

The Multicomputer Toolbox:
*Scalable Parallel Libraries for Large-Scale Concurrent Applications**

Anthony Skjellum[†]
Mississippi State University
Department of Computer Science
PO Drawer CS
Mississippi State, MS 39762

Chuck Baldwin
Numerical Mathematics Group
Lawrence Livermore National Laboratory (LLNL)

Original Version: December 10, 1991
Revised: December 6, 1994

Report #: UCRL-JC-109251, MSU: CS-TR-941206

*Work performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48. Last revision 6-Dec-94.

[†]To whom correspondence should be addressed. FAX: (601)325-8997. Electronic Mail: tony@cs.msstate.edu.

The Multicomputer Toolbox:
Scalable Parallel Libraries for Large-Scale Concurrent Applications

Abstract

In this paper, we consider what is required to develop parallel algorithms for engineering applications on message-passing concurrent computers (multicomputers). At Caltech, the first author studied the concurrent dynamic simulation of distillation column networks [19, 21, 20, 14]. This research was accomplished with attention to portability, high performance and reusability of the underlying algorithms. Emerging from this work are several key results: first, a methodology for explicit parallelization of algorithms and for the evaluation of parallel algorithms in the distributed-memory context; second, a set of portable, reusable numerical algorithms constituting a “*Multicomputer Toolbox*,” suitable for use on both existing and future medium-grain concurrent computers; third, a working prototype simulation system, *Cdyn*, for distillation problems, that can be enhanced (with additional work) to address more complex flowsheeting problems in chemical engineering; fourth, ideas for how to achieve higher performance with *Cdyn*, using iterative methods for the underlying linear algebra [16, 15, 17, 14]. Of these, the chief emphasis in the present paper is the reusable collection of parallel libraries comprising the *Toolbox*. Concurrent algorithms for the solution of dense and sparse linear systems, and ordinary differential-algebraic equations were developed as part of this work. Importantly, we have embedded the notion of data distribution independence — support for algorithmic correctness independent of data locality choices — into the underlying methodology and practical software. This together with carefully designed message passing primitives, and concurrent data structures encapsulating data layout of matrices and vectors, provides an excellent basis for building and then tuning the performance of whole applications, rather than single computational steps.

In our on-going research efforts at the Lawrence Livermore National Laboratory (LLNL), within the Numerical Mathematics Group, we are working to extend the parallel methodology and *Multicomputer Toolbox* to further applications (including “grand challenges”) as well as diverse multicomputer environments. This systematic effort in top-down and down-up concurrent algorithms research and design is driven by application goals on the one hand, and by the competing desires for high performance and portability on the other. We have experience with existing and newer machines, such as the nCUBE/2 6400, the Caltech Intel Delta prototype, and BBN TC2000, in that the *Toolbox* works well on all of these machines. Soon the Thinking Machines CM-5 (running in “MIMD-mode”) will be added to this list. We also note, in passing, that we are progressing with new parallel libraries for optimization, and iterative linear equation solving. Finally, we indicate the huge potential relevance of our efforts to U.S. industry, and discuss plans to distribute and share our methodology and software technology with U.S. industry.

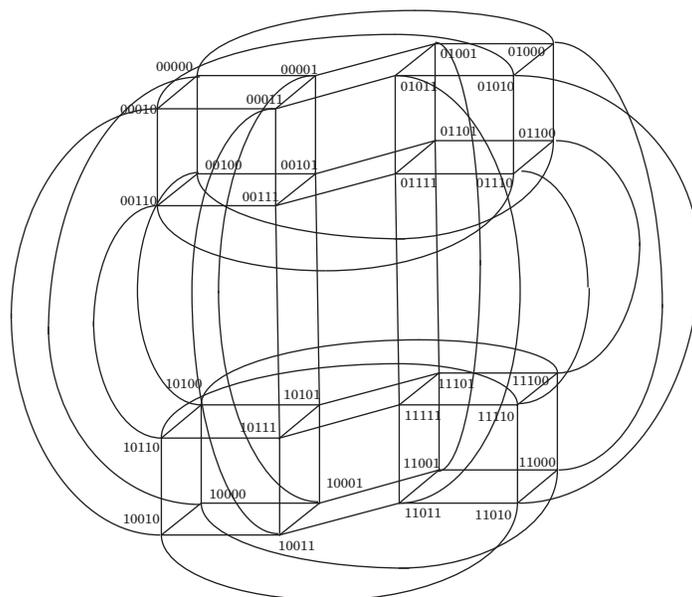
1 Introduction

The primary goal of massively parallel computing is to obtain *orders-of-magnitude* speedup in wallclock execution time by using large, distributed-memory, medium-grain computer ensembles (“parallel processing”) [11, 12, 2, 4, 14]. Contemporary machines bring huge distributed memories, commensurately large overall distributed computation power, and wide aggregate communication bandwidth to bear on a calculation. We are largely unconcerned with the aggregate of CPU cycles used to accomplish high performance on multicomputers (conserving operations is much less important than reducing time to solution). 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, physics and chemistry applications. Present technology offers hardware *scalability* to large ensembles and currently available commercial products incorporate up to roughly one thousand computational nodes. These machines support medium-grain computational tasks, requiring nominally one to six hundred floating-point operations between communications for fifty-percent efficiency. Typical high-end machines offer four to eight gigabytes of memory (distributed evenly among the ensemble nodes), two to six double-precision megaflops per node (scalar floating point) and, optionally, nodal vector and/or pipelined processing capability (optimized matrix-matrix kernels, can exceed thirty megaflops per node, double precision). Node-to-node communication bandwidths continue to hover around ten megabytes per second, though bandwidth is expected to grow and user-level transmission latency is expected to drop significantly over the next two years. A binary n-cube (“hypercube”) (the archetype of massively parallel machines) and a two-dimensional mesh architecture are represented symbolically in Figures 1, 2.

2 Issues and Abstractions

One of the challenges of massively parallel computation is the tremendous growth and change present in the industry, as well as the paucity of software. This effort aims at addressing the software-gap problem, while addressing portability between existing machines of multiple vendors. Furthermore, we seek ways to abstract sources of performance, while hiding sources of confusion and error, in order to allow the libraries developed in the *Toolbox* (and applications based thereon) to survive for several generations of hardware, changing vendors, as needed, to stay with the highest performance machines. We provide tuning “knobs” at a higher level. The *Toolbox* software and its underlying computational methodology address four main issues: the need for an open system (see Figure 4.), the need to support diverse uses of communication resources in complex applications (see Figure 3.), the need to provide vendor-independent software (see Figure 5.), and the need to manage data-locality and transformation in a concurrent application (see Figure 6.).

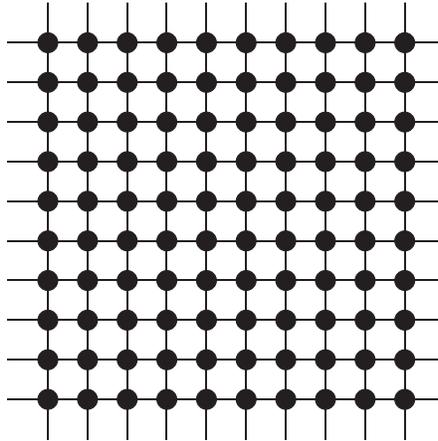
Figure 1. Schematic of a Five-Dimensional Hypercube.



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.

Tacitly, we have chosen a more complex economic objective function by demanding portability, continued usability, and high-level abstractions. We do not code assembly language for each machine, reserving all such optimizations to standard Unix functions, and Basic Linear Algebra Communication Subprograms, all to be vendor provided, and vendor independent. We do expect to get up to speed quickly on new machines, while achieving 50–90% efficiency over time, without having to start over each time or substantially rework software each time a new medium-grain machine is introduced. We are not even willing to accept new vendor calling sequences for familiar operations. As code grows, we must insulate it from such expensive, global changes. Most of the abstractions posed in the *Toolbox* are extremely cheap. Even message-passing layering is not an overtly costly feature. It is also heartening that, if economics demand it, and vendors prove cooperative, all communication primitives offered in the *Toolbox* can equal vendor-specific communication performance. We see the more highly expressive communication calls in the *Toolbox* as opportunities for better performance, rather than lower performance, ultimately, since the application programmer provides more information about the nature and context of the desired operation, and the participants. This facet of the work is leading toward successively more object-oriented

Figure 2. Schematic of a Two-Dimensional Mesh.

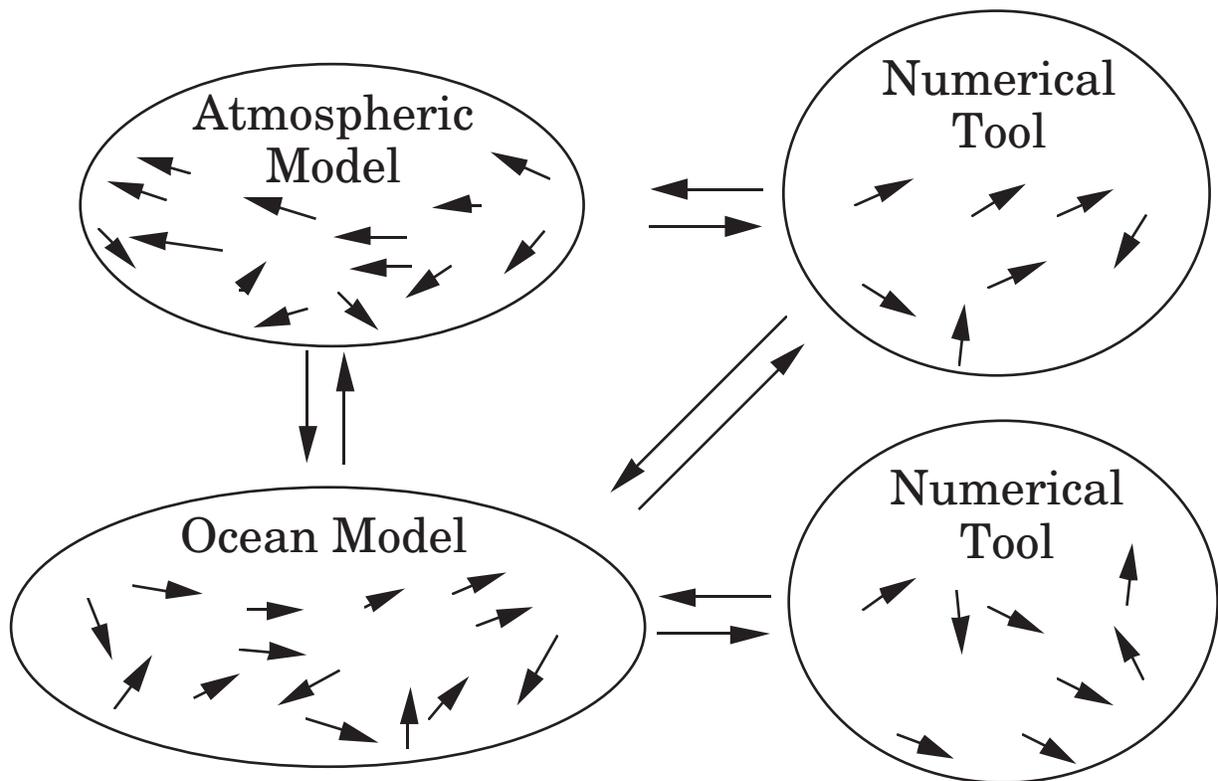


The Symult S2010, Intel Delta and Paragon architectures use a two-dimensional communication network with parallel channels along with an improved routing mechanism for much higher message-passing performance than first-generation multicomputers, as described in [2]. Mesh “edges” provide a rich potential source of input-output bandwidth for visualization, or disk farms.

methods for communication primitives and collections of communicating processes [14, 18].

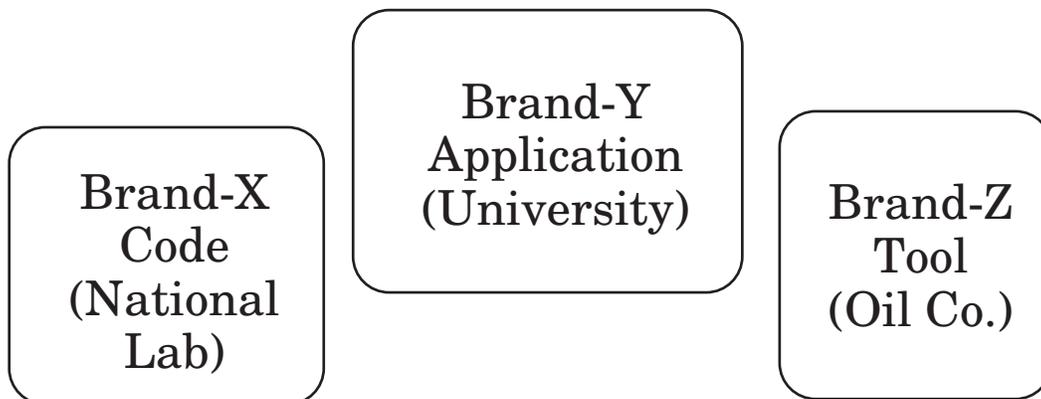
A lot more can be said about the communication issues that the *Toolbox* addresses and supports, and that underly grid-based communication described here, as a particular example; this discussion appears elsewhere [14, 18], and we will not consider these issues further at present. We will also omit details of the data structures involved in the abstraction from communication layers to concurrent data structures, leaving this for [18] and further planned publications.

Figure 3. *Toolbox Supports Multiple Independent Contexts in Multicomputer Communication.*

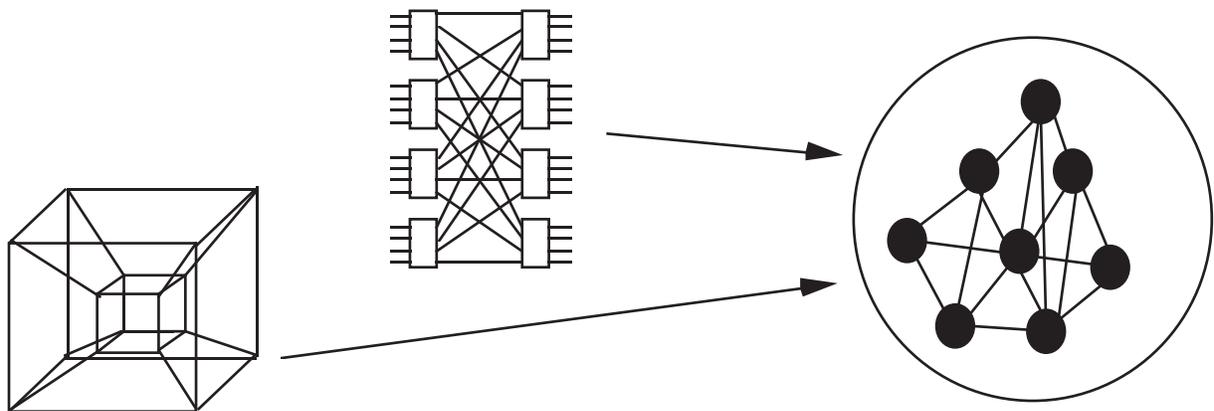


A natural consequence of an open system is multiple uses of shared resources by diverse software; the most prominent in the multicomputer area is the use of the communication resource, which must be managed well, to promote complex applications. Complex “grand-challenge” applications, such as global climate modeling alluded to in this figure, are likely to make serious demands on the software flexibility as well as machine capability of forthcoming parallel supercomputers. The manifest complexity of such codes will require useful abstractions to promote correctness and maintainability.

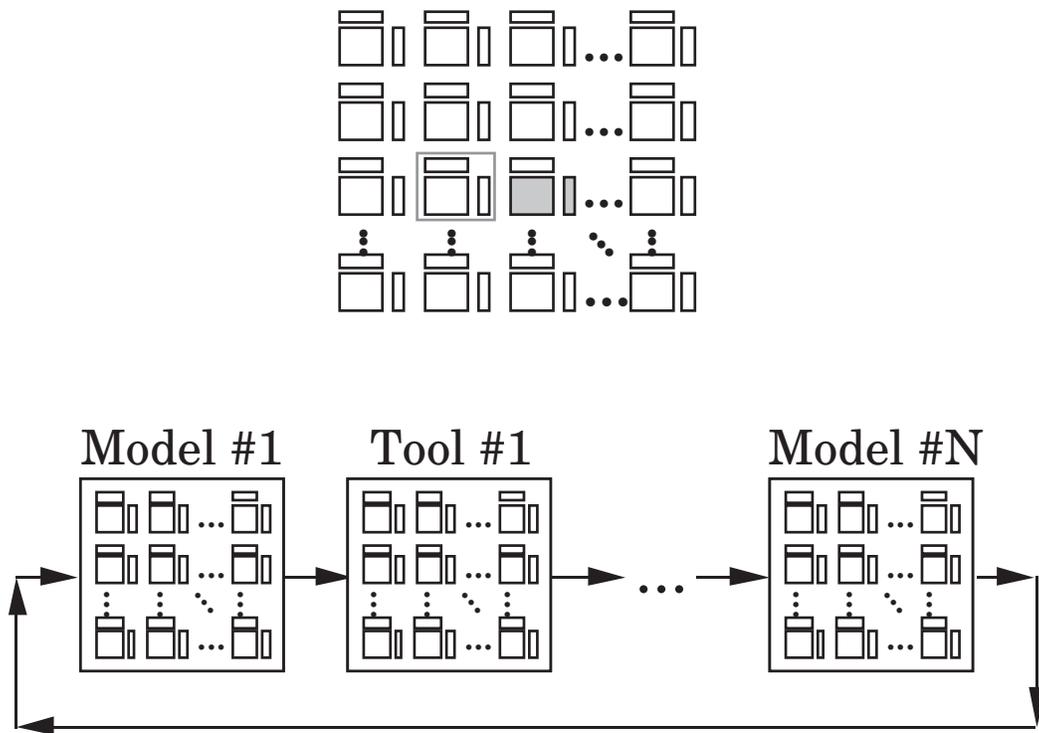
Figure 4. *Toolbox* Anticipates Multiple Sources for both Applications and Useful Tools.



The MIT X-Windows system is successful because it provides an open system within which many authors can create and augment useful tools, to be widely shared subsequent to their completion. Though more modest at present, we have analogous long-term aims for both the scope and impact of the *Toolbox* on parallel library use and development.

Figure 5. *Toolbox* Abstracts Underlying Hardware.

A number of medium-grain machines appear outwardly different, but are commensurable within measures of node count, message latency, message bandwidth, and so on. The *Toolbox* methodology suggests programming all these machines equivalently, while tuning for specific details using data-distributions and grid layouts.

Figure 6. *Toolbox* Manages Data Layout Inherent in Multicomputer Applications.

Each step of a concurrent calculation has its own naturally “optimal” data layout and emphasis on concurrency, given a finite numbers of nodes, and finite communication speed. When assembling a calculation, the goal is overall reduction of time to solution. It is therefore necessary to compromise between mutually conflicting optimal layouts in a calculation; the *Toolbox* software is designed with this in mind.

3 Basic Computing Issues

Amdahl's law is an often-quoted (and misquoted) measure of the upper limit on achievable performance for a concurrent algorithm [1, 3, 8]. It is a fundamental statement that sequential bottlenecks limit the ability of concurrency to provide *speedup*. Elsewhere, additional discussion of *scaled performance measures* is offered [14, Appendix F]; we omit such discussion here.

3.1 Nomenclature

Definition 3.1 (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*

$$S_p \equiv \frac{T_1}{T_p}. \quad (1)$$

Definition 3.2 (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*

$$\hat{S}_p \equiv \frac{T_{seq}}{T_p}, \quad (2)$$

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 3.3 (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*

$$\eta_p \equiv \frac{S_p}{p}. \quad (3)$$

Definition 3.4 (Fair Efficiency) *The fair efficiency is defined as*

$$\hat{\eta}_p \equiv \frac{\hat{S}_p}{p}, \quad (4)$$

in analogy to the relative efficiency.

Definition 3.5 (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

$$T_p \equiv \alpha T_1 + \frac{(1 - \alpha)T_1}{p}, \quad (5)$$

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

$$S_p = \frac{p}{1 + (p - 1)\alpha} = \frac{1}{\alpha + (1 - \alpha)/p} \leq S_\infty \equiv \alpha^{-1} \quad (6)$$

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 3.6 (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 3.2).

Definition 3.7 (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

$$S_p^{p_0} \equiv \frac{T_{p_0}}{T_p}. \quad (7)$$

$S_p^{p_0}$ is a measure of performance that we can actually measure on a given ensemble without resorting to arbitrary or external performance references. Clearly, for small problems, this is identical to the relative speedup (Amdahl-brand) defined in above. See also [14, Appendix F]. 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 [14, Appendix F] and [5].

Limited communication capability (communication startup latency, bandwidth, and contention between nodes for total 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 set of design heuristics, denoted the "granularity design groups," help in the semi-empirical performance evaluation of existing algorithms and in the top-down design of new concurrent algorithms. In the following, we tacitly assume that work has been sensibly balanced over the ensemble, so that the design groups measure an ensemble-average of work (*i.e.*, not grossly unbalanced levels of computation or communication, given an overall problem size).

Definition 3.8 (Granularity Design Groups) *Let γ be the startup cost in μs needed for transmitting a message between any two processes. Let δ be the per-byte incremental cost of a message transmission, also in μs , and let y be the number of bytes per operand (e.g., $y = 8$ for double precision real numbers on IEEE-standard machines) 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 local granularity design equation is as follows:*

$$g_{loc}(n, o) \equiv \frac{\gamma + ny\delta}{\tau on}, \quad (8)$$

where g_{loc} is a measure of the per-operand cost of the message transmission. For a hypothetical, but reasonable, multicomputer, $\gamma \sim 100\mu\text{s}$, $\delta \sim .1\mu\text{s}$, and $\tau \sim .2\mu\text{s}$. Consequently, assuming these values, for a message of ten operands, with ten operations per operand, $g_{loc}(10, 10) = 5.4$. By way of comparison (with the same hardware parameters), $g_{loc}(1, 1) \approx 504$. $g_{loc}(1, 1)$ as a measure of system granularity misses the economic idea that there are fixed and incremental costs of transmission, and that good algorithms amortize communication over larger messages. Assuming a global operation is involved, a "global" design group can be posed (for P participants):

$$g_{glob}(n, o, P) \equiv \frac{(\gamma_1 + ny\delta_1) + (\gamma_2 + ny\delta_2) \log_2 P}{\tau on}, \quad (9)$$

Again, an ensemble-averaging of overall work is presupposed.

These design groups should be applied wherever possible a priori in the top-down design of concurrent algorithms to determine qualitatively 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 degree of concurrency, P , and, by implication, the average “grain size” for the problem formulation. For existing machines, good estimates for the τ , γ and δ parameters may be made by direct measurement.

The qualitative use of these and other dimensionless groups, describing hardware and algorithmic parameters, offers potentially powerful ways to describe the scalability of algorithms and performance on parallel machines. This area should be investigated further, in preference to non-physical performance models, involving dozens or hundreds of arbitrary parameters.

Definition 3.9 (Process Grain Size) *For a given operation, if its $g_{oper} \sim g_{oper}(1, 1)$, it is said to be fine-grain, while if its $g_{oper} \ll g_{oper}(1, 1)$, it is said to be coarse grain.*

Definition 3.10 (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 paper. They have several megabytes of memory, and up to several hundred nodes. See also [9].*

The final generic source of inefficiency¹ 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 below (scatter-scatter distribution of matrices).

Flatt ([3]) indicates extended Amdahl's-law arguments for computational models including communication and load-imbalance effects, which we build upon here. We begin

¹See [5, pages 636-637] concerning other sources of inefficiency.

again with a more detailed model of the execution time:

$$T_p = \left(\alpha + \frac{(1 - \alpha)}{p} + \varsigma(p) \right) T_1, \quad (10)$$

where $\varsigma(p)$ is a measure of overheads, such as communication and load imbalance; $\varsigma(1) \equiv 0$. Substituting Equation 10 into the definition for speedup, Equation 1,

$$S_p = \frac{p}{1 + (p - 1)\alpha + p\varsigma(p)} = \frac{1}{\alpha + (1 - \alpha)/p + \varsigma(p)}. \quad (11)$$

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

$$\varsigma(p) = \varsigma_0 \log_2 p + \varsigma_1(p - 1) + \varsigma_2 \left(\frac{1}{p} - 1 \right). \quad (12)$$

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.”

3.2 The Concurrency Diagram

Flatt ([3]) 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 7. 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.

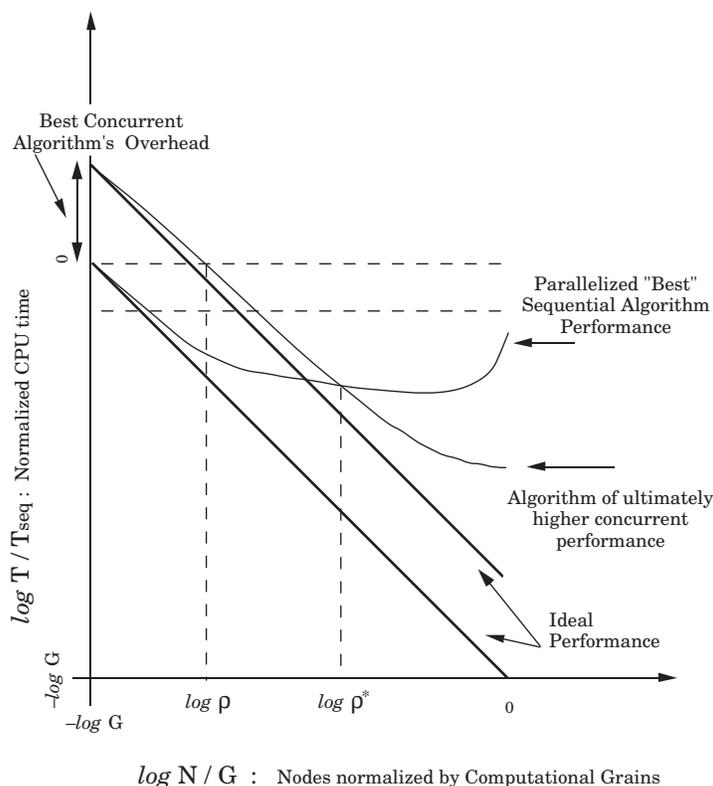
Of course, we have to work to design the “best” concurrent algorithm so that it has these desirable qualities. This will prove non-trivial, in general. We also note that, sometimes, the process of searching for new concurrent algorithms actually generates a better sequential algorithm. In this case, we must relabel our diagram so that, again, the “best” sequential algorithm has the lowest time for the single-processor limit (see, for example, [26]). Otherwise, we are artificially inflating our results.

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.

3.3 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.*, [13]) 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 [7].

Figure 7. 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^*).

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 [19, 21] 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. Results cited in [10] are promising in this connection.

3.4 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 or emulated² on a number of parallel and standard computers. Thus, our applications port immediately between Symult s2010 and Intel iPSC/1, iPSC/2, iPSC/860 (Gamma), Delta, nCUBE/2 6400, and BBN TC2000 multicomputers as well as Sequent multiprocessor systems and conventional ether-networks of Sun or IBM 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 [12].

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) [6]. These primitives provide untyped, blocked and unblocked message passing of arbitrary

²We have developed and support such *Reactive Kernel* emulation on a number of the machines mentioned.

length messages. In our applications, we include an application-oriented communications layer, *Zipcode*, thus avoid direct reference to either the *Reactive Kernel* or vendor-specific primitives. We cover this work in detail in [18].

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.

4 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 8). We describe linear-algebra procedures at a qualitative level within the grid formalism.

4.1 Grids and Communication, Primitive Operations

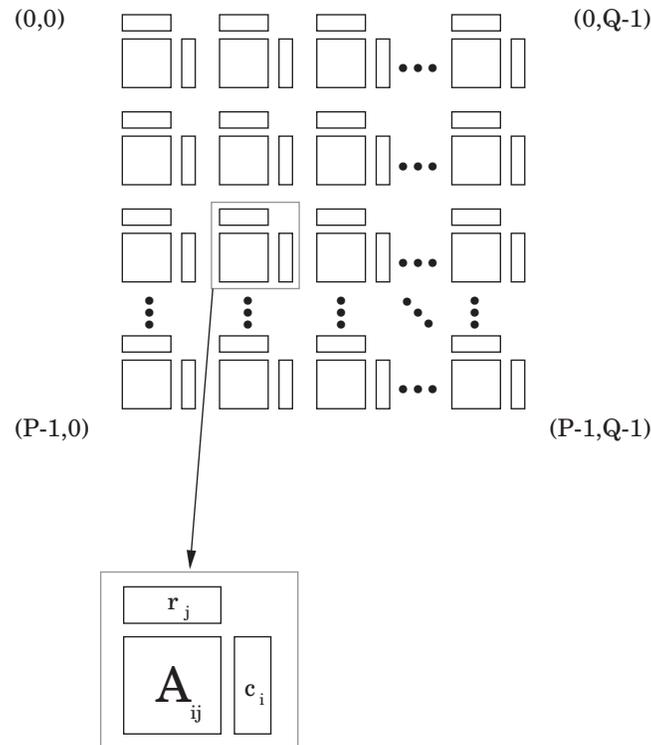
The key operations that connect parts of an ensemble calculation in a multicomputer are the communication steps. Vendors provides very low-level commands with the result that applications tailor their own primitives on top of the basic operations. We've defined a set of higher-level operations realized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

4.1.1 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 $0 \dots R - 1$ 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

Figure 8. 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 replicated in the direction orthogonal to their distribution. Matrices are distributed but not replicated. Replication is used to reduce communication.

integers $(p, q) \mapsto (0 \dots R - 1)$, 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 process 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.

4.1.2 Scalable 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 allotted 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 elements³ (z_0, \dots, z_{r-1}), the first process row, (z_r, \dots, z_{2r-1}), and so forth, with the last ($P-1$ st) process row receiving the final elements (z_{N-r}, \dots, 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 linear system solution via LU factorization. It's common in LU factorization for successive rows and/or columns of the matrix to be eliminated in turn⁴. 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. 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 [14]).

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

³For simplicity of exposition, assume here that $N = rP$, something that won't generally happen.

⁴under partial pivoting strategies.

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 below). So, for example, the identical LU factorization code would function (somewhat 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. Further below, we show how an actual *Toolbox* dense LU code ran on a variety of grid shapes, for different data distributions.

Scalable, 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.

4.2 Selected Communication Operations

4.2.1 The “Combine” Operation — Recursive Doubling

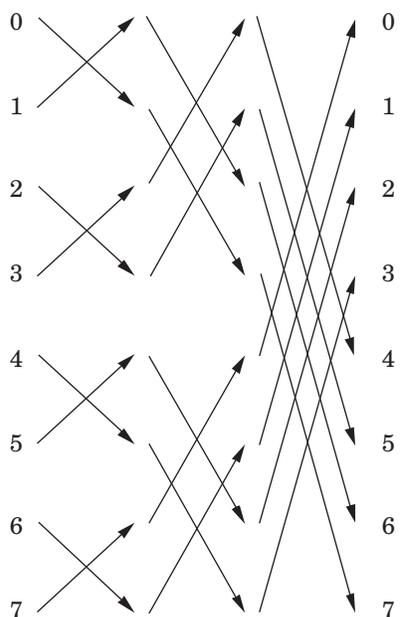
Recursive doubling is the ubiquitous operation of multicomputer programming that lets us accumulate data in an efficient way. It is defined as follows:

Definition 4.1 (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 9. See also [22].*

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).

Figure 9. Recursive Doubling Schematic



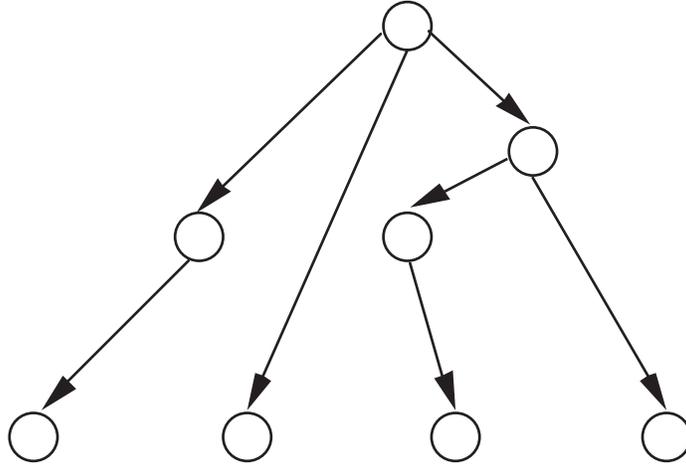
With 2^3 participants, the recursive doubling procedure completes in three steps, illustrating its logarithmic complexity in the number of participants.

4.2.2 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.

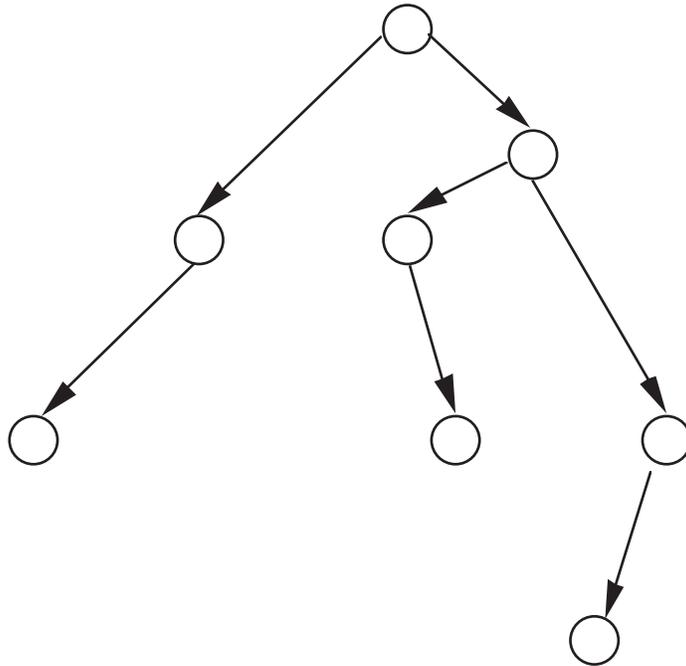
Figures 10, 11, and 12 depict three styles of *broadcast* possible among eight participants. The time for completion of the first is $\lceil \log_2 R \rceil$ for R participants. The second and third variations require one more communication phase each, $\lceil \log_2 R \rceil + 1$. Here we indicate the 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 10. Broadcast Type #1 Schematic



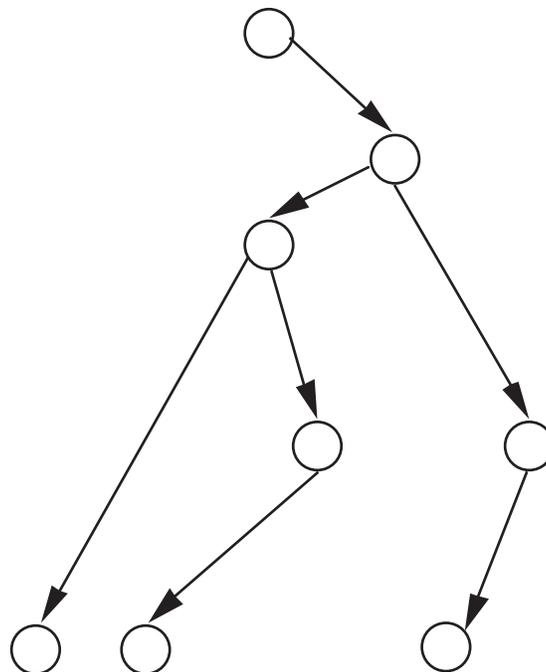
This *broadcast (fanout)* requires the optimum number of communication phases, $\lceil \log_2 R \rceil$ for R participants, but loads the originating node most heavily. This is the mechanism for *broadcasts* currently used in the *Zipcode* layer described in [14, Chapter 3].

Figure 11. 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 12. 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.

4.2.3 The “Fanin” Operation — Collapsing and Operating

Definition 4.2 (Fanin Operation (Collapse)) *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 fanin procedure produces the combined result in one of the R processes in $\lceil \log_2 R \rceil$ steps. Each process must know in advance which process is to be the ultimate destination for the result. This procedure can be visualized by looking at any of the three alternative fanout operations, but arrows reversed, where the associative-commutative operation is applied whenever a data item arrives, yielding one item to be sent on.*

As with *combine* and *fanout*, *fanins* on subcollections are often useful, for instance, row and column fanins on logical process grids.

A *fanin* followed by a *fanout* (for any specified *fanin*-destination, *fanout*-source process) produces the same effect as a *combine* operation, though with less messages and no repetitive calculation. For conditions where reduced communication contention as a result

of less messages or calculation cost as a result of non-repetitive calculations produces better effective performance (both probable for the long data item limit), variants on *combine* based on *fanins* and fanouts can be considered as alternative methods for effecting the same concurrent operations. Such detailed optimizations can fortunately be made transparent to the average user, and readily available to the advanced user.

4.3 Data Distributions

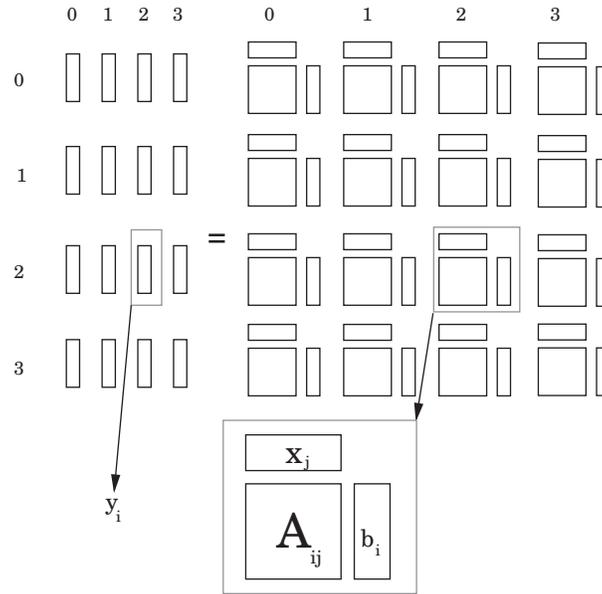
Mappings between global and local coefficient names are fundamental to the control of data locality in multicomputation. Prior to this work, the prevalent forms of data distribution were the simple linear and scatter distributions. Here, we have generalized the linear and scatter distributions by introducing 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 [14, Chapter 5], we demonstrate the effectiveness of these new distributions in sparse linear algebra. Here, we quantify evaluation costs of some of these functions in Table 1.

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.

Every concurrent data structure in our grid-oriented computations is associated with a logical process grid at creation (*cf.*, Figure 13.) Recall that 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 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.* [4] 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 (adaptability to application requirements).

Definition 4.3 (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*

Figure 13. Process Grid Data Distribution of $Ax = b$ 

Representation of a concurrent matrix, and distributed-replicated concurrent vectors on a 4×4 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 .

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 .

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:

$$\mathcal{I}^{p_1} \cap \mathcal{I}^{p_2} = \emptyset, \quad \forall p_1 \neq p_2, \quad 0 \leq p_1, p_2 < P \quad (13)$$

$$\bigcup_{p=0}^{P-1} \mathcal{I}^p = \{0, \dots, M-1\} \equiv \mathcal{I}_M. \quad (14)$$

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:

$$\mathcal{I}^p = (I_0^p, I_1^p, \dots, I_{\mu^\sharp(p, P, M)-1}^p), \quad (15)$$

Figure 14. Example of Process-Grid Data Distribution

$$\begin{pmatrix} A^{0,0} & A^{0,1} & A^{0,2} & A^{0,3} \\ A^{1,0} & A^{1,1} & A^{1,2} & A^{1,3} \\ A^{2,0} & A^{2,1} & A^{2,2} & A^{2,3} \\ A^{3,0} & A^{3,1} & A^{3,2} & A^{3,3} \end{pmatrix}_{\mathcal{G}} = \begin{pmatrix} a_{0,1} & a_{0,5} & a_{0,2} & a_{0,6} & a_{0,3} & a_{0,7} & a_{0,0} & a_{0,4} & a_{0,8} \\ a_{1,1} & a_{1,5} & a_{1,2} & a_{1,6} & a_{1,3} & a_{1,7} & a_{1,0} & a_{1,4} & a_{1,8} \\ a_{2,1} & a_{2,5} & a_{2,2} & a_{2,6} & a_{2,3} & a_{2,7} & a_{2,0} & a_{2,4} & a_{2,8} \\ a_{3,1} & a_{3,5} & a_{3,2} & a_{3,6} & a_{3,3} & a_{3,7} & a_{3,0} & a_{3,4} & a_{3,8} \\ a_{4,1} & a_{4,5} & a_{4,2} & a_{4,6} & a_{4,3} & a_{4,7} & a_{4,0} & a_{4,4} & a_{4,8} \\ a_{5,1} & a_{5,5} & a_{5,2} & a_{5,6} & a_{5,3} & a_{5,7} & a_{5,0} & a_{5,4} & a_{5,8} \\ a_{6,1} & a_{6,5} & a_{6,2} & a_{6,6} & a_{6,3} & a_{6,7} & a_{6,0} & a_{6,4} & a_{6,8} \\ a_{7,1} & a_{7,5} & a_{7,2} & a_{7,6} & a_{7,3} & a_{7,7} & a_{7,0} & a_{7,4} & a_{7,8} \\ a_{8,1} & a_{8,5} & a_{8,2} & a_{8,6} & a_{8,3} & a_{8,7} & a_{8,0} & a_{8,4} & a_{8,8} \\ a_{9,1} & a_{9,5} & a_{9,2} & a_{9,6} & a_{9,3} & a_{9,7} & a_{9,0} & a_{9,4} & a_{9,8} \\ a_{10,1} & a_{10,5} & a_{10,2} & a_{10,6} & a_{10,3} & a_{10,7} & a_{10,0} & a_{10,4} & a_{10,8} \end{pmatrix}$$

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 $\mathcal{G} \equiv \left(\left\{ (\lambda_2, \lambda_2^{-1}, \lambda_2^\#); P = 4, M = 11 \right\}, \left\{ (\sigma_1, \sigma_1^{-1}, \sigma_1^\#); Q = 4, N = 9 \right\} \right)$. Coefficient Subscripts (*i.e.*, $a_{I,J}$) are the global (I, J) indices.

where

$$I_i^p \equiv \mu^{-1}(p, i, P, M). \quad (16)$$

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 here (and in [25]) for unit block size, but differ in coefficient placement. This distinction is necessary for efficient implementations.

4.3.1 Conventional Functions

Definition 4.4 (Linear) *The conventional linear, load-balanced distribution, inverse and coefficient cardinality functions are as follows:*

$$\begin{aligned} \hat{\lambda}(I, P, M) &\mapsto (p, i), \\ p &\equiv \max \left(\left\lfloor \frac{I}{\hat{l} + 1} \right\rfloor, \left\lfloor \frac{I - \hat{r}}{\hat{l}} \right\rfloor \right), \end{aligned} \quad (17)$$

$$i \equiv I - p\hat{l} - \min(p, \hat{r}), \quad (18)$$

while

$$\hat{\lambda}^{-1}(p, i, P, M) \equiv i + p\hat{l} + \min(p, \hat{r}) \mapsto I, \quad (19)$$

$$\hat{\lambda}^{\sharp}(p, P, M) \equiv \left\lfloor \frac{M + P - 1 - p}{P} \right\rfloor, \quad (20)$$

where

$$\hat{l} = \left\lfloor \frac{M}{P} \right\rfloor, \quad \hat{r} = M \bmod P, \quad (21)$$

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 4.5 (Scatter) *The conventional scatter distribution, inverse and cardinality functions are as follows:*

$$\hat{\sigma}(I, P, M) \equiv \left(I \bmod P, \left\lfloor \frac{I}{P} \right\rfloor \right) \mapsto (p, i), \quad (22)$$

$$\hat{\sigma}^{-1}(p, i, P, M) \equiv iP + p \mapsto I \quad (23)$$

and where

$$\hat{\sigma}^{\sharp}(p, P, M) = \hat{\lambda}^{\sharp}(p, P, M), \quad (24)$$

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. ■

4.3.2 Block Versions

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

Definition 4.6 (Block-Linear) *The block-linear, load-balanced distribution, inverse and coefficient cardinality functions (with blocksize $B \geq 1$) are as follows:*

$$\hat{\lambda}_B(I, P, M) \mapsto (p, i),$$

$$p \equiv \max \left(\left\lfloor \frac{I_B}{l+1} \right\rfloor, \left\lfloor \frac{I_B - r}{l} \right\rfloor \right), \quad (25)$$

$$i \equiv I - B(pl + \min(p, r)), \quad (26)$$

while

$$\hat{\lambda}_B^{-1}(p, i, P, M) \equiv i + B(pl + \min(p, r)) \mapsto I, \quad (27)$$

$$\lambda_B^\sharp(p, P, M) \equiv B \left\lfloor \frac{b + P - 1 - p}{P} \right\rfloor, \quad (28)$$

where b, l, r, I_B are as defined on page 31 (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:

$$\max \left(\left\lfloor \frac{I_B}{l+1} \right\rfloor, \left\lfloor \frac{I_B - r}{l} \right\rfloor \right). \quad (29)$$

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 $0, \dots, p - 1$. 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

$$l = \left\lfloor \frac{b}{P} \right\rfloor, \quad r = b \bmod P, \quad (30)$$

and, consequently,

$$\left\lfloor \frac{b + \tilde{p}}{P} \right\rfloor = l + 1 \quad \text{for } P - r \leq \tilde{p} \leq 2P - r - 1. \quad (31)$$

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

$$\hat{\sigma}(I, P, M) \equiv \left(I_B \bmod P, \left(B \left\lfloor \frac{I_B}{P} \right\rfloor + I \bmod B \right) \right) \mapsto (p, i), \quad (32)$$

$$\hat{\sigma}^{-1}(p, i, P, M) \equiv B \left(p + \left\lfloor \frac{i}{B} \right\rfloor P \right) + i \bmod B \mapsto I. \quad (33)$$

and where

$$\hat{\sigma}_B^\sharp(p, P, M) \equiv \hat{\lambda}_B^\sharp(p, P, M). \quad (34)$$

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 $0, \dots, I_B - 1$. 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. ■

4.3.3 New Data Distributions

Definition 4.8 (Generalized Block-Linear) *The definitions for the generalized block-linear distribution function, inverse, and cardinality function are*

$$\lambda_B(I, P, M) \mapsto (p, i),$$

$$p \equiv P - 1 - \max\left(\left\lfloor \frac{I_B^{\text{rev}}}{l+1} \right\rfloor, \left\lfloor \frac{I_B^{\text{rev}} - r}{l} \right\rfloor\right), \quad (35)$$

$$i \equiv I - B(p l + \Theta^1(p - (P - r))), \quad (36)$$

while

$$\lambda_B^{-1}(p, i, P, M) \equiv i + B(p l + \Theta^1(p - (P - r))), \quad (37)$$

$$\lambda_B^\sharp(p, P, M) \equiv B\left(\left\lfloor \frac{b+p}{P} \right\rfloor - \theta\right) + (M \bmod B)\theta, \quad (38)$$

where B denotes the coefficient block size,

$$b = \begin{cases} \frac{M}{B} & \text{if } M \bmod B = 0 \\ \left\lfloor \frac{M}{B} \right\rfloor + 1 & \text{otherwise,} \end{cases} \quad (39)$$

$$I_B = \left\lfloor \frac{I}{B} \right\rfloor, \quad I_B^{\text{rev}} = b - 1 - I_B, \quad (40)$$

$$l = \left\lfloor \frac{b}{P} \right\rfloor, \quad r = b \bmod P, \quad (41)$$

$$\Theta^k(t) \equiv \begin{cases} 0 & t \leq 0 \\ t^k & t > 0, k > 0 \\ 1 & t > 0, k = 0 \end{cases}, \quad (42)$$

$$\theta = \left\lfloor \frac{p+1}{P} \right\rfloor \Theta^0(M \bmod B), \quad (43)$$

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 $p = 0, \dots, P - 1$ 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 4.9 (Generalized Block-Scatter) *The generalized block-scatter distribution, inverse and coefficient-cardinality function are, in turn, as follows:*

$$\sigma_B(I, P, M) \equiv \left(P - 1 - (I_B^{\text{rev}} \bmod P), \right. \quad (44)$$

$$\left. B \left(\left\lfloor \frac{b+p}{P} \right\rfloor - 1 - \left\lfloor \frac{I_B^{\text{rev}}}{P} \right\rfloor \right) + I \bmod B \right) \quad (45)$$

$$\mapsto (p, i),$$

$$\sigma_B^{-1}(p, i, P, M) \equiv B \left(((b+p) \bmod P) + P \left\lfloor \frac{i}{B} \right\rfloor \right) + (i \bmod B) \mapsto I, \quad (46)$$

$$\sigma_B^\sharp(p, P, M) \equiv \lambda_B^\sharp(p, P, M), \quad (47)$$

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 $p = 0, \dots, P - 1$, 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 4.10 (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:*

$$\xi_{B,S}(I, P, M) \mapsto (p, i) \equiv \begin{cases} (p_0, i_0) & \Lambda_0 \geq l_S \\ (p_1, i_1) & \Lambda_0 < l_S \end{cases}, \quad (48)$$

where

$$l_S \equiv \left\lfloor \frac{l}{S} \right\rfloor, \quad \Lambda_0 \equiv \left\lfloor \frac{i_0}{BS} \right\rfloor, \quad (49)$$

$$(p_0, i_0) \leftarrow \lambda_B(I, P, M), \quad (50)$$

$$I_{BS} = p_0 l_S + \Lambda_0, \quad (51)$$

$$p_1 \equiv I_{BS} \bmod P, \quad (52)$$

$$i_1 \equiv BS \left\lfloor \frac{I_{BS}}{P} \right\rfloor + (i_0 \bmod BS), \quad (53)$$

with

$$\zeta_{B,S}(I, P, M) \equiv \xi_{1,S}(I, P, M), \quad (54)$$

$$\xi_{B,S}^\sharp(p, P, M) \equiv \lambda_B^\sharp(p, P, M), \quad (55)$$

$$\zeta_S^\sharp(p, P, M) \equiv \lambda_1^\sharp(p, P, M), \quad (56)$$

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

$$\xi_{B,S}^{-1}(p, i, P, M) \mapsto I = \lambda_B^{-1}(p^*, i^*, P, M), \quad (57)$$

$$(p^*, i^*) \equiv \begin{cases} (p, i) & \Lambda \geq l_S \\ (p_2, i_2) & \Lambda < l_S \end{cases}, \quad (58)$$

$$\Lambda \equiv \left\lfloor \frac{i}{BS} \right\rfloor, \quad I_{BS}^* = p + \Lambda P, \quad (59)$$

$$p_2 \equiv \left\lfloor \frac{I_{BS}^*}{l_S} \right\rfloor, \quad (60)$$

$$i_2 \equiv BS (I_{BS}^* \bmod l_S) + (i \bmod BS), \quad (61)$$

with

$$\zeta_S^{-1}(p, i, P, M) \equiv \xi_{1,S}^{-1}(p, i, P, M). \quad (62)$$

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

Definition 4.11 (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:

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

respectively,

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

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 [14, Appendix D]. Figure 14. illustrates the effects of linear and scatter data-distribution functions on a small rectangular array of coefficients.

Definition 4.12 (Data-Distribution Projection) A data-distribution projection is a mapping of the form

$$\Gamma_{\nu \circ \mu^{-1}}(p, i, P, M; Q, N) \equiv \nu(\mu^{-1}(p, i, P, M), Q, N) \mapsto (q, j) \quad (63)$$

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

$$\mathcal{G}^I \equiv \{(\mu, \mu^{-1}, \mu^\sharp); P, M\}, \quad (64)$$

$$\mathcal{G}^{II} \equiv \{(\nu, \nu^{-1}, \nu^\sharp); Q, N\}, \quad (65)$$

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 63's "inverse" is defined as:

$$\Gamma_{\mu \circ \nu^{-1}}(q, j, Q, N; P, M) \equiv \mu(\nu^{-1}(q, j, Q, N), P, M) \mapsto (p, i), \quad (66)$$

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 63 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 66 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 4.13 (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*

$$\Upsilon_{\nu \circ \mu^{-1}}(p, i, P, Q, N) \equiv \begin{cases} j & \text{for } q = p \\ -1 & \text{otherwise} \end{cases}, \quad (67)$$

and

$$\Upsilon_{\mu \circ \nu^{-1}}(q, j, Q, P, N) \equiv \begin{cases} i & \text{for } p = q \\ -1 & \text{otherwise} \end{cases}, \quad (68)$$

where -1 connotes “none.”

Equation 67 (68), 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 4.1 (Local Projections & Weak Distributions) *The local projections are correct if weak data distributions ω , ν substitute for the strong data distributions μ , ν in Equations 67, 68.*

Proof We prove correctness for Equation 67. Equation 68 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 ν 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, $\nu(I, Q, N)$ immediately completes the local projection,

$$\Upsilon_{\nu \circ \omega^{-1}}(p, i, P, Q, N), \quad (69)$$

as claimed. ■

Corollary 4.1 (Mixtures of Strengths) *The local data-distribution projections below (section 5) are correct if either strong data distribution μ or ν in Equations 67, 68 is replaced by a weak data distribution.*

Proof By inspection. ■

Corollary 4.2 (Weak 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. ■

5 Some Toolbox Kernels

Here, we introduce concurrency kernels of interest, first weighted vector sum and dense matrix-vector multiply. The two closely related grid-based vector operations, *transpose_row_to_column* and *transpose_column_to_row*, are introduced subsequently. 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 latter kernel does the opposite: a column-distributed vector is transformed into a row-distributed vector. The last kernels are three formulations of grid-based inner product, including a special version of inner product, “skew inner product,” as well as a communication-free outer product, “skew outer product.” The skew inner (outer) product forms the inner (outer) product of a row-distributed and a column-distributed vector; in connection with it, we also discuss two variants of the regular inner product operation, slight generalizations of an unweighted norm operation.⁵

5.1 The Weighted-Vector Sum

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

Definition 5.1 (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.

5.2 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.

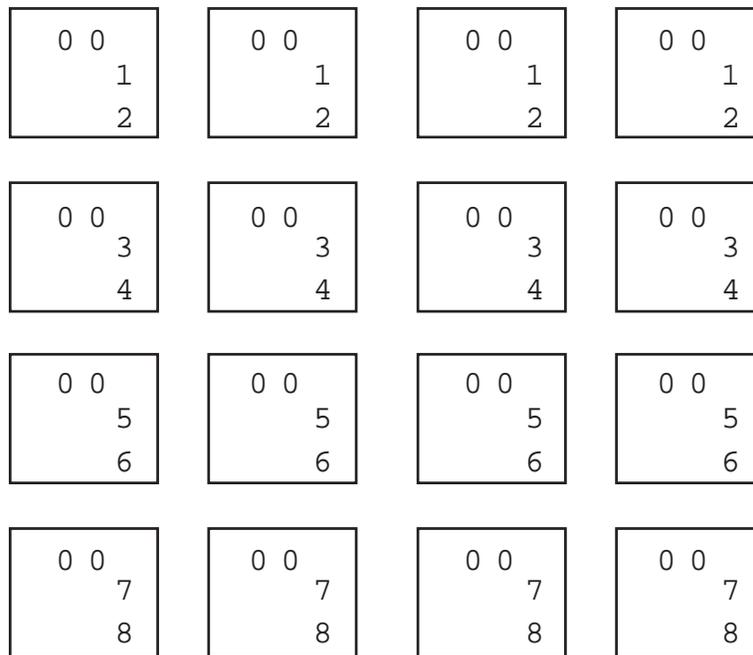
Definition 5.2 (MVP: The Matrix-Vector Product) *Given a grid \mathcal{G} , an $M \times N$ matrix A distributed on \mathcal{G} , a column-distributed N -vector \mathbf{x} (distributed as if it were a row of A*

⁵A full BLAS-compatible parallel vector norm is to be added, for its desirable, and standard, numerical properties.

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.⁶ 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 4×4 process grid in Figure 13.

Figure 15. 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 4×4 process grid. The destination column-oriented vector has all its elements set to zero initially.

⁶The portable Basic Linear Algebra Subprograms are replacing existing loop-level sequential operations in the *Toolbox*. This upgrade allows for higher performance on non-scalar-node multicomputers, while the *Toolbox* user interface remains the same. Technically, the BLAS level-2 operation `dgemv` replaces the nodal dot-product-based operations of the original implementation, and are often highly optimized on newer machines.

5.3 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. The identical need for this data motion occurs in an iterative linear system solution based on matrix-vector multiplies.

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

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

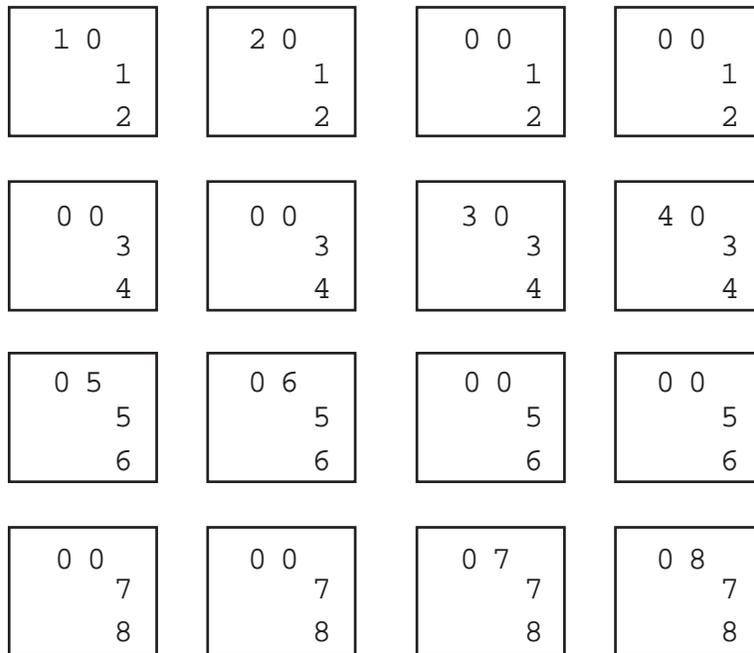
and

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

Definition 5.3 (*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 15. 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 16., copying of elements according to the local projections has been accomplished. The rest of the elements remain zero. In Figure 17., the column-wise combine operations have been effected, completing the conversion.

Definition 5.4 (*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,*

Figure 16. Step #2 of the *transpose_row_to_column* operation

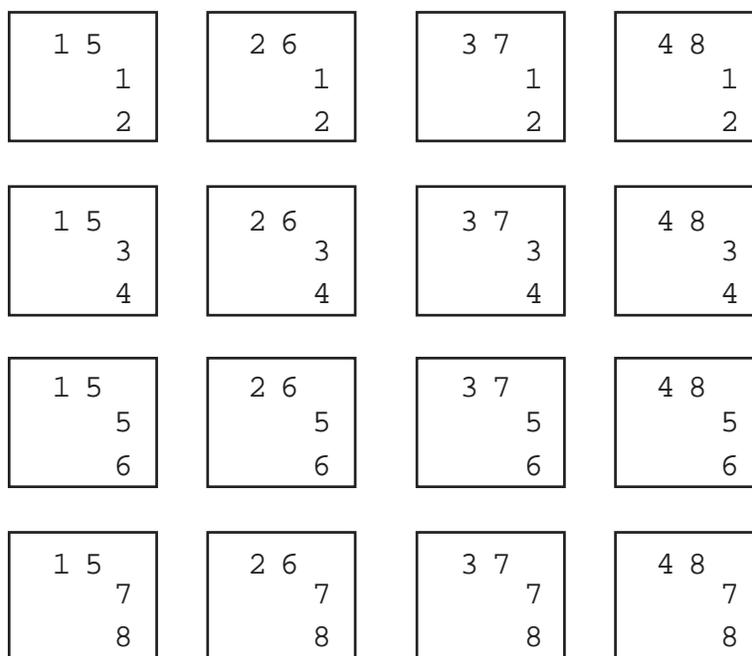
According to the local projection of coefficients, appropriate elements are copied into the column-oriented vectors. The remaining elements in the column-oriented vector are untouched and remain zero.

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.

5.4 Inner Products

First we define two conventional inner products per, for example, [23, 24]. Then we consider a new kernel, the skew inner product, and a communication-free kernel, skew outer product.

Definition 5.5 (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-*

Figure 17. Step #3 of the *transpose_row_to_column* operation

Applying a *combine* to each process column converts the representation of Step #2 in Figure 16 to the final form depicted here. The combination operation is a simple element-by-element addition.

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, 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 preceding and proceeding the inner product.

Technical details are changed in the above as we move to the use of level-1 Basic Linear Algebra Subprograms as the *local* computation kernels (BLAS), replacing loops. Efficient BLAS are both portable and take advantage of machine features not handled as efficiently by compilers.

Definition 5.6 (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.

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 our *Toolbox* preconditioned conjugate gradient kernel (PCG), but we don’t have scope to pursue this further here.

Definition 5.7 (Skew Outer Product) *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. In the latest *Toolbox* software, each local outer-product is accomplished using the portable sequential level-2 (BLAS) function `dger`, for highest performance on machines where pipelining and/or vectorization are supported.*

6 A Toolbox Milestone

As a proof-of-principle, as well as a significant related area of research, we have recently worked on high performance, data-distribution-independent dense LU factorization algorithms, and demonstrated these on the Caltech Intel Delta prototype, during its “acceptance test phase,” when, despite machine instability, we were able, within the *Toolbox* framework, to generate sustained performance of three double-precision gigaflops (3.0×10^9) with a non-blocking, right-looking, data-distribution independent LU factorization, on an order 10,000 dense matrix. This level of performance occurs for a particular logical grid shape, 18×28 , with a scatter-scatter distribution of both rows and columns. Less-than-optimal grid shapes and data distributions generate somewhat inferior results (see Tables 3, 4, 5). However, for a number of variations, performance degrades only slightly. Hence, the needs of the application generating the matrix, in terms of row- and column- parallelism, and in terms of data locality can still be factored into the overall tuning of an application. For problems not much smaller than our test example, explicit redistribution of data is prohibitively expensive on the Delta and similar machines. Consequently, the data-distribution-independent algorithm is the most important one: it can generate the high performance results for the special distribution that is “optimal” for it; yet, it can also function when an application code needs to generate some other distribution.

For the present solver, the computational kernel is a BLAS level-2, rank-1 update called *dger*, the least efficient of level-2 operations, because of its data reuse characteristics. Its single-node performance is depicted in Table 2. Our further work, still in progress, has led us to a data-distribution-independent level-3 BLAS right-looking solver, capable of approximately eight gigaflops for the best data distribution, and useful (with somewhat degraded performance) for other distributions that may occur in real applications. When complete, this solver will offer higher computational performance, without sacrificing data-distribution independence. In other words, we find that, with greater effort, we can still exploit nodal pipelining or vectorization, without seriously compromising data distribution independence. We will report on this further when the work is finalized.

To summarize, for this class of operations, data-distribution independence becomes less important for $N > N^*$, where N^* is the smallest matrix size for which explicit data redistribution costs an order-of-magnitude less than factoring in the “optimal data layout,” both operations considered with the same number of nodes. Even for $N > N^*$, it may not be economical to redistribute data, however, from nearly optimal or even mediocre distributions, to the optimal distribution, because, the improvement in performance may be marginally less valuable than the cost of redistribution. Hence, the data-distribution-independent algorithm remains relevant for large N , depending more or less on the application requirements. For computational steps with lower time complexity, problems will have to be even larger in dimension before one can begin to ignore data distribution independence.

<i>Size</i>	<i>Assembled</i>	<i>Compiled</i>
100	10.343	8.368
250	11.221	9.171
500	11.434	9.560
1000	11.434	9.901

Performance of single-node double-precision rank-1 update of a dense matrix, on the Intel i860 chip, based on code optimized with a compiler, or by hand coding. This serves as the computational kernel for our non-blocking, data-distribution-independent LU factorization.

<i>Shape</i>	<i>Time</i>	<i>Gflops</i>	<i>Mflops/node</i>
6x84	264.827s	2.517	4.995
7x72	242.605s	2.748	5.452
9x56	233.643s	2.853	5.661
10x51	226.765s	2.940	5.765
11x46	225.850s	2.952	5.834
12x42	224.525s	2.969	5.891
13x39	221.401s	3.011	5.939
14x36	224.419s	2.971	5.894
15x34	219.533s	3.037	5.954
16x31	219.273s	3.040	6.130

The Intel Delta performance for the right-looking LU algorithm generated these performance results. Data-distribution-independent, partial-row pivoting LU factorization utilized the assembly coded level-2 rank-1 update dger. Giga-flops were computed as $\frac{2}{3}N^3 10^{-6}/T$ where $N = 10,000$ here, and T was the observed maximal runtime in seconds. Each time T quoted is the average of two or more repetitive runs. A number of similar grid shapes and node counts produce similar performance, suggesting an important degree of freedom left to applications that will call this kernel

<i>Shape</i>	<i>Time</i>	<i>Gflops</i>	<i>Mflops/node</i>
18x28	217.508s	3.065	6.081
21x24	223.732s	2.980	5.912
24x21	241.501s	2.761	5.477
28x18	249.990s	2.667	5.291
32x8	406.676s	1.639	6.404
16x16	351.916s	1.894	7.400
8x32	353.402s	1.886	7.369

Here we see, for the LU factorization, the optimal runtime achieved for the 18×28 grid shape. We also see that using less nodes (last three lines utilize 256 nodes) achieves higher effective per-node performance, but longer overall runtime.

<i>Distribution</i>		<i>Time</i>	<i>Gflops</i>	<i>Mflops/node</i>
<i>Row</i>	<i>Col</i>			
Scat	Scat	225.230s	2.960	5.873
Scat	Lin	254.054s	2.624	5.210
Lin	Scat	343.812s	1.940	3.850
Lin	Lin	444.143s	1.501	2.980

The LU factorization on the efficient 8×63 grid shape, with various data locality choices. Performance degrades with the deviation from scatter-scatter distribution, but by not more than a factor of two in the worst case.

7 Conclusion

Several manufacturers currently produce or plan to produce medium-grain multicomputers with supercomputing potential far beyond traditional vector supercomputers. Unlike the vector supercomputer environment, compiler technology promises little performance for existing “dusty decks” in the massively parallel environment. The alternative, pursued in this research effort, is to provide high quality numerical algorithms, support libraries, and data-structure abstractions to streamline explicit parallel programming, while providing needed portability. *The Multicomputer Toolbox* is a unique effort intended to provide data-distribution-independent programming for a large number of important algorithms, while supporting many real machines. We described the methodology, some of the concur-

rent operations and data structures. We also illustrated our successes and experiences on the Caltech Intel Delta prototype with dense LU factorization in the *Toolbox* framework; we generated examples with more than three gigaflops double-precision performance, with strong expectations of increasing that further, while keeping the flexibility of data-distribution-independence. We argued throughout that data-distribution-independent algorithms are the most generically important to applications.

The most comprehensive description of the *Toolbox* communication support — The *Zipcode* and *UPU* systems — is to be found in [18], while the most complete discussion of data-distribution issues and linear algebra is still [14]; further articles are in preparation. Both [14] and [18] are available on request from the first author. The *Toolbox* software, including on-line documentation, will be widely distributed starting in 1992, first to a growing circle of “friendly collaborators,” and later to the general public. A simple registration process will be required, but the software will be electronically distributed and updated without charge. Users of homogeneous networks of Sun or IBM workstations will be able to test and debug codes, in parallel, for use on actual parallel machines including the nCUBE/2 6400 series and successor, the Intel Gamma, Delta and Paragon machines, the BBN TC2000 machine at LLNL, and on the new Thinking Machines CM-5 series.⁷ Potential users and developers are encouraged to contact the first author to be placed on the *Toolbox* mailing list in one or more of these categories: all, communication support, *Toolbox* infrastructure, numerical tools, chemical engineering applications, other engineering applications, physics applications, chemistry applications. As the *Toolbox* is an open system, we intend to categorize and list all papers related to *Toolbox* use, not only those developed at LLNL.

⁷The Paragon machine will be supported as soon as it is made available in prototype form to us, perhaps in early 1993. The CM-5 port is expected to be available during the second quarter of 1992.

References

- [1] 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.
- [2] William C. Athas and Charles L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, pages 9–24, August 1988.
- [3] H. P. Flatt. A simple model for parallel processing. *IEEE Computer*, page 95, November 1984.
- [4] 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.
- [5] 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.
- [6] C. A. R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, August 1978.
- [7] Selahattin Kuru. *Dynamic Simulation with an Equation Based Flowsheeting System*. PhD thesis, Carnegie Mellon University, 1981. Chemical Engineering Department.
- [8] Sven Mattisson. *CONCISE: A Concurrent Circuit Simulation Program*. PhD thesis, Lund Institute of Technology, Sweden, 1986. Department of Applied Electronics.
- [9] Lena Peterson. A Study of Convergence-Enhancing Techniques for Concurrent Waveform Relaxation, May 1989. *Teknologie Licenciat* Degree Thesis, Lund University, Dept. of Applied Electronics.
- [10] Argimiro R. Secchi, Manfred Morari, and Evaristo C. Biscaia. The Waveform Relaxation Method in the Concurrent Dynamic Simulation. AIChE 1991 Annual Meeting, paper 136d; Submitted to *Computers & Chemical Engineering*, October 1991.
- [11] Charles L. Seitz. The Cosmic Cube. *CACM*, 28(1):22–33, January 1985.
- [12] 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.
- [13] M. G. Singh and A. Titli. *Systems Decomposition, Optimization and Control*. Pergamon, 1978.

- [14] Anthony Skjellum. *Concurrent Dynamic Simulation: Multicomputer Algorithms Research Applied to Ordinary Differential-Algebraic Process Systems in Chemical Engineering*. PhD thesis, Chemical Engineering, California Institute of Technology, May 1990. (You can request a copy from the author).
- [15] Anthony Skjellum and Alvin P. Leung. LU factorization of sparse, unsymmetric jacobian matrices on multicomputers: *Experience, Strategies, Performance*. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, pages 328–337. IEEE, April 1990.
- [16] Anthony Skjellum and Alvin P. Leung. *Zipcode: A Portable Multicomputer Communication Library atop the reactive kernel*. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, pages 767–776. IEEE, April 1990.
- [17] Anthony Skjellum and Manfred Morari. *Concurrent DASSL Applied to Dynamic Distillation Column Simulation*. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, pages 595–604. IEEE, April 1990. Simulation Paradigms Minisymposium.
- [18] Anthony Skjellum and Manfred Morari. *Zipcode: A Portable Communication Layer for High Performance Multicomputing – Practice and Experience*. Technical Report UCRL-JC-106725, Lawrence Livermore National Laboratory, March 1991. Accepted by *Concurrency: Practice & Experience*. In minor revision.
- [19] 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.
- [20] Anthony Skjellum, Manfred Morari, Sven Mattisson, and Lena Peterson. *Highly Concurrent Dynamic Simulation in Chemical Engineering: Issues, Methodologies, Model Problems, Progress*. Contributed paper AIChE 1988 Annual Meeting, December 1988.
- [21] 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.
- [22] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, 1987.
- [23] Eric F. Van de Velde. *Data Redistribution and Concurrency*. Caltech Applied Mathematics, May 1988.
- [24] Eric F. Van de Velde. *Implementation of Linear Algebra Operations on Multicomputers*. Caltech Applied Mathematics, October 1988.

- [25] Eric F. Van de Velde. Experiments with Multicomputer LU-decomposition. *Concurrency: Practice and Experience*, 2(1):1–26, March 1990.
- [26] Stefan Vandewalle. Parallel Waveform Relaxation Methods for Solving Parabolic Partial Differential Equations. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, pages 575–584. IEEE, April 1990.