Local and Multi-Level Parallel Processing Mill

Achi Brandt
Dept. of Applied Mathematics & Computer Science
Weizmann Institute of Science
Rehovot 76100, Israel

General Statement

Many, perhaps most, of the massive computational tasks facing us today have a certain feature in common: they are local-relation problems, usually formulated on a low-dimensional grid. This feature can be exploited by a certain type of algorithms, which combine local processing with inter-scale transfers. These algorithms are generally more efficient than any. The efficiency can be much further enhanced if a special hardware is built with some orientation toward such algorithms, taking full advantage of their inherent parallelism. Moreover, we shall see that such hardware, and corresponding software, can be constructed modularly, and that most of it can very effectively serve general-purpose large-scale parallel processing as part of an open-ended, scalable enterprise.

Local-relation grid problems

The feature common to many large-scale problems is that they have many variables, or unknowns, and that these variables are nicely ordered geometrically. A multitude of unknowns is seldom arbitrary: It is usually produced in an orderly fashion, and most often it arises as some discretization of a continuum. This continuum is either two dimensional, or, in the more massive problems, three and even four dimensional. (Still higher-dimensional problems loom, e.g. in quantum mechanics, waiting further computational breakthroughs). Each unknown is therefore most often naturally assigned with a position in a d-dimensional space, with $2 \leq d \leq 4$, such that the equations relating the unknowns to each other are based on their positions: Usually (e.g., in discretized partial-differential systems) each equation is a *local* equation, connecting just a few neighboring unknowns, except possibly for a few special equations which have a more global nature ("global conditions"). In other cases (e.g., in many discretized integral or integro-differential systems) each equation may connect many, even all unknowns, but the connection to unknowns far from the "center" of the equation is in a certain sense very weak (or very smooth—which would computationally amount to the same thing).

The list of problems falling into the above category is very long. It includes all flow problems, most structural problems, electromagnetism, magnetohydrodynamics, quantum mechanics, statistical physics, image reconstruction, pattern

recognition, some tomography problems, etc., etc. Their fast solution will fundamentally have an impact on many fields of science and technology. We briefly call them *local-relation problems.*

Ordinarily, the positioning of the unknowns in the d-dimensional space can be considered (or chosen) to form *a rectangular grid*, that is, a subset of an array of m rows by n columns (if $d = 2$; and analogously in higher dimensions). Even a non-uniform grid, whose meshsize varies in space, can normally be formed either as a rectangular grid, with non-uniformity being created through coordinate transformation, or as a union of rectangular grids. Indeed, one very effective way of creating flexible non-uniform discretizations is to place uniform grids one on top of the other, with finer grids covering smaller subdomains, some of them possibly also using local coordinate transformations. This approach is especially suitable for operating with the fast solvers described below. It is at any rate only rarely that the continuum contains so much forced structure as to be much better discretized by finite elements which are *not*, at least piecewise, arranged in rectangular grids.

Local processing and inter-scale transfers

Fast solvers of local-relation problems must generally include two types of processes: Local processing and long-range transfers.

Local processing or "*relaxation*", is the classical way for improving any given approximate solution: The equations are taken one by one, or block by block, in some order. Each equation (or block of equations) in its turn is individually solved (or advanced toward its solution) by changing the current value of some neighboring unknowns. Since the equations are not really individual, but rather coupled to each other, one such sweep through all the equations does not solve the system as a whole. However, if a proper relaxation method is chosen, an infinite sequence of such sweeps will give a sequence of approximations converging to the solution, and to obtain some prescribed accuracy a finite number of sweeps will do.

The required number of sweeps may be very large, however. It will normally increase at least linearly, and often much faster, with the number of unknowns per row or column. The general reason is that, since processing at each sweep is local, it must take many sweeps for information to propagate across the array of unknowns, and it must so propagate back and forth several (often many) times before the inter-dependence between far unknowns can properly be accounted for to the prescribed accuracy. Indeed, the normal behavior of the relaxation process is that in the first few sweeps it exhibits fast convergence, which is the fast convergence of *local* errors, i.e., errors which yield large residuals (discrepancies) in the local equations; but then the convergence becomes very slow, because the remaining errors are no longer local: They are much larger than the small residuals they show in the local equations. Such small residuals can breed such large errors

only because they reinforce each other (e.g., they have the same sign) over some larger subdomains. The typical size of these subdomains is called *the scale* of the error function. The larger the scale the more global, and the less local, is the error function, and the slower is the convergence of any strictly local processing.

A general way around this slowness of relaxation is the *multi-level*, or *multi-grid, method*. As soon as the local processing shows signs of slow convergence, the scale of the error function must be large enough for that function to be approximable on a coarser grid. So the method is to formulate at this point a *residual problem* on a coarser grid, whose solution, when interpolated to the original grid, will give a good approximation to the error. Introducing then this approximation as a correction to the solution, a much smaller error is obtained. Moreover, this new error is again mostly local and can efficiently further be reduced by relaxation. When relaxation again slows down, another transfer to the coarser grid is performed, and so on. Each coarse-grid problem is itself solved in a similar manner, that is, by a combination of relaxation sweeps (on that coarser grid) with transfers to a still coarser grid which supplies still longer-scale corrections. The method thus recursively uses a sequence of increasingly coarser grids. The coarsest grid includes just few unknowns, and its equations are therefore inexpensively solved by any method.

There is a number of ways to define at each stage the next coarser grid, and to formulate the residual problem on it. In case of two-dimensional rectangular grids, for example, the coarser grids are also rectangular, each typically obtained by omitting every other row and every other column from the previous grid, so that the number of unknowns is typically reduced by a factor of about 1:4. As for the equations on these coarser grids, in simple linear cases they look the same as the original equations, except that the right-hand side of each coarse-grid equation is formed as some accumulation (e.g., averaging) of neighboring residuals of the next finer grid. In non-linear cases another version (FAS) is used, but the linear case is enough to represent here all the interesting computer processes.

Thus, all in all, multi-level algorithms include relaxation processes on a sequence of grids, as well as fine-grid-to-coarse-grid transfers of residuals, and coarse-to-fine interpolations of corrections. This solution process is very efficient because information can propagate quickly on all levels.

In fact, for many (conjecturally for nearly all) local-relation problems the efficiency of suitable multi-level algorithms is essentially as high as it can conceivably be. Namely, the amount of computational work needed to solve the problem to its meaningful accuracy (e.g., to the accuracy of the discretization errors) is just a small (less than 10) multiple of the amount of arithmetic operations involved in simply writing out all the participating equations.

Moreover, concurrently with their role in obtaining fast solutions, multilevel processes can be very useful in many other ways. They provide very efficient, and completely local, grid adaptation capabilities, by allowing at any desired solution stage the addition of new finer local levels, each possibly with its own local

coordinate system. the inter-level interactions also supply natural criteria for deciding when and where further grid adaptation is actually needed. Additionally, the distinction between local processing and global corrections gives a straightforward way to overcome a conflict often encountered between accuracy and stability. Furthermore, in problems which have to be re-solved many times over for some changing data, multi-level techniques can most often save the bulk of reprocessing: For example, relaxation on the finer grids can often be limited to the neighborhood of the changed data. Continuation processes, for treating difficult nonlinear problems, can mainly be done at the coarser levels, with rare visits to finer ones. And so on.

The needed computer processes

We now summarize the kinds of computer processes needed in the multilevel solution of local-relation problems. The discussion will mainly be in terms of uniform grids (e.g., rectangular grids in two dimensions), which is the most common case.

From an implementation point of view, we can classify the processes into interior local processing, boundary local processing, inter-grid transfers, including perhaps grid transformations, and input-output and control operations.

The main computational effort is usually invested in *interior local processes*, especially on the finer grids. These include the relaxation sweeps as well as calculation of averaged residuals destined to be transfered to coarser grids. Fortunately, each of these processes can simultaneously be performed either at all gridpoints or at least at every other one (red-black relaxation): Although processing the equation at a gridpoint requires using solution values at some neighboring points, these values can be old ones, obtained at the previous iteration, hence full use can be made of systolic systems with as many (or at least half as many) parallel processors as gridpoints. Each processor should then have access to the results of several neighboring processors. Note that the notion of neighborhood, or the "*topology*", changes with the dimension d, and that $d = 3$ is likely to be the most important case. The topology is also strongly affected by the grid structure and by other, less fundamental factors, such as periodicity of boundary conditions and size of the discretization stencil.

On the other hand it does not seem generally possible to parallelize two different stages of the algorithm, such as the relaxation sweeps on two different grids: While the coarser grid operates on the fine-grid residuals, the latter must remain fixed.

The local processing needed near the *boundary* of a problem is often very different from the interior one. It tends to be much less systematic, much less uniform. The location of the boundary itself is not uniform, and the equations defined on it or near it may depend sensitively on its geometry. There usually appear many more external data and arithmetic operations at each boundary equation

that at any interior one. This greater complexity per equation is not of much concern when conventional computers are used, because the number of equations near the boundary is ordinarily much smaller than at the interior. With the high degree of parallelism mentioned above, however, the boundary processing may easily become the bottleneck, even though it is itself as parallelizable. Normally it is possible to relax on the boundary in parallel to the interior relaxation.

As explained above, to facilitate fast convergence of long-range interactions, local processing must be supplemented with *inter-grid transfers*. These include coarsening, refinement and shear transfers. It is often enough to have *full-coarsening* transfers, e.g., transfers between any rectangular grid and the grid obtained from it by omitting every other row and every other column. But sometimes *semi-coarsening* is also desired, i.e., a coarser grid obtained by omitting only every other row (or every other column, but not both). Or vice versa: Sometimes it is desired to create, and interact with, a new grid which is the *refinement* of a given grid, in the sense that between any two rows, and/or between any two columns, an additional line is introduced. Sometimes such a refinement is needed only locally, at some part (a proper subdomain) of the given grid.

The minimal processing capability required for implementing fast and sufficiently general local coordinate transformations is to have fast *shear transfers*. In a shear transfer all grid values, or part of them, are shifted in the same grid direction, e.g., to a new position in the same row. The distance each value is shifted is constant in each row, but may change from one row to another. Such shears, employed also in columns, and applied to the local refinement grids mentioned above, can efficiently provide for almost all desired discretization patterns.

It should be remarked that inter-grid transfers are inherently sequential to each other and to any of the local processing. Moreover, a transfer (coarsening, refinement or shear) is usually needed once per quite many (typically 10 to 100) local-processing operations. There therefore seems little to gain in parallelizing transfers with local processing.

The amount of *input and output* varies substantially. Most large-scale problems are "autonomous", i.e., the local equations at interior points include no external data, or the data are specified by a general rule, not by inputting lists with one or more item per gridpoint. Data lists are then only applied at the boundary, supplying geometrical information (location of the boundary) as well as boundary conditions (conditions the solution must satisfy at or near the boundary). Some problems, however, are not autonomous, requiring input data at every gridpoint.

Similarly, in many problems it is not required to output the full solution; of real interest are only some solution functionals, such as its values at some specific points, its integral along some curves, or its overall performance in some sense, its maxima, etc. Often, however, a user would like to have the whole solution displayed in a variety of ways.

Fast solvers are often employed because a sequence of many similar problems

need to be solved, with some data changes in between subsequent problems. The data change per problem is often limited to just few parameters, so the input per problem may be very small. As for the output, however, users may often like to see the full evolution of the sequence of solutions displayed as vividly as possible.

The *control* needed for the interior local processing is often very simple, since the same sequence of operations is usually needed at all gridpoints. A SIMD machine would then do. But the processing near the boundary is likely to be much less uniform, and at some particular points or curves (which may formally be all treated as boundary, even when some of them are in the interior) the execution of special routines may be required.

Add to all this the unpredictable, and you get some idea about the challenge which should be met by future hardware and software design.

Hardware development: Preliminary outlook

To solve many pressing scientific and technological problems, computational power of several orders of magnitude beyond present-day capacity is required. To obtain this power, very large scale parallelism is mandatory: The number of parallel processors should not in principle be bounded. An overall design is therefore needed which will be open-ended and scalable. Even though fewer processors may presently be projected, the overall system should not be saturated. It should be extendable, and not just in scale, not only in terms of storage size and processors number, but also in concept. That is, it should accommodate future variations in problems, in algorithms, and even in computer technologies. On the other hand the design should also allow full efficiency, i.e., taking full advantage of the inherent simplicity (the simple geometry and the SIMD nature) of interior local processing, matching it with compatible efficiency of boundary procressing and inter-scale transfers.

It seems indeed doubtful that a multitude of processors can be put to work effectively on local-relation problems without taking advantage of their local nature. Complete generality of multi processing is limited by the technical fact that each processor can be directly connected to only few others. Without a close relation between these hardware connections and the inter-processor transfers required by the algorithm, the whole operation will be jammed by the traffic. Another alternative of course is to be content with a lower degree of parallelism, e.g., with many gridpoints per processor ("coarse graining"), in which case the efficiency of inter-processor communications is less critical. We may use this alternative in our system, but we do not want the system to be for ever handicapped by this limitation.

Various architectures are proposed which can very effectively perform both local processing and inter-level transfers. These include for example processor arrays with nearest-neighbors plus perfect-shuffle connections; the pyramid-like

connected arrays, exactly corresponding to a two-dimensional multigrid structure; binary-tree arrangement of processors, which nicely corresponds to a possible multigrid hierarchy, and is useful in many other ways, but is not fully effective in performing local processing; etc. Each such design may be very useful for some problems, but by itself is too narrow: Its performance will be degraded very much for any problem with different topology, e.g., different dimension or different periodicity, or for any problem with excessive amount of processing near boundaries, or with too many levels of local refinement, or with different topology of intergrid transfers, or with another graining, or any other variation.

The design tentatively proposed here takes advantage not only of the vast parallelism inherent in every stage of the multi-level algorithms, but also of the *sequential* nature of these stages *with respect to each other*. The basic idea is that these different stages need not all use the same parallel-processing "engine". An open-ended *set* of engines should be constructed. One kind of engine will specially be suited for interior local processing, using for instance SIMD or systolic arrays, without being plagued by excessive processing near boundaries or by the need to implement inter-grid transfers. The latter can be carried out by other engines, special "transfer engines". Moreover, different transfer engines can be used in order to re-arrange grids in various ways, which effectively create the desired topology for the work of the interior-processing engine. A separate, presumably MIMD-type engine with fewer but more powerful processors, will perform boundary local processing (usually in parallel to the interior processing on the SIMD engine). On sufficiently coarse grids, lighter (i.e., with fewer processors) engines can be employed for both interior and boundary processing, freeing the heavier engines for concurrent utilization by other jobs.

Is all this possible? All that seems to be required as a basis for this design is a set of basic memories, with fast bulk rate of copying one into the other. Each of the engines can then be attached to one such memory. Thus, for example, an interior-relaxation SIMD engine can have one such memory serving it by being divided into "memory elements", each accessible to one or several of the engine's processing elements. Having processed in this memory a certain step of relaxation or residual calculation, its data can then be copied into the memory of a transfer engine which would rearrange the data. Depending on the transfer engine used, or on its program, this rearrangement may represent either an inter-grid transfer or some transformation between the topology of the problem and that of the interior-relaxation engine. In fact, with a suitable collection of transfer engines, an interior-relaxation engine can be used which have no topology at all, that is, its processing elements can be disconnected from each other, with each memory element being accessible to only one of them. The transfer engine should then fully represent the topology of the problem by transferring each computed result to *several* suitable new locations.

The required speed of copying these memories into each other seems to be fully within technological reach. A bulk rate of one gigabyte per second, which

is already produced today, can for example support an interior-relaxation engine with several giga-FLOPS. Much higher speeds are conceivable because of the large amount of data to be copied each time, the fully parallel nature of this operation, and the possible standardization of the basic memories. It is also conceivable to base some of the engines on *two* such basic memories, one being in a state of preparation (output then input) while the other is used by the engine.

The transfer engines can also attain the required speeds, since each of them can be wired to one particular mode of data rearrangement. All transfer operations of interest can be generated by a rather small collection of such engines, working sequentially to each other. If this collection is not rich enough, some complicated transfers may take longer, but the heavy relaxation engines need not wait: They can at the same time execute a relaxation step of another job.

An attractive feature of this overall design is its modularity. Different engines can be developed, including software, by different research groups. New engines can always be added, either heavier in processing power, with finer graining, or more specialized to some specific problems of particular interest. Various "display engines" can be constructed. One of them, for example, can directly display the values in its memory as greyness levels and colors on a screen, so that repeated copying into this memory would create a movie history of an evolving solution. And so on.

Moreover, engines can be added which serve purposes other than those described above: Fast Fourier transforms, sorting and other data processing, matrix multiplication, elimination, etc. some of these operations can very efficiently be programmed on the same systolic engine used for interior relaxation. This whole project can thus become part of a more general development, combining the simplicity and high efficiency of SIMD and special transfer engines, with the versatility of MIMD machines in a generalized, open-ended and scalable environment, which may be called a "*parallel processing mill*". To establish such mills all that seems to be needed is to decide on some common specifications for the basic set of memories, and then to coordinate the manufacturing of various engines by different research groups and, eventually, companies.