omega}(I,P,M) \equiv (p,i) & \text{if } p = \tilde{p} \\ (-1,-1) & \text{otherwise} \end{cases}, \tag{C.52}$$

$$\omega^{-1}(p,i,P,M) \mapsto \begin{cases} \tilde{\Omega}^{-1}(p,i,P,M) \equiv I & \text{if } p = \tilde{p} \\ -1 & \text{otherwise} \end{cases}, \tag{C.53}$$

$$\omega^{\sharp}(p,P,M) \equiv \begin{cases} \tilde{\Omega}^{\sharp}(p,P,M) & \text{if } p = \tilde{p} \\ -1 & \text{otherwise} \end{cases}, \tag{C.54}$$

where $\tilde{\Omega}$, $\tilde{\Omega}^{-1}$ *and* $\tilde{\Omega}^{\sharp}$ *are respectively the corresponding hypothetical strong distribution, inverse and cardinality functions that provide our desired mapping, and which we could implement with* $O(M)$ *memory complexity or* $O(\lceil \log_2 P \rceil)$ *time complexity.*

We can readily construct a weak distribution ω that require $O(\lceil log_2 \lceil M/P \rceil \rceil)$ time complexity and $O(\lceil M/P \rceil)$ space complexity. The local coefficients of process p are sorted alongside their global counterparts, using the global coefficients as the ascending sort key. Computing $\omega(I,P,M)$ consequently requires a binary-search lookup. (Perfect hashing functions, if feasible, reduce access time to $O(1)$.) We can also construct a weak inverse ω^{-1} requiring $O(1)$ time complexity, and $O(\lceil M/P \rceil)$ space complexity. We store the global coefficients corresponding to local coefficients in local-coefficient order. Then, the local index becomes a perfect hash for the weak inverse. Finally, the weak cardinality function $\omega^{\sharp}$ can be stored with $O(1)$ memory in each process, and evidently requires $O(1)$ time to evaluate.

Corollary C.1 (Relationship to Strong Data Distributions) *Strong data distributions are weak data distributions.*

Proof For example, the strong data distributions cited in this thesis require $O(1)$ time and $O(1)$ storage. Since they have the entire mapping and inverse mappings available, they have the locally required mappings available as a subset. The claim is consequently true for these distributions. It's also true more generally (this is not a deep result). ■

Lemma C.3 ("Strengthening" Distributions) *Any weak data distribution can be made into a strong data distribution (with $O(\lceil \log_2 P \rceil)$ time complexity) by introducing a combine operation among the P participating processes in the data distribution, inverse and coefficient cardinality functions.*

Proof Appropriate associative-commutative combination operations can be defined for use with *combine* for each of the three weak functions to be "strengthened." Each of these combination operations selects the appropriate result from its candidate pairs, always rejecting a "-1 datum," which represents a lack of information. In this way, it's easy to see that *combine* globalizes the local information of each of the three weak functions. ■

In principle, this lemma offers a useful connection of weak distributions to strong distributions. However, the cost of *combines* makes the overhead for mappings defined in this way unacceptably high in practice. Evidently, we must use memory proportional to $O(M)$ in each process, or accept the cost of "globalizing" information at each instance, and of synchronizing all participants as well. However, there are perhaps circumstances where intermediate amounts of memory (that we can bound) can be used, with concommitant bounds on the amount of communication required. We consider this idea next.

The construction of hashing functions for weak data distributions is another area of possible research. Perfect hash functions for $\omega(I, P, M)$ would remove the need for a binary search to discover the local component of the mapping $I \mapsto (p, i)$ in process p, or its absence. This is a topic for future consideration.

Definition C.9 (Sequentially Accessed Weak Distributions) *Assume a weak data distribution family ω partitions M coefficients among P processes. Further, assume that the order of accesses in transforming global coefficients K to local coefficients (p, k) by $\omega(K, P, M)$ follows the natural order $K = 0, \ldots, M-1$ from start to end (and/or $K = M-1, \ldots, 0$). Then, we can define a strong distribution Ω based on ω that requires $O(\lceil 2M/\tilde{B} \rceil)$ storage in each process, and that has average time complexity*

$$O\left(\left\lceil \frac{\lceil \log_2 P \rceil}{\tilde{B}} \right\rceil\right). \tag{C.55}$$

$\tilde{B}$ is denoted the coefficient window (or blocking) parameter.

Proof We construct the mechanism supposed above by means of the generalized block-linear data distribution function λ_B and the *broadcast* primitive. Each process p $(0 \leq p < P)$ reserves "window storage" $\mathcal{W}$ of length (in appropriate integer-length units) $2 \max_{\tilde{p}} \lambda^{\sharp}_{\tilde{B}}(\tilde{p}, P, M)$, and a current window number W_N (set initially to -1). Furthermore, each process p, $0 \leq p < P$, reserves an additional "local storage" area $\mathcal{W}_p$ of length $2\lambda^{\sharp}_{\tilde{B}}(p, P, M)$ (in appropriate integer-length units). Then, in the kth location of the local storage $\mathcal{W}_p$ in process p, we store

$$\begin{aligned} \mathcal{W}_p[k] \equiv\ & \tilde{\Omega}(\lambda^{-1}_{\tilde{B}}(p, k, P, M), P, M), \\ & k = 0, \ldots, \lambda^{\sharp}_{\tilde{B}}(p, P, M) - 1, \\ & p = 0, \ldots, P-1. \end{aligned} \tag{C.56}$$

$\tilde{\Omega}$ is the hypothetical strong data distribution function accomplishing the desired global-local mapping. As constructed by Equation C.56, the first $\tilde{B}$ coefficient pairs are stored once and for all to block $\mathcal{W}_0$, the next $\tilde{B}$ coefficient pairs go to block $\mathcal{W}_1$, and so forth, with the last (possibly foreshortened) block of coefficient pairs stored to $\mathcal{W}_{P-1}$. By construction, the data for global coefficient K is stored in $\mathcal{W}_p[k]$, where $(p,k) \equiv \lambda_{\tilde{B}}(K,P,M)$.

Now, when a global coefficient K is to be transformed, we form $(p,k) \equiv \lambda_{\tilde{B}}(K,P,M)$. If $W_N \neq p$, a broadcast of the local storage $\mathcal{W}_p$ is effected among the P processes. This communication step replaces the local storage window $\mathcal{W}$ in each process with $\mathcal{W}_p$; subsequently, we set $W_N = p$. Since $W_N = p$ by construction, the local window storage $\mathcal{W}$ in each process already contains the desired local process number and offset of coefficient K at location $\mathcal{W}[k]$. This construction completes the strong distribution Ω. (The inverse distribution and cardinality functions remain weak.) $\tilde{\Omega}$, which is the hypothetical, "expensive" distribution function, does not figure again after the one-time construction of the local storage areas (presumably effected when a program is initialized).

Clearly, if K is accessed in ascending or descending order, then the broadcasts will only occur once in every $\tilde{B}$ calls to the transformation $\Omega(K,P,M)$, yielding the average time complexity claimed. However, the distribution is correct regardless of the access-order for K, but more random access generates higher communication overheads, just as observed in previous discussion further above. ■

Windowing mechanisms such as this prove useful for pivoting strategies in LU factorization. See appendix D.

Appendix D

Details on Concurrent Sparse Linear Algebra

Abstract

We present abstract and practical LU factorization and triangular-solve algorithms in abridged UNITY notation (*cf.*, appendix G). At first, we do not address the implications of sparsity explicitly. Hence, the first few algorithms pertain to both dense and sparse concurrent linear algebra. Implicitly, this discussion shows how the reduced-communication pivoting techniques apply to dense and sparse linear algebra algorithms alike, a fact we have not stressed heavily in the main text; all dense solvers utilizing partial row and/or column pivoting, or preset pivoting strategies should be modified to use this approach and thereby reduce communication requirements, with immediate non-trivial performance improvement. In any event, sparse matrices are our key interest at present, and we know that the efficiency of sparse linear algebra algorithms cannot be divorced from the details of sparsity. Hence, we eventually specialize our discussion here to sparsity issues. For brevity, at this latter stage of the appendix we describe the algorithm more anecdotally, referring to small sections of UNITY code, and explaining their transformation and/or specialization to practical sparse-matrix code.

D.1 Basic Algorithms

The basic LU factorization and triangular solve algorithms shown in Figures D.1., D.2, and D.3. are well known (see, for example, [22,21]). Furthermore, dense concurrent algorithms have been presented by Van de Velde at various times (*e.g.*, [57,56]). Here, we skip the formal transformation steps between abstract and practical and pass immediately to our improved specification for LU factorization (Figures D.4, D.5), an algorithm that implicitly includes the reduced communication pivoting feature, per chapter 5. Our presentation of the triangular solves in Figures D.6, D.7 appear quite similar to that of Van de Velde, though the specialization to sparse matrix structures will eventually reveal differences in our approach. Sparsity issues are described in the next section.

We choose a data distribution

$$\mathcal{G} \equiv \left(\left\{(\mu, \mu^{-1}, \mu^{\sharp}); P, N\right\}, \left\{(\nu, \nu^{-1}, \nu^{\sharp}); Q, N\right\}\right),$$

and partition the matrix A accordingly; index sets $\mathcal{I}^p$, $\mathcal{J}^q$, $p = 0, \ldots, P-1$, $q = 0, \ldots, Q-1$ form the partition of $\mathcal{I}$, $\mathcal{J}$, respectively. For brevity, in the UNITY notation we use

$$m^p \equiv \mu^{\sharp}(p, P, N), \quad n^q \equiv \nu^{\sharp}(q, Q, N).$$

In the practical algorithms, process superscripts on variables (*e.g.*, $a_{piv}^{p,q}$, $\mathcal{I}^{p,q}$) are suppressed whenever variables are guaranteed to hold the same value in all unnoted processes (respectively, a_{piv}, $\mathcal{I}^p$) as a result of loose synchronizing procedures (*i.e.*, global communication inherent in a pivoting strategy, row/column *broadcasts*) or implicit replication (*e.g.*, row/column vectors). Variables p, q, which determine process context, are never superscripted. These ellisions help implicitly to reinforce the notion

of data evolution as the calculation progresses in the ensemble.

Interprocess communication steps are, of course, introduced to share quantities that were "globally available" in the abstract procedures. For the LU factorization, this results in pivot-row and multiplier-column broadcasts, and, in general, implies at least some global communication within the pivot strategy. The grid formulation uses vector replication: column vectors are process-row distributed and process-column replicated while row vectors are process-column distributed and process-row replicated (see chapter 2). In Algorithms *LU–2* (see Figures D.4, D.5), *FWD–2* (Figure D.6), and *BCK–2* (Figure D.7), we have also specified global communication operations distinctly from [57]. Specifically, we recognize that row/column broadcasts can be accomplished by logarithmic *fanouts* rather than by a linear transmission procedure envisaged in the foregoing reference and related publications, and as implemented in that author's experimental linear algebra codes. We illustrate the structure and performance of our *Zipcode*-based *broadcast* operation in chapters 2, 3, respectively. Algorithm *LU–2*, the "practical" LU Factorization, uses implicit pivoting and an arbitrary, user-defined pivoting strategy symbolized by pivot_fn(). Delaying the test for numerical singularity is part of the strategy that permits reduced communication pivoting, while slightly slowing down termination of the worst case scenario, a numerically rank-deficient matrix.

Figure D.1. Algorithm *LU–1*

```
procedure LU–1  {LU Factorization of A = [a[i,j]] ∈ ℜ^{N×N}}
declare
    i, j, k, rank      :   integer
    r_k, c_k           :   integer
    I, J               :   set of integer
    a_piv              :   real
    a                  :   array [0..N − 1, 0..N − 1] of real
initially
    rank = 0       {numerical rank initially zero}
    I    = {i : 0 ≤ i < N}   {All rows and}
    J    = {j : 0 ≤ j < N}     {columns active initially}
assign
    ⟨; k : 0 ≤ k < N ::
        {Acquire pivot for kth step: acquire pivot value and row, column: r_k, c_k}
        {(Pivot storage mechanism is a side-effect of pivot_fn()):}
        a_piv, r_k, c_k ← pivot_fn(k, a, M, N, I, J);

        {Deactivate pivot row (holds r_kth row of U) and
             column (will hold c_kth column of L):}
        I := I \ {r_k};  J := J \ {c_k};

        {Calculation of the Multiplier Column (c_kth column of L): }
        if a_piv ≠ 0.0 then begin ⟨ || i : i ∈ I :: a[i, c_k] := a[i, c_k]/a_piv ⟩ ; end

        {Belated test for numerical singularity:}
        if a_piv = 0.0 then terminate else rank := k + 1;

        {Effect the elimination:}
        ⟨|| i, j : i ∈ I and j ∈ J ::
             if a[r_k, j] ≠ 0.0 and a[i, c_k] ≠ 0.0 then
                  a[i, j] := a[i, j] − a[r_k, j]a[i, c_k];
        ⟩
    ⟩
end LU–1
```

Abstract representation of LU Factorization with implicit pivoting and arbitrary pivoting strategy symbolized by pivot_fn(). Delaying the test for numerical singularity is significant to the concurrent implementation.

Figure D.2. Algorithm *TRI–1*

procedure *TRI–1* {Given L, U factors, right-hand-side b, solve $Ax = b$}
declare
x : array $[0..N-1]$ of real
b : array $[0..N-1]$ of real
See also Figure D.1 ...
initially
$\mathcal{I} = \{i : 0 \leq i < N\}$
assign
{Column-Oriented, In-place Forward-Substitution overwrites b ($Ly = b$):}
$\langle ; k : 0 \leq k < N ::$
{Recover Pivot Indices from storage mechanism:}
$r_k, c_k \leftarrow$ pivot_idx_load($k, \mathcal{I}, \mathcal{J}$);
$\mathcal{I} := \mathcal{I} \setminus \{r_k\}$; {Deactivate r_kth row of L}
{By now, $b[r_k]$ contains kth Forward-Substitution solution element.}
$\langle \| i : i \in \mathcal{I} ::$
if $b[r_k] \neq 0.0$ **and** $a[i, c_k] \neq 0.0$ **then**
$b[i] := b[i] - a[i, c_k] b[r_k]$;
$\rangle$
$\rangle$

{Column-Oriented, Back-Substitution: $\left(\mathsf{P}_C^T \mathsf{P}_R\right) Ux = y \equiv b$:}
$\mathcal{I} := \{i : 0 \leq i < N\}$; {Re-initialize index set}
$\langle ; k : \leftarrow N - 1 \geq k \geq 0 ::$
{Recover Pivot Indices from storage mechanism:}
$r_k, c_k \leftarrow$ pivot_idx_load($k, \mathcal{I}, \mathcal{J}$);
$\mathcal{I} := \mathcal{I} \setminus \{r_k\}$; {Deactivate r_kth row of U}
{Divide by Pivot Element: $a[r_k, c_k]$ to finalize $x[c_k]$:}
$x[c_k] := b[r_k]/a[r_k, c_k]$;
{Now, $x[c_k]$ contains kth Backward-Substitution solution element.}
$\langle \| i : i \in \mathcal{I} ::$
if $b[r_k] \neq 0.0$ **and** $a[i, c_k] \neq 0.0$ **then**
$b[i] := b[i] - a[i, c_k] x[c_k]$;
$\rangle$
$\rangle$
end *TRI–1*

Abstract representation of triangular solves compatible with Algorithm *LU–1*, Figure D.1. The (application-defined) pivot-storage mechanism implicit in pivot_fn() provides pivot indices to the triangular solves through the auxiliary function pivot_idx_load().

Figure D.3. Algorithm *LU–1a*

```
procedure LU–1a  {LU Factorization of A = [a[i,j]] ∈ ℜ^{N×N}; forward elim. of b}
declare
    i, j, k, rank      :   integer
    r_k, c_k           :   integer
    I, J               :   set of integer
    a_piv              :   real
    a                  :   array [0..N − 1, 0..N − 1] of real
    b                  :   array [0..N − 1] of real
initially
    rank = 0       {numerical rank initially zero}
    I    = {i : 0 ≤ i < N}
    J    = {j : 0 ≤ j < N}
assign
    ⟨; k : 0 ≤ k < N ::
        a_piv, r_k, c_k ← pivot_fn(k, a, M, N, I, J);   {Acquire pivot}
        I := I \ {r_k};  J := J \ {c_k};   {Deactive Pivot Row, Column}
        if a_piv ≠ 0.0 then begin ⟨ || i : i ∈ I :: a[i, c_k] := a[i, c_k]/a_piv ⟩ ; end
        if a_piv = 0.0 then terminate else rank := k + 1;
        ⟨|| i, j : i ∈ I and j ∈ J ::   {Effect the elimination}
            if a[r_k, j] ≠ 0.0 and a[i, c_k] ≠ 0.0 then
                a[i, j] := a[i, j] − a[r_k, j]a[i, c_k];
        ⟩
        ⟨|| i : i ∈ I ::   {Forward Substitution}
            if b[r_k] ≠ 0.0 and a[i, c_k] ≠ 0.0 then
                b[i] := b[i] − a[i, c_k]b[r_k] ;
        ⟩
    ⟩
end LU –1a
```

Algorithm *LU–1* augmented with forward substitution on the right-hand-side b.

Figure D.4. Algorithm *LU–2*, "Practical" LU Factorization, Part I.

procedure *LU–2* {LU Factorization of $A = [a[i,j]] \in \Re^{N \times N}$}
constant
$FIRST\_ROW\_FANOUT = 1$
declare
$p, q, rank$: integer
$\hat{p}^{p,q}, \hat{q}^{p,q}, k$: integer
$i^{p,q}, j^{p,q}$: integer
$\mathcal{I}^p, \mathcal{J}^q$: set of integer
$a_{piv}^{p,q}$: real
$a^{p,q}$: array $[0..m^p - 1, 0..n^q - 1]$ of real
l^p : array $[0..m^p - 1]$ of real
u^q : array $[0..n^q - 1]$ of real
$\mathcal{G}$: data_distribution $\equiv \left(\left\{(\mu, \mu^{-1}, \mu^\sharp); P, N\right\}, \left\{(\nu, \nu^{-1}, \nu^\sharp); Q, N\right\}\right)$
initially
$rank = 0$ {numerical rank initially zero}
$\mathcal{I}^p = \{i : 0 \le i < m^p\}$, $\mathcal{J}^q = \{j : 0 \le j < n^q\}$
assign
$\langle\| \; p : 0 \le p < P$ **and** $q : 0 \le q < Q :: $
$\langle \; ; \; k : 0 \le k < N :: $
{Acquire / Store kth pivot info.: get local names for r_k, c_k}
$a_{piv}^{p,q}, \hat{p}^{p,q}, \hat{\imath}^{p,q}, \hat{q}^{p,q}, \hat{\jmath}^{p,q}, mode \leftarrow$ pivot_fn$(k, a^{p,q}, m^p, n^q, \mathcal{I}^p, \mathcal{J}^q; \mathcal{G}, p, q)$;
if $mode = FIRST\_ROW\_FANOUT$ **then begin**
{Case $\hat{p}$ globally known; Process Row $\hat{p}$ has correct $\hat{\imath}, \hat{q}$:}
if $p = \hat{p}$ **then begin**
$\mathcal{I}^{\hat{p}} := \mathcal{I}^{\hat{p}} \setminus \{\hat{\imath}\}$; {Deactivate Pivot Row (r_kth row of U)}
if $q = \hat{q}$ **then** $\mathcal{J}^{\hat{q}} := \mathcal{J}^{\hat{q}} \setminus \{\hat{\jmath}\}$; {Deactivate before Broadcast}
end
{Broadcast Pivot Row (plus, in-principle Pivot Value, $\hat{q}, \hat{\jmath}^q$):}
fanout $a^{\hat{p},q}[\hat{\imath}, j^q : j^q \in \mathcal{J}^q], a_{piv}^{\hat{p},q}, \hat{\imath}, \hat{q}, \hat{\jmath}^q$
on $\mathcal{G} : (\bullet, q) \rightarrow u^q[\bullet], a_{piv}^{\hat{p},q}, \hat{\imath}, \hat{q}, \hat{\jmath}^q$;
{All have correct $\hat{p}, \hat{\imath}, \hat{q}$; Process Column $\hat{q}$ has correct $\hat{\jmath}$.}
if $q = \hat{q}$ **then begin**
if $p \neq \hat{p}$ **then** $\mathcal{J}^{\hat{q}} := \mathcal{J}^{\hat{q}} \setminus \{\hat{\jmath}\}$; {Deactivate Pivot Column}
{Form Multiplier Column (c_kth column of L):}
if $a_{piv}^{\hat{p},\hat{q}} \neq 0.0$ **then begin**
$\langle \; ; \; i^p : i^p \in \mathcal{I}^p :: a^{p,\hat{q}}[i^p, \hat{\jmath}] := a^{p,\hat{q}}[i^p, \hat{\jmath}] / a_{piv}^{\hat{p},\hat{q}} \; \rangle$;
end
end

Figure D.5. Algorithm *LU-2*, "Practical" LU Factorization, Part II.

```
                {Broadcast Multiplier Column, Pivot Value, Promote ĵ:}
                fanout a^{p,q̂}[i^p : i^p ∈ I^p, ĵ], a^{p̂,q̂}_{piv}, ĵ on G : (p,•) → l^p[•], a_{piv}, ĵ;
                if singularity_test(a_{piv}) = TRUE then
                    terminate else rank := k + 1;
            else begin
                {Case q̂ globally known; Process Column q̂ knows p̂, ĵ, Pivot Value:}
                if q = q̂ then begin
                    J^q̂ := J^q̂ \ {ĵ};  {Deactivate Pivot Column}
                    {Form Multiplier Column (c_k th column of L):}
                    if a^{p̂,q̂}_{piv} ≠ 0.0 then begin
                        ⟨ ; i^p : i^p ∈ I^p :: a^{p,q̂}[i^p, ĵ] := a^{p,q̂}[i^p, ĵ]/a^{p̂,q̂}_{piv} ⟩ ;
                    end
                    if p = p̂ then I^p̂ := I^p̂ \ {î};  {Deactivate before Broadcast}
                end
                {Broadcast Multiplier Column, Correct Pivot Value, Promote Indices:}
                fanout a^{p,q̂}[i^p : i^p ∈ I^p, ĵ], a^{p̂,q̂}_{piv}, p̂, î^p, ĵ on G : (p,•) → l^p[•], a_{piv}, p̂, î^p, ĵ;
                {Now, all have correct p̂, q̂, ĵ; a_{piv}. Process Row p̂ has correct î.}
                if singularity_test(a_{piv}) = TRUE then
                    terminate else rank := k + 1;
                if p = p̂ and q ≠ q̂ then
                    I^p̂ := I^p̂ \ {î};  {Deactivate Pivot Row (r_k th row of U)}
                {Broadcast the Pivot Row , Promote Index î:}
                fanout a^{p̂,q}[î, j^q : j^q ∈ J^q], î on G : (•,q) → u^q[•], î;
            end
            pivot_index_store(k, p̂, î, q̂, ĵ; G, p, q);  {Store Pivot Indices}
            ⟨ ; i^{p,q} : i^{p,q} ∈ I^p ::  {Effect local eliminations}
                if l^p[i^{p,q}] ≠ 0.0 then begin
                    ⟨ ; j^{p,q} : j^{p,q} ∈ J^q ::
                        a^{p,q}[i^{p,q}, j^{p,q}] := a^{p,q}[i^{p,q}, j^{p,q}] − l^p[i^{p,q}] u^q[j^{p,q}]
                    ⟩
                end
            ⟩
        ⟩ {End of k-quantification}
    ⟩_G  {End of p, q-quantification}
end LU-2
```

Figure D.6. Algorithm *FWD–2*, "Practical" Forward-Solve

procedure *FWD–2* {Column-Oriented, Forward-Solve overwrites b ($Ly = b$):}
declare
b^p : array $[0..m^p - 1]$ of real
$v^{p,q}$: real
See also Figure D.4. ...
initially
$\mathcal{I}^p = \{i : 0 \le i < m^p\}$
assign
$\langle\!\|\ p : 0 \le p < P$ **and** $q : 0 \le q < Q$ ::
{Recover 0th Pivot Indices from storage mechanism:}
$\hat{p}, \hat{\imath}, \hat{q}, \hat{\jmath} \leftarrow$ pivot_idx_load$(0; \mathcal{G}, p, q)$;
$\langle ;\ k : 0 \le k < N$ ::
{Recover $k+$ 1st Pivot Indices from storage mechanism:}
if $k < N - 1$ **then**
$\hat{p}_+, \hat{\imath}_+, \hat{q}_+, \hat{\jmath}_+ \leftarrow$ pivot_idx_load$(k+1; \mathcal{G}, p, q)$;
if $p = \hat{p}$ **then** $\mathcal{I}^{\hat{p}} := \mathcal{I}^{\hat{p}} \setminus \{\hat{\imath}\}$; {Deactivate r_kth row of L}
if $q = \hat{q}$ **then begin**
{$b^{\hat{p},\hat{q}}[\hat{\imath}]$ contains kth Forward-Substitution solution element.}
if $p = \hat{p}$ **then** $v^{\hat{p},\hat{q}} := b^{\hat{p},\hat{q}}[\hat{\imath}]$;
fanout $v^{\hat{p},\hat{q}}$ **on** $\mathcal{G} : (\bullet, \hat{q}) \rightarrow v^{\hat{q}}$;
if $v^{\hat{q}} \neq 0.0$ **then begin**
$\langle\ ;\ i^p : i^p \in \mathcal{I}^p$ ::
$b^{p,\hat{q}}[i] := b^{p,\hat{q}} - a^{p,\hat{q}}[i, \hat{\jmath}] v^{\hat{q}}$;
$\rangle$
end
{Retransmission step for $\hat{q}_+ \neq \hat{q}$:}
if $k < N - 1$ **and** $\hat{q}_+ \neq \hat{q}$ **then send** $b^{p,\hat{q}}[\bullet]$ **to** $\mathcal{G} : (p, \hat{q}_+)$;
end
if $k < N - 1$ **and** $q = \hat{q}_+$ **and** $\hat{q} \neq \hat{q}_+$ **then**
receive $b^{p,\hat{q}}[\bullet]$ **from** $\mathcal{G} : (p, \hat{q}) \rightarrow b^{p,\hat{q}_+}[\bullet]$;
end
$\hat{p} := \hat{p}_+$; $\hat{\imath} := \hat{\imath}_+$; $\hat{q} := \hat{q}_+$; $\hat{\jmath} := \hat{\jmath}_+$;
$\rangle$
$\rangle$
end *FWD–2*

Practical representation of triangular solves compatible with Algorithm *LU–2*, Figures D.4, D.5. The (application-defined) pivot-storage mechanism implicit in pivot_idx_load() retrieves pivot indices.

Figure D.7. Algorithm *BCK–2*, "Practical" Back-Solve Algorithm

procedure *BCK–2* {Column-Oriented Back-Solve: $\left(\mathsf{P}_C^T \mathsf{P}_R\right) Ux = y \equiv b$:}
declare
x^q : array $[0..n^q - 1]$ of real
b^p : array $[0..m^p - 1]$ of real
See also Figures D.4, D.6. ...
initially
$\mathcal{I}^p = \{i : 0 \leq i < m^p\}$
assign
$\langle\!\|\ p : 0 \leq p < P$ **and** $q : 0 \leq q < Q ::$
{Recover $N - 1$st Pivot Indices from storage mechanism:}
$\hat{p}, \hat{\imath}, \hat{q}, \hat{\jmath} \leftarrow$ pivot_idx_load$(N - 1; \mathcal{G}, p, q)$;
$\langle ;\ k : \leftarrow N - 1 \geq k \geq 0 ::$
{Recover $k - 1$st Pivot Indices from storage mechanism:}
if $k > 0$ **then** $\hat{p}_-, \hat{\imath}_-, \hat{q}_-, \hat{\jmath}_- \leftarrow$ pivot_idx_load$(k - 1; \mathcal{G}, p, q)$;
if $p = \hat{p}$ **then** $\mathcal{I}^{\hat{p}} := \mathcal{I}^{\hat{p}} \setminus \{\hat{\imath}\}$; {Deactivate r_kth row of U}
if $q = \hat{q}$ **then begin**
if $p = \hat{p}$ **then** $x^{\hat{p},\hat{q}}[\hat{\jmath}] := b^{\hat{p},\hat{q}}[\hat{\imath}] / a^{\hat{p},\hat{q}}[\hat{\imath},\hat{\jmath}]$;
fanout $x^{\hat{p},\hat{q}}[\hat{\jmath}]$ **on** $\mathcal{G} : (\bullet, \hat{q}) \rightarrow x^{\hat{q}}[\hat{\jmath}]$;
if $x^{\hat{q}}[\hat{\jmath}] \neq 0.0$ **then begin**
$\langle\ ;\ i^p : i^p \in \mathcal{I}^p ::$
$b^{p,\hat{q}}[i] := b^{p,\hat{q}} - a^{p,\hat{q}}[i,\hat{\jmath}] x^{\hat{q}}[\hat{\jmath}]$;
$\rangle$
end
{Retransmission step for $\hat{q}_- \neq \hat{q}$:}
if $k > 0$ **and** $\hat{q}_- \neq \hat{q}$ **then send** $b^{p,\hat{q}}[\bullet]$ **to** $\mathcal{G} : (p, \hat{q}_-)$;
end
if $k > 0$ **and** $q = \hat{q}_-$ **and** $\hat{q} \neq \hat{q}_-$ **then**
receive $b^{p,\hat{q}}[\bullet]$ **from** $\mathcal{G} : (p, \hat{q}) \rightarrow b^{p,\hat{q}_-}[\bullet]$;
end
$\hat{p} := \hat{p}_-$; $\hat{\imath} := \hat{\imath}_-$; $\hat{q} := \hat{q}_-$; $\hat{\jmath} := \hat{\jmath}_-$;
$\rangle$
$\rangle$
end *BCK–2*

Practical representation of the back-solve compatible with Algorithm *LU–2*, Figures D.4, D.5. The (application-defined) pivot-storage mechanism implicit in pivot_idx_load() retrieves pivot indices.

D.2 Sparsity Issues

The basic structure of these algorithm specifications carries over to the sparse case, though with some significant, practical changes. In this section, we mention issues in the progression from algorithms *LU–2*, *FWD–2*, and *BCK–2*, to their fully sparse forms. *LU–2* will split into the A- and B-modes introduced in chapter 5. We comment upon the means for implementing indexation that avoid quadratic work. We also clarify some of the comments made in chapter 5. Though we enlargen here on the discussion in chapter 5, a much more complete discussion of (sequential) unsymmetric sparse matrix linear algebra is offered by Duff, *et al.* [16].

D.2.1 Indexation

Index sets must be handled carefully in order to avoid "$O(N^2)$ traps." For example, an index set uses memory proportional to the local problem size, say l, where $l \sim \lceil N/L \rceil$, with N the global problem size, and L the number of processes (either P or Q). Therefore, work of $O(l)$ cannot be done within the outer k-loop of the factorization or triangular solves without implying a trap. Innocuous $O(l)$ work occurs if, for example, clearing an index set dutifully zeroes the entire memory area of length $O(l)$. In short, we require that testing, activating or deactivating an index require $O(1)$ work, and that clearing, or any a systematic access to an index set, be done in time proportional to the number of active entries. So, in loops that consider the active entries of an index set, this should not be a loop over l with tests for active/inactive status of elements. Creating an appropriate, efficient indexation mechanism turns out to be a straightforward excercise, once the need is realized. We comment on this fact because another multicompuer sparse solver of which we are aware, the prototype sparse solver considered in [53], incorporates index set manipulations that violate these requirements, and consequently, its LU factorization has a sequential complexity of at least $O(N^2)$.

D.2.2 LU Factorization

We consider means to convert Algorithm *LU–2* into appropriate sparse forms. There is a significant amount of machinery in the actual concurrent code that we do not consider here; our present purpose is to convey the key notions. We know that the naive access to the matrix elements $a^{p,q}[i,j]$ will have to be refined in order to exploit sparsity, in addition to circumspect index manipulation. This means that linked lists will have to be maintained; we must consider the style of linked lists to be used. Since we will not store the entire $O(N^2)$ elements, fill-in will require manipulation of the linked-list structure. There are many such issues, and it boots little to quote them out of context. In short, we elect a two-mode factorization, as detailed in the main text. In A-mode, we construct the linked-lists dynamically based on (hopefully stable) pivots selected by the user-defined pivot function. In B-mode, we resolve to re-use the same pivotal sequence and linked-list structure, and monitor stability. Then, we can flag instability when necessary. A-mode is repeated as often as necessary, in practice. This strategy reflects the successful style of Harwell's *MA28* code [16].

UNITY Sparse Notation

Up to now, we have considered matrix manipulations within the dense arrays $[a^{p,q}[i,j]]$ in the "practical" algorithms. In the sparse algorithm, we will not have this data structure. Instead, we will manipulate linked lists so the sparse notational prototype

$$\langle\ ;\ \ell\ :\ \ell \in \bigcirc : \{\mathsf{L},\ \searrow,\ \searrow\searrow \ldots\}\ ::\ \ldots\rangle$$

shall be a quantification with sequential traversal by variable ℓ. Instances of ℓ are the objects referred to by L, and links to L in turn, with linkages specified by operation $\bigcirc$, up to but not including the termination of the list $\emptyset$. For matrix elements, the operations $\bigcirc$ will be *NEXT_ROW* for row-oriented linked-list accesses, and *NEXT_COL*

for column-oriented accesses. The objects referred to by ℓ are, for the present discussion, matrix elements of the type specified in Figure 5.2. $\ell\searrow\mathsf{T}$ shall refer to the component of the object ℓ named T, so that $\ell\searrow i$ is the local row index i of the sparse matrix entry referenced by ℓ.

Stability Test

We choose a very simple stability test, as suggested by our experience and that of colleagues [4] with Harwell's *MA28* , this is sufficient for many practical circumstances. Naturally, the implementations can easily be changed to incorporate more involved tests, should this prove necessary in unforseen applications.

The stability test is as follows (*cf.*, [58]). We compute the growth factor γ

$$\gamma \equiv \frac{\max_{0\le i<N,\ 0\le j<N} a[i,j]^{(N-1)}}{\max_{0\le i<N,\ 0\le j<N} a[i,j]} \tag{D.1}$$

where by $a[i,j]^{(N-1)}$ we mean the values in the L and U factors upon completion of the factorization. If γ is large, then instability is evidently possible. However, it is not our goal to judge the quality of the pivoting algorithm chosen by the user, but rather to reveal the degradation of stability caused by repeated use of a pivotal sequence on similar but numerically unequal matrices. If γ_A is defined as the growth factor for a matrix factored with a user-specified pivoting function, we can consider the ratio

$$\gamma_{AB} \equiv \frac{\gamma_B}{\gamma_A} \tag{D.2}$$

where γ_B is the growth factor computed for a structurally similar matrix, but factored with the same pivotal sequence used for the initial matrix. The quotient γ_{AB} indicates the relative loss of stability resulting from the fixed pivotal sequence. A user tolerance (*e.g.*, 10) can be set to reflect the allowable size of γ_{AB} before instability is declared.

Even before an initial factorization, the entire sparse matrix is to be stored in

linked-lists of row pointers $U_{row}[i]$, $i = 0, \ldots, m^p - 1$, and $L_{row}[i]$, $i = 0, \ldots, m^p - 1$, some of which may be trivial ($\emptyset$). (By A-mode convention (below), nothing is to be stored initially in L_{row} pointers, but the following is still valid.) Then, both before and after a factorization, a valid computation for the growth factor γ is achieved through the use of the procedure in Figure D.8. once before and once after a factorization. The ratio of the initial and final values of c thus attained equals γ.

Figure D.8. Computation of Growth Factor γ

```
declare
    c^{p,q}                :    real,
    L^{p,q}_{row}[i]       :    array [0..m^p − 1] of linked_list of matrix_entry pointers
    U^{p,q}_{row}[i]       :    array [0..m^p − 1] of linked_list of matrix_entry pointers
initially
    c^{p,q} = 0;
assign
    ⟨|| p : 0 ≤ p < P and q : 0 ≤ q < Q ::
        ⟨; i^p : 0 ≤ i^p < m^p ::
            if L^{p,q}_{row}[i^p] ≠ ∅ then begin
                ⟨ ; ℓ : ℓ ∈ NEXT_ROW : {L^{p,q}_{row}[i^p], ↘, ↘↘ ...} ::
                    c^{p,q} := max(c^{p,q}, ℓ ↘ Value);
                ⟩
            end
            if U^{p,q}_{row}[i^p] ≠ ∅ then begin
                ⟨ ; ℓ : ℓ ∈ NEXT_ROW : {U^{p,q}_{row}[i^p], ↘, ↘↘ ...} ::
                    c^{p,q} := max(c^{p,q}, ℓ ↘ Value);
                ⟩;
            end
        ⟩;
    ⟩;
    combine[max] c^{p,q} on G : (•, •) → c;
end
```

D.2.3 A-mode

A-mode begins with a computation of the c-value described above, and likewise, ends with a matching computation, yielding γ_A. We demand that the entire A matrix be pre-stored in row-oriented linked-lists, $U^{p,q}_{row}[i^p]$, $i^p = 0, \ldots, m^p - 1$, $p = 0, \ldots, P - 1$,

$q = 0, \ldots, Q-1$. At completion, the $U^{p,q}_{row}[i^p]$, $L^{p,q}_{row}[i^p]$ contain pointers to the rows of U and L, respectively, within which additional entries have been added to incorporate fill-in. Furthermore, $U^{p,q}_{col}[j^q]$ and $L^{p,q}_{col}[j^q]$, $j^q = 0, \ldots, n^q - 1$ are column-wise pointers (constructed at the conclusion of A-mode) that reference the same matrix structure.

We do not presort the entries of the sparse matrix prior to factorization. Nor do we sort the linked-lists for L and U during or after the factorization, for example, into elimination sort order. However, when a pivot element is located in a process, say in $(\hat{p}, \hat{q})$ at location $(\hat{\imath}, \hat{\jmath})$, it's placed at the beginning of its $U^{\hat{p},\hat{q}}_{row}[\hat{\imath}]$ and $U^{\hat{p},\hat{q}}_{col}[\hat{\jmath}]$ pointer lists, for convenient access in the triangular solves.

D.2.4 B-mode

B-mode is a simplification of A-mode. First of all, the pivotal strategy is always preset pivoting; the preset sequence is exactly the sequence created dynamically by A-mode. As a consequence of the construction of linked-lists in A-mode, a new sparse matrix provided to B-mode must have all the "fill-in" set to zero, and be stored within the L/U linked-list pointers created by A-mode for the initially factored matrix. The L/U pointers hold entries that will eventually become part of L and U, respectively. However, it is convenient to segregate the operations over these linked-list pointers from the beginning of B-mode. In the actual implementation, the elimination phase contains a loop over U pointers and L pointers. Counter intuitively, the elements touched by any elimination phase works on exactly those elements not yet part of the triangular factors.

The style of linked-list traversal during the local elimination phase has an important effect on performance. When eliminating, we can first utilize the activity information given by either $\mathcal{I}^p$'s or $\mathcal{J}^q$'s to restrict effort, depending on whether we eliminate across rows or down columns. Inside any active row or column linked-list, however, we are obliged to touch each element at every iteration k, even if this is

just to test for activity of a column (resp., row) element. Were we to sort the linked-lists in elimination order, we could avoid touching elements known to be inactive and hence reduce work further. Presently, we eliminate anachronistically as follows: the U linked-list (representing entries eventually part of U) are traversed row-wise, and the L linked-list (representing entries eventually part of L) are traversed column-wise. Optionally, the user could have control over these choices, but A-mode linked-list sorting is a better solution.

Finally, we compute γ_B analogously to the computation of γ_A in A-mode. At the end of the computation, γ_{AB} is formed, and checked against a user tolerance (γ_{AB}^{max}) for this ratio of growth factors. A warning flag is raised if the ratio proves too large.

D.2.5 Triangular Solves

The triangular solves are quite similar to the *FWD–2* and *BCK–2* with the exception that column-oriented linked-lists are utilized. Each element in L and U are touched exactly once. The calculation complexity of these operations is consequently proportional to the number of entries (plus fill-in) contained in the respective factors. As noted in chapter 5, the retransmission step can constitute higher-order work than the calculation itself (since we have to transmit whole local b^p vectors horizontally), if Q is too large, and/or too much scattering of matrix column data is inherent in the ν distribution. For $Q = 2$, this higher-order work can be avoided easily, but, in general, the use of an appropriate column data distribution is essential.[1]

For example, the inner quantification

if $x^{\hat{q}}[\hat{\jmath}] \neq 0.0$ **then begin**
$\langle\ ;\ i^p\ :\ i^p \in \mathcal{I}^p\ ::$

[1]We note *en passant* that this component of the linear algebra calculation illustrates how the scaled performance paradigm mentioned in appendix F can lead one astray. Since communication cost grows faster than computation cost for typical grid configurations and data distributions, scaling the problem to larger matrix sizes for such configurations makes performance worse.

$b^{p,\hat{q}}[i] := b^{p,\hat{q}} - a^{p,\hat{q}}[i,\hat{\jmath}]x^{\hat{q}}[\hat{\jmath}];$

$\rangle$

end

from Algorithm *BCK–2* is replaced by:

if $x^{\hat{q}}[\hat{\jmath}] \neq 0.0$ **then begin**

$\langle\ ;\ \ell^p : \ell^p \in NEXT\_ROW : \{U_{col}^{p,q}[\hat{\jmath}],\ \searrow,\ \searrow\searrow \ldots\}\ ::$

$b^{p,\hat{q}}[i] := b^{p,\hat{q}} - \ell^p \searrow \text{Value} \cdot x^{\hat{q}}[\hat{\jmath}];$

$\rangle$

end

Appendix E

Details on Waveform Relaxation

Abstract

This appendix has one purpose: to reveal the underlying differential-algebraic problem formulation inherent in *CONCISE* and comment upon it. Here we build on recent discussions with and suggestions of Mattisson and Söderlind [34]. This appendix is included mainly to spur future investigation rather than for conclusiveness. Such investigations must include a study of convergence results for DAE systems and applicability to chemical process flowsheeting, our key interest.

E.1 *CONCISE*'s Problem Formulation

In circuit simulation, the form of the differential-algebraic equation to be solved is

$$I(v) + \frac{dQ(v)}{dt} = 0, \tag{E.1}$$

with I the static current, Q the charge, and dQ/dt the dynamic current. Let's divorce the discussion from electronics, and consider that we want to solve the following implicit ODE system:

$$\frac{dq(v)}{dt} = -f(v). \tag{E.2}$$

By introducing a dummy variable w, we can reformulate the Equation E.2 as follows:

$$\left.\begin{aligned} \tfrac{dw}{dt} &= -f(v), \\ 0 &= w - q(v), \end{aligned}\right\} \tag{E.3}$$

which is a semi-explicit DAE system. Equations E.3 are solved for v with error control on v; w is included for notational convenience and is never evaluated.

Referring back to electronics, we make the connections

$$\left.\begin{aligned} I(v) &\equiv f(v), \\ Q(v) &\equiv q(v), \end{aligned}\right\} \tag{E.4}$$

with v bearing the same connotation as before. By comparison, for a typical flowsheeting applications (distillation simulation), we do not exploit the general nonlinear mapping $q(v)$, and so choose $q(v) \equiv v$; thus, Equations E.3 become for this case:

$$\left.\begin{aligned} \tfrac{dw}{dt} &= -f(v), \\ 0 &= w - v. \end{aligned}\right\} \tag{E.5}$$

In *CONCISE*'s electrical model library, we note that $Q(v) = Cv$ for a linear capacitor, with C the capacitance, and that $Q(v) = q_0 * \sqrt{v - v_0}$ for a diode with a particular doping profile. Analogous mappings are defined for other circuit devices. Typically, there are explicit ways to compute Q as a function of v, but the inverse is not necessarily easy to derive. Thus, in the electrical simulation scenario, $f(v)$ is seen to be the static current flowing through conductors and controlled sources (transconductors – used to model transistors and other devices with gain), $q(v)$ is the static charge stored in capacitors and pn-junctions, and $dq(v)/dt$ is the dynamic current flowing through the charge-storing devices.

To solve the equation system, *CONCISE* device model routines return $f(v)$ and

$q(v)$ (electrically, $I(v)$ and $Q(v)$). The differentiation of $q(v)$ (to get dw/dt) is done by the BDF-routines (one differentiation) and

$$\frac{dw}{dt} + f(v) = 0 \tag{E.6}$$

is solved for v by means of Newton-Raphson iterations (for each time point in a waveform). The system is evidently index 1 provided that $\partial q/\partial v$ is non-singular. There is no error control on w. For the BDF-solution, the history of w is needed (electrically, the companion source). Error estimation in w should be a straightforward extension of the BDF routines as defined in [32] and implemented in *CONCISE*, but is thought to be unnecessary for this DAE structure.

As noted above, there are two formulations: Equation E.2., and Equations E.3. The difference between the above two formulations is that Equation E.3. is an implicit ODE (and thus not really an DAE as long as we have a non-singular capacitance matrix dq/dv), while Equation E.3. is a differential-algebraic system of index 1 under the same assumption. If the capacitance matrix is structurally singular or drops in rank, then Equation E.2. is of course also a DAE, and it is important to observe that the index for Equation E.3 always is 1 + the index for Equation E.2. The extra help variable w thus gives an artificial index raise that is not desirable from a numerical point-of-view. Thus, the two formulations are mathematically but not numerically equivalent. By introducing error control on w, Equation E.3. essentially becomes numerically equivalent to Equation E.2. as well, specifically for the case where Equation E.2 is of index 0 (and when solved with a BDF method). While for the electrical simulation the current formulation is acceptable, the index-raising feature is a problem for the typical index 1, 2 systems we wish to simulate in chemical engineering flowsheeting. A reformulation of the *CONCISE* integration mechanism is therefore required to broaden its applicability and avoid bad results because of an

artificial index problem.

Appendix F

On Concurrent Performance

Abstract

In this appendix, we review and comment upon alternative measures of concurrent performance, specifically the "scaled performance" measures advanced by Gustafson, Benner and Montry [25,23] with recent extensions [24].[1] In recent review texts, (*e.g.*, [11]), these measures have been quoted without sufficient critical analysis as to their practical implications and limitations. Our purpose is to add perspective on the validity of such measures, to indicate some of the limitations of their usefulness, and also to suggest what aspects of their definitions are semantic in nature. By presenting this alternative reaction to "scaled measures," our intention is to spur further critical discussion, not to discount the contributions made by the abovementioned authors to the advancement of concurrent supercomputing, even insofar as one strictly considers the "fixed-size" (traditional) speedups attained in Gustafson *et al.* in [25] for interesting large-scale problems.

This appendix, like the others, is limited in scope for the sake of brevity; a more detailed presentation of our views on performance is reserved for a future article devoted specifically to this topic. In passing, we recommend an interesting article, "Speedup Versus Efficiency in Parallel Systems" published recently by Eager *et al.* [18]. It introduces the idea of "average parallelism," and centers on the notion of the

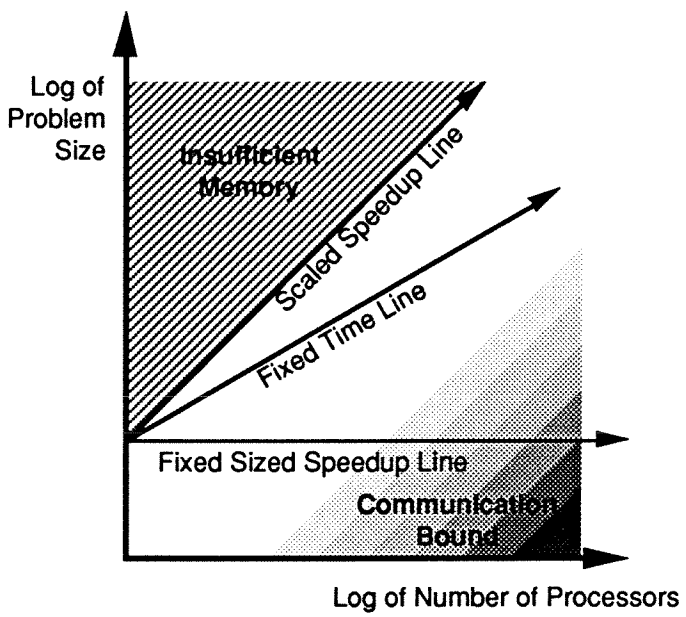
[1] John Gustafson kindly provided machine-readable copies of his performance diagrams for this appendix.

trade-off between performance and utilization of nodes (efficiency). Lower bounds for speedups are offered as a function of average parallelism, and the article makes for interesting reading.

F.1 Scaled Performance Definitions

In this section we offer basic definitions.

Figure F.1. The Scaled Performance Diagram



Definition F.1 (Scaled Speedup) *Given a fixed algorithm $\mathcal{A}$ and problem $\mathcal{P}(p)$, where the problem size is "scaled" up with increasing p (*i.e., *as large as local memories permit), then a single processor of equivalent power (and sufficiently large storage, with uniformly efficient memory access time as compared to ensemble nodes) would ideally require time*

$$T_{serial} + T_{||} \times p \tag{F.1}$$

to execute the same task. Therefore, we define the scaled speedup as

$$S_p^{scaled} = \frac{T_{serial} + T_{||} \times p}{T_{serial} + T_{||}} \tag{F.2}$$

$$\equiv \frac{T_{serial}}{T} + \frac{T_{||}}{T} \times p \tag{F.3}$$

$$\equiv p + (1-p) \times \frac{T_{serial}}{T} \tag{F.4}$$

$$\equiv p + (1-p) \times \tilde{\alpha}, \tag{F.5}$$

where $\tilde{\alpha} \equiv T_{serial}/T$ is identified as the sequential fraction of the computation, assumed independent of p (that is, independent of both problem size and ensemble size).

The standard scaled performance diagram illustrating the idea of scaled speedup is presented in Figure F.1. The fixed-time line presented in the figure represents a measure of performance improvement when the runtime $T(p)$ is to be held constant as the problem size and ensemble size are both increased.

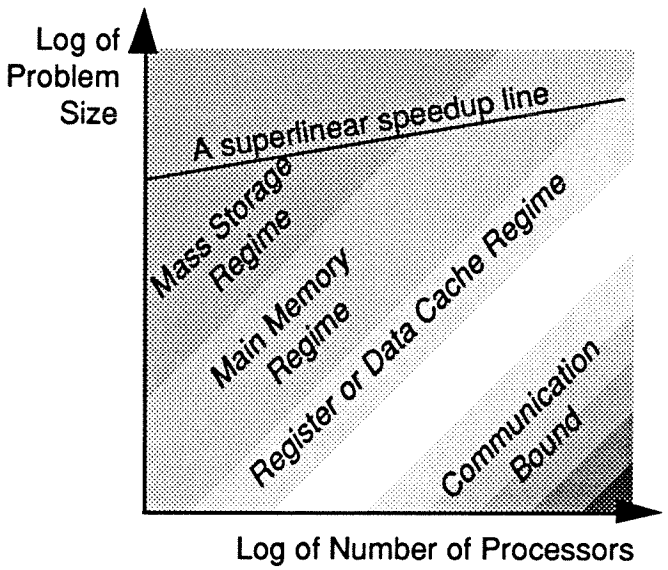
More recently, Gustafson [24] has advanced the idea that speedup is a ratio of speeds rather than times:

Definition F.2 (Velocity Ratio)

$$V - Speedup \equiv \frac{\left(\frac{Sequential\ Work}{Sequential\ Time}\right)}{\left(\frac{Parallel\ Work}{Parallel\ Time}\right)}. \tag{F.6}$$

He presents extensions in the "insufficient memory" regime that describe the increasing cost of problem solution on a single processor resulting from the need to use tiers of memory (successively "farther-away," slower memory, then virtual memory, and so forth). Based on this and other arguments, he contends that superlinear speedup is readily possible within a framework in which one assumes a fixed-time model, increasing both ensemble size and problem size under that constraint. See Figure F.2.

Figure F.2. The Scaled Performance Diagram Revisited



F.2 Why "Scaled Speedup" is not our Favorite Performance Measure

The arguments forwarded for scaled performance are indeed extremely inviting because they define the performance of many parallel problems to be excellent. Without quibbling about semantics further, the key problem we foresee with scaled performance measures is primarily that many problems of interest *do not scale much if at all.* All the argumentation offered for these new performance models fail when a problem cannot be scaled homogeneously over some range of problem and ensemble sizes. For instance, VLSI circuit simulations and chemical process flowsheet simulations are modelled by systems of ordinary differential-algebraic equations. There is no way to change the number of equations very much. There is no regular grid to make finer. What of accuracy? We can increase the complexity of modeling equations to some degree, and this is interesting, though limited. It's true that we can demand greater accuracy for a fixed physical model, and hence increase ("scale") effort in this way. Increasing accuracy by one decimal digit does not however increase work by a

factor of ten. Since accuracy is governed by iterations that are usually superlinearly convergent (Newton iterations), work increases only slowly as we demand more accuracy. Furthermore, both the model and model data limit the amount of accuracy we can sensibly demand. Typically, the accuracy required is posed as part of the problem solution and we attain negligible "credit" for this problem domain by providing "excess" accuracy.

We question the validity of the scaled paradigm in that it suggests increasing the time required to solve a problem in order to attain higher scaled speedup. We prefer to decrease the time required to solve an interesting fixed-sized problem as our key motivation for parallel computing. Furthermore, even the fixed-time case is mainly disinteresting, because, again we want faster solutions. We can resolve to solve larger problems – this is certainly a valid goal of concurrent computing. However, once we pick very large (fixed) problems, we will be satisfied to solve them at all, and have little opportunity for (traditional) speedup measurements because of per-processor memory limitations. The success of our concurrent research is then to be judged by our ability to tackle previously out-of-reach problems.

For three-dimensional partial-differential equation problems (not considered in this thesis), it is clear that the fixed-time paradigm is of interest, because we almost always want to solve more accurate (that is, bigger) problems and a concurrent computer ensemble is clearly a route to such higher performance solution. Furthermore, it is easy to achieve "superlinear performance" in the fixed-time paradigm, even if the problem is of fixed size [24]. This is not a panacea, however. It is probable that new concurrent algorithms based on new numerical analysis could still outperform the parallelized sequential algorithms kept/made attractive by scaled and fixed-time performance measures.

In general, speedup is an obsolescent if not a worn-out concept, not only because of the recent redefinitions discussed here. (We don't like the idea of coupling our

view of performance to an arbitrary sequential execution time.) And, for instance, judging performance of a parallel computer based on a detailed model of virtual-memory requirements of an underlying sequential processor is somewhat beside the point [24]. Concurrency research should center on the concurrency issues, not on the sequential issues. The aim of parallel research is that we wish effectively to depart the sequential arena of computation. It is true that a single processor of equivalent power to an ensemble node would require an expensive memory hierarchy to run a "very large" problem. But does this make concurrent algorithm research or our parallel algorithms implicitly better? We are convinced that it does not. These optimistic measures certainly obscure the need to develop intrinsically parallel algorithms in that they yield superlinear performance results almost trivially. And, insofar as performance measures cloud the need for such new algorithms, they are counterproductive independent of their potential merits.

We are convinced that it is more meaningful to compare the execution time on a faster, bigger computer with ensemble performance in order to judge the effectiveness of a parallel algorithm and/or ensemble. This, of course, is a more conservative approach. Practically, one is more likely to choose a Cray for a large problem, and note performance (time required), rather than choosing an arbitrarily bad memory hierarchy and a low-power single processor. It is fair to say, for instance, that an efficient parallel algorithm on a 1000-node machine runs ten times faster than the best algorithm for the same problem run on a brand-X supercomputer. It is also fair to look at the fixed-time model with a supercomputer solution time as the temporal reference. For example, we could ask how much more accurate (bigger) three-dimensional PDE problems can be solved on a 1000-node concurrent system while still requiring the same time as a Cray would require. Since we view these comparisons as fruitful, it is our confirmed intention to compare dynamic simulation performance of chemical process flowsheets in concurrent ensembles against sequential supercomputer perfor-

mance on these problems in the future. In that way, we will be certain to strive for algorithms capable of achieving absolutely useful performance. Price-performance aspects can also be discussed, and are important, but these are topics for future consideration as well.

Are there fair relative measures (ensemble-ensemble comparisons)? Relative measures of benefit (decreased time, or increased problem accuracy, when meaningful) *vs.* resources applied (processors, memory) are fair. Based on ordinary differential-algebraic systems, and other unscalable problem domains, it proves interesting to reflect on the comparative performances (pure times) of various parallel algorithms run on ensembles of different sizes, as suggested by an unnormalized variation of the Concurrency Diagram, Figure 2.1. On the y-axis, we would plot pure times rather than times normalized by an arbitrary, arguably inaccurate, sequential time. We can compare between algorithms and among ensemble sizes. If we insist on a speedup-like measure, then such a measure could be the ratio of the time required on the smallest feasible ensemble size (because of memory limitations) to that larger ensemble affording the shortest time, as follows:

Definition F.3 (Ensemble-Relative Speedup) *Recognizing that storage limitations prohibit single-processor execution for many interesting problems, we still wish to pose a fair, if conservative, measure of concurrent performance rooted only in the concurrent ensemble. Let p_0 be the minimum number of ensemble nodes required to execute a fixed problem $\mathcal{P}$ with algorithm $\mathcal{A}$, and let T_{p_0} be the time required for completion of problem $\mathcal{P}$ with algorithm $\mathcal{A}$. Let $p \geq p_0$ be some larger ensemble size, which requires time T_p for completion. Then, the ensemble-relative speedup is*

$$S_p^{p_0} \equiv \frac{T_{p_0}}{T_p}. \tag{F.7}$$

$S_p^{p_0}$ is a measure of performance that we can actually measure on a given ensem-

ble without resorting to arbitrary or external performance references. Clearly, for small problems, this is identical to the relative speedup (Amdahl-brand) defined in chapter 2.

Undoubtedly, arguments about performance will persist into the indefinite future. However, it is and will remain important to recognize what constitutes high performance for a given problem domain, and to measure that as fairly as possible. Achieving high parallel performance need not be "easy," and we shouldn't intentionally or inadvertently attempt to obscur such possible difficulties.

Appendix G
Abridged UNITY Notation

Following the work of Chandy and Misra [12], and the lead of Van de Velde [53], we embrace an abridged, simplified form of Chandy's UNITY language for the description of concurrent codes.[1] We describe this briefly, without attention to its full power, or formal properties, both of which are beyond our scope here. We extend slightly what's been presented in [53] to reflect the possibility of multiple process grids. Full program notation is defined implicitly with the UNITY programs in Section 2.

Definition G.1 (Quantification) *The notational prototype (called a quantification) is*

$$\langle\ \mathcal{S}\ i\ :\ i \in \mathbf{O}\mathcal{I}\ ::\ \mathsf{E}(i)\ \rangle, \tag{G.1}$$

where $\mathcal{S}$ is denoted a separator; for example "$\|$" for concurrent evaluation of, and ';' to denote sequential evaluation of expressions $\mathsf{E}(i)$. Each expression may itself be a further quantification, or sequences of assignments separated by ';' and / or '$\|$'.

We think of the Quantification G.1 as a shorthand for an enumeration of all the expressions $\mathsf{E}(i)$, delimited by separator $\mathcal{S}$. A set ordering operator $\mathbf{O}$ replaces the natural (implicitly defined) ordering of index set $\mathcal{I}$, if present.

For example, we define $\mathbf{O} \equiv \leftarrow$ when we wish to indicate reversal of the natural

[1] This is becoming the descriptive concurrency notation of choice at Caltech, and we wish to promulgate this standard, rather than inventing new, colloquial "pidgin" notations from time to time.

ordering of evaluation:

$$\langle\, \mathcal{S}\, i \;:\; i \in \leftarrow\!\mathcal{I} \;::\; \mathsf{E}(i) \,\rangle.$$

Sequential evaluation of expressions $\mathsf{E}(i)$ often looks like:

$$\langle\, ;\, i \;:\; i_1 \leq i \leq i_2 \;::\; \mathsf{E}(i) \,\rangle.$$

For sequential evaluation, the set ordering is important because "old" values of variables are updated in turn by assignments as the evaluation of expressions proceeds. Consequently, if the expressions are dependent, correctness depends on the ordering. The concurrent evaluation of the same expressions:

$$\langle\, \|\, i \;:\; i_1 \leq i \leq i_2 \;::\; \mathsf{E}(i) \,\rangle$$

has the same meaning if the expressions are independent, although set ordering has no effect in any event for the concurrent quantification. With the concurrent separator, we militate that all expressions use the "old" values of variables at evaluation. Consequently, if the expressions are dependent, the two separators are not trivially interchangeable.

A key feature of UNITY is the ability to transform successively from very abstract program designs to very practical, realizable designs (while preserving program correctness). When we pass to the degree of specificity where process grids are defined, the placement of data and evaluations becomes, of course, important. Unambiguous representation is complicated slightly when there are multiple process grids. We remove such ambiguity by subscripting quantifiers with appropriate data distributions, as defined in chapter 4, such as:

$$
\begin{array}{l}
\langle \| \ p \ : \ 0 \leq p < P_{\mathcal{G}} \ :: \\
\qquad \langle \| \ i \ : \ i \in \mathcal{I}^p_{\mathcal{G}} \ :: \ b[i] \ := \ 0 \ \rangle \\
\rangle_{\mathcal{G}}
\end{array}
\quad \| \quad
\begin{array}{l}
\langle \| \ p \ : \ 0 \leq p < P_{\mathcal{G}} \ :: \\
\qquad \langle \| \ i \ : \ i \in \mathcal{I}^p_{\mathcal{G}} \ :: \ c[i] \ := \ 0 \ \rangle \\
\rangle_{\mathcal{G}}
\end{array}
$$

Bibliography

[1] Gita Alaghband. *Multiprocessor Sparse LU Decomposition with Controlled Fill-in.* PhD thesis, University of Colorado, Boulder, 1986. Department of Computer and Electrical Engineering.

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

[3] Eugene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Spring Joint Computer Conference*, pages 483–485. AFIPS Press, 1967. AFIPS Conf. Proceedings vol. 30.

[4] Henrik Weisberg Andersen and Lionel Frederic Laroche, 1988–1990. — Private Communications on *Chemsim.*

[5] Steven Ashby, 1990. — Private Communication on *Iterative DASSL.*

[6] William C. Athas and Charles L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, pages 9–24, August 1988.

[7] David Bailey, Eric Barszcz, Rod Fatooki, Horst Simon, Sisira Weeratunga, Victor Jackson, and Gary Withers. Performance of the DARPA Touchstone Gamma System Prototype Parallel Supercomputer. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, page in press. IEEE Press, April 1990. Charleston.

[8] R. K. Brayton et al. A new efficient algorithm for solving differential-algebraic systems using implicit backward differentiation formulas. *Proceedings of IEEE*, 60(1):98–108, January 1972.

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

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

[11] Graham F. Carey, editor. *Parallel Supercomputing: Methods, Algorithms and Applications.* Wiley, 1989.

[12] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.

[13] T. S. Chua and P. M. Dew. The design of a variable-step integrator for simulation of gas transmission networks. *Int. J. for Num. Meth. Eng.*, 20:1797–1813, 1984.

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

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

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

[17] D. Dumlugöl. *Segmented Waveform Relaxation Algorithms for Large Scale Circuit Simulation.* PhD thesis, Katholieke Universiteit Leuven, 1986. Dept. of Elektrotechniek.

[18] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Trans. on Comp.*, 38(3):408–423, March 1989.

[19] H. P. Flatt. A simple model for parallel processing. *IEEE Computer*, page 95, November 1984.

[20] Geoffrey C. Fox, Mark A. Johnson Gregory A. Lyzenga, Steve W. Otto John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, 1988.

[21] Joel N. Franklin. *Matrix Theory.* Prentice Hall, 1968.

[22] G. H. Golub and C. F. Van Loan. *Matrix Computations.* John Hopkins University Press, second edition, 1989.

[23] John L. Gustafson. Re-Evaluating Amdahl's Law. *CACM*, 11(5):532–533, May 1988.

[24] John L. Gustafson. Fixed Time, Tiered Memory, and Superlinear Speedup. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, page in press. IEEE Press, April 1990.

[25] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of Parallel Methods for a 1024-Processor Hypercube. *Siam J. Scientific and Stat. Comp.*, 9(4):609–638, July 1988.

[26] J. J. Guy. Computation of unsteady gas flow in pipe networks. *I. Chem. E. Symposium Series No. 23*, 1967.

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

[28] C. D. Holland and A. I. Liapis. *Computer Methods for Solving Dynamic Separation Problems.* McGraw Hill, 1983.

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

[30] Ekachai Lelarasmee et al. The waveform relaxation method for time-domain analysis of large scale integrated circuits. *IEEE Trans. on CAD of Int. Circ. and Sys.*, CAD-1(3):131–145, July 1982.

[31] Dardo Marqués. *On-line Optimization of Large Dynamic Systems.* PhD thesis, University of Wisconsin, Madison, 1985. Chemical Engineering Department.

[32] Sven Mattisson. *CONCISE: A Concurrent Circuit Simulation Program.* PhD thesis, Lund Institute of Technology, Sweden, 1986. Department of Applied Electronics.

[33] Sven Mattisson, 1990. — Private Communications on Sequent Symmetry *Reactive Kernel* implementation.

[34] Sven Mattisson and Lena Peterson, 1990. — Private Communications on multicast message transmission and applications.

[35] Ulla Miekkala. Dynamic Iteration Methods Applied to Linear DAE Systems. Technical Report A252 (revised), Helsinki University of Technology, 1988. Institute for Mathematics.

[36] Ulla Miekkala and Olavi Nevanlinna. Convergence of Dynamic Iteration Methods for Initial Value Problems. *Siam J. Scientific and Stat. Comp.*, 8(4):459–482, July 1987.

[37] Lena Peterson. A Study of Convergence-Enhancing Techniques for Concurrent Waveform Relaxation, May 1989. *Teknologie Licenciat* Degree Thesis, Lund University, Dept. of Applied Electronics.

[38] L. R. Petzold. DASSL: Differential Algebraic System Solver. Technical Report Category #D2A2, Sandia National Laboratories — Livermore, 1983.

[39] R. A. Saleh. Parallel Waveform-Newton Algorithms for Circuit Simulation. In *IEEE Proceedings of ISCAS '87*, pages 660–663, July 1983.

[40] Charles L. Seitz. The Cosmic Cube. *CACM*, 28(1):22–33, January 1985.

[41] Charles L. Seitz et al. The C Programmer's Abbreviated Guide to Multicomputer Programming. Technical Report Caltech-CS-TR-88-1, California Institute of Technology, January 1988.

[42] Charles L. Seitz, Sven Mattisson, William C. Athas, Charles M. Flaig, Alain J. Martin, Jakov Seizovic, Craig M. Steele, and Wen-King Su. The architecture and programming of the ametek series 2010 multicomputer. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications (HCCA3)*, pages 33–36. ACM Press, January 1988. (Symult s2010 Machine).

[43] Jakov Seizovic. The Reactive Kernel. Technical Report Caltech-CS-TR-88-10, California Institute of Technology, 1988.

[44] M. G. Singh and A. Titli. *Systems Decomposition, Optimization and Control.* Pergamon, 1978.

[45] Stig Skelboe. Stability Properties of Linear Multirate Formulas. Technical report, University of Copenhagen, April 1986. Institute of Datalogy.

[46] Stig Skelboe. Stability Properties of Implicit Multirate Formulas. In *Proc. European Conf. on Circuit Theory and Design (ECCTD '87)*, volume 2, pages 795–806, September 1987.

[47] Anthony Skjellum, Manfred Morari, and Sven 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.

[48] Anthony Skjellum, Manfred Morari, Sven Mattisson, and Lena 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.

[49] Anthony Skjellum, Lena Peterson, Sven Mattisson, and Manfred Morari. Application of Multicomputers to Large-Scale Dynamic Simulation in Chemical and Electrical Engineering: *Unifying Themes, Software Tools, Progress.* IFIP 11th World Conference — San Francisco; Paper #253, August 1989.

[50] Sigurd Skogestad. *Studies of Robust Control of Distillation Columns.* PhD thesis, California Institute of Technology, 1987. Chemical Engineering.

[51] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, 1987.

[52] Wen-King Su. *Reactive-Process Programming and Distributed Discrete-Event Simulation.* PhD thesis, California Institute of Technology, 1989. Computer Science; Caltech-CS-TR-89-11.

[53] Eric F. Van de Velde. A Concurrent Direct Solver for Sparse Unstructured Systems. Technical Report C^3P Report #604, California Institute of Technology, March 1988. Caltech Concurrent Computation Project.

[54] Eric F. Van de Velde. Data Redistribution and Concurrency. Caltech Applied Mathematics, May 1988.

[55] Eric F. Van de Velde. Implementation of Linear Algebra Operations on Multicomputers. Caltech Applied Mathematics, October 1988.

[56] Eric F. Van de Velde. The Formal Correctness of an LU-Decomposition Algorithm. Technical Report C^3P Report #625, California Institute of Technology, June 1988. Caltech Concurrent Computation Project.

[57] Eric F. Van de Velde. Experiments with Multicomputer LU-decomposition. *Concurrency: Practice and Experience*, 2(1):1–26, March 1990.

[58] Eric F. Van de Velde and Jens 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.

[59] Stefan Vandewalle. Parallel Waveform Relaxation Methods for Solving Parabolic Partial Differential Equations. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5).* IEEE *in press*, April 1990.

[60] Stefan Vandewalle and Dirk Roose. The Parallel Waveform Relaxation Multigrid Method. In *Proceedings of the Third SIAM Conf. on Parallel Process. for Sci. Comp.*, December 1987.

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

[62] Jacob White and A. L. Sangionvanni-Vincentelli. Partitioning Algorithms and Parallel Implementations of Waveform Relaxation Algorithms for Circuit Simulation. In *IEEE Proceedings of ISCAS '85*, pages 221–224, July 1985.