General Application Issues

Geoffrey Fox Indiana University

In the first and last chapters of this book, we have described how parallel computers and large-scale simulations have and will have profound impact on many fields. Here we assume this motivation and in the application section of the book (Chapters 4 through 8) try to answer the following question:

I have an application – can and should it be parallelized and if so, how should this be done and what are appropriate target hardware architectures; what is known about clever algorithms and what are recommended software technologies?

Most of the answers to these questions are implicitly described in other parts of this book and we attempt to aid the reader's identification of where to go by combining an exposition of general principles with several case studies. The latter consists of in depth discussions in Chapters 5, 6 and 7 of computational fluid dynamics, energy and environmental studies and chemistry. We also present in Chapter 8 a set of 11 short discussions of applications illustrating interesting features of their computational structure. This is followed by an overview of all the applications here plus those in two other books in Section 8.12. This overview discusses algorithmic and software issues in each case, and is a possible resource if one wishes to find applications exhibiting particular computational features. We also highlight Chapter 16, which has a pedagogical description of Poisson's equation with message passing, HPF and OpenMP programming models. This is not a "real" application like the others in our selection but acts as a simple prototype for discussing general issues.

So we now step through the thought processes involved in analyzing a given application and in this way illustrate certain general characteristics that are useful in classifying the issues involved in parallelizing general applications. We first review the same Poisson equation of Chapter 16 and revisit the discussions of Chapters 3 and 9 from an application rather than a parallel programming perspective

4.1 Application Characteristics in a Simple Example

Simple 2D electrostatic problems can be reduced to solving Laplace's or Poisson's equation and as described in Chapter 16, this is often solved numerically by finite difference methods. These could involve adaptive meshes and hierarchical multigrid methods but in the simplest formulation are set up as a regular grid of field values where the basic iterative update links two dimensional nearest neighbors as in Figure 4.1.

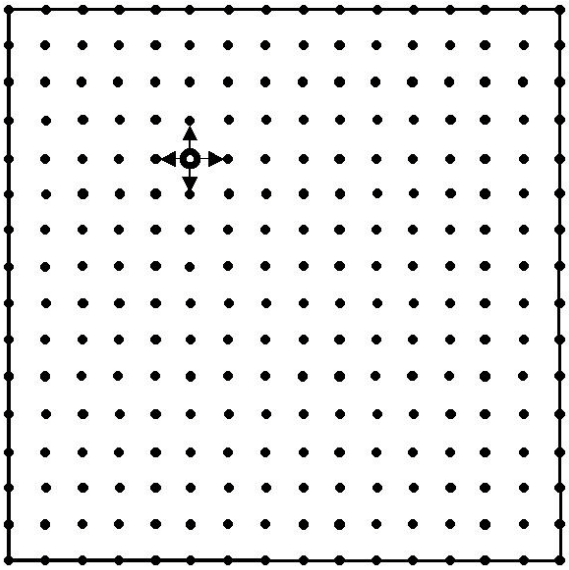

Figure 4.1: 16 by 16 2-D Mesh with an illustration of basic nearest neighbor update used in Jacobi's method of Chapter 16

If the points are labeled by an index pair *(i,j)*, then Jacobi's method defined precisely in equation. (4.1), can be written

$$\phi_{New}(i,j) = (\phi_{Left} + \phi_{Right} + \phi_{Up} + \phi_{Down}) / 4 \qquad (4.1)$$

corresponding to the stencil given below where the subscript *Left* corresponds to index pair *(i-1,j)* etc.

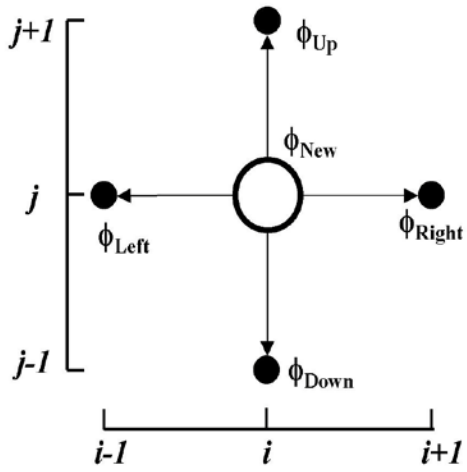

Figure 4.2: Stencil for Jacobi Iteration of Figure 4.1

Such problems would usually be parallelized by a technique that is often misleadingly called "domain decomposition" or "data parallelism". As these terms already have particular meaning in the algorithm and parallel programming fields respectively, we can use the term *block data decomposition*, which is essentially the nomenclature used in HPF. In this problem, parallelism is achieved by exploiting the feature that we can view the problem as an algorithm (4.1) applied to a set of data points. Parallelism is naturally found for such problems by dividing the domain up into parts and assigning each part to a

different processors as seen in Figure 4.3. For problems coming from nature this geometric view is intuitive as say in a weather simulation, the atmosphere over California evolves independently from that over Indiana and so can be simulated on separate processors. This is only true for short time extrapolations – eventually information flows between these sites and their dynamics are mixed. Of course it is the communication of data between the processors (either directly in a distributed memory or implicitly in a shared memory) that implements this eventual mixing.

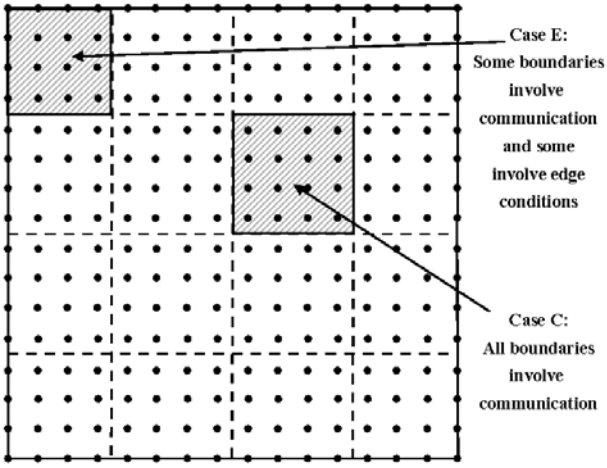

Figure 4.3 16 by 16 Mesh Divided among 16 Processors with a 2-D grid chopped into rectangular sub-domains

Block data decompositions typically lead to a SPMD (Single Program Multiple Data) structure with each processor executing the same code but on different data points and with differing boundary conditions. In this figure, the "edge" (case E) processor has a mix of conventional and communication boundaries while in the "general" case C, a processor is "in charge" of 16 points with communication at the four edges providing the necessary boundary data. As described in Chapter 16, one often uses a set of halo grid points seen in Figure 4.4, to represent these communicated values. The term ghost is often used instead of halo. This type of data decomposition implies the so-called "owner's-compute" rule. Here we imagine each data point as being owned by the processor to which the decomposition assigns it. The owner of a given data-point is then responsible for performing the computation that "updates" its corresponding data values. This produces a common scenario where parallel program consists of a loop over iterations divided into compute-communicate phases:

- *Communicate:* At the start of each iteration, first communicate any outside data values needed to update the data values at points owned by this processor.
- *Compute:* Perform update of data values with each processor operating without need to further synchronize with other machines.

This general structure is preserved even in many complex physical simulations with fixed albeit irregular decompositions. Dynamic decompositions introduce a further step where data values are migrated between processors to ensure load balance but this is usually still followed by similar communicate-compute phases. The communication phase naturally synchronizes the operation of the parallel processors and provides an efficient barrier point which naturally scales. The above discussion uses a terminology natural for distributed memory hardware or message passing programming models. With a shared

memory model like OpenMP, communication would be implicit and the "communication phase" above would be implemented as a barrier synchronization.

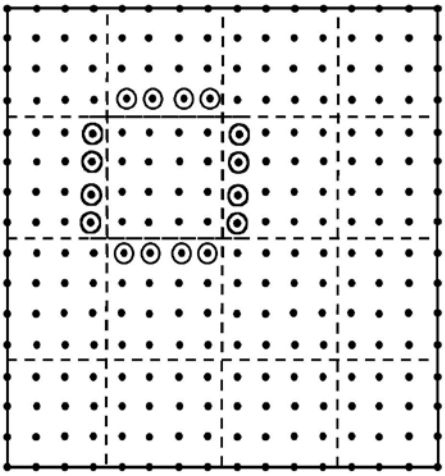

Figure 4.4: Communication Structure for Poisson equation Example. The circled communicated points form the halo or ghost grid points.

4.2 Communication Structure in Jacobi's Method for Poisson's Equation

On a distributed memory machine, the geometrically local structure of the linked entities of Figure 4.1, leads to a classic communication structure with the communication volume proportional to surface area (in 2D this is the sides of the rectangle) of each sub-domain while computation is proportional to volume. Further in this case and most such cases, one can usually "block" the communication to transmit all the needed points in a few messages. Chapters 3, 7 and 10 explain why this is important to reduce effect of latency of messaging system. We can use our current Poisson equation example to produce some good rules of thumb to allow estimates of the performance of many parallel programs.

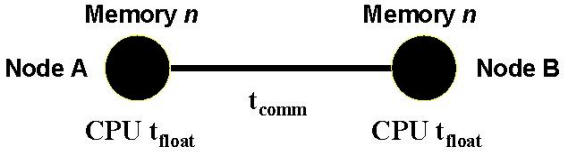

Figure 4.5: Parameters determining performance of loosely synchronous problems.

As shown in Figure 4.5, we characterize the node of a parallel machine by a parameter t_{float}, which is time taken for a single floating point operation. t_{float} is of course not very well defined as depends on the effectiveness of cache, possible use of fused multiply-add and other issues. This implies that this measure will have some application dependence reflecting the goodness of the match of the problem to the node architecture. We let n be the grain size – the number of data locations owned by a typical processor. n is 16 in the trivial example of Figure 4.3 but in a realistic example would be larger and limited by the memory of each processor. For a hypothetical 10^3 by 10^3 by 10^3 3D mesh solved on a 1000 processor machine, n would of course be 10^6. Communication performance – whether through a shared or distributed memory architecture – can be parameterized as

$$\textit{Time to communicate } N_{comm} \textit{ words} = t_{latency} + N_{comm}\, t_{comm}$$

This ignores issues like bus or switch contention but is a reasonable model in most cases. It is dangerous to quote explicit values for these parameters as hardware is always

improving. Very roughly $t_{latency}$ has a value around $1\mu s$ on shared memory machines while it is at least an order of magnitude higher, say $40\mu s$ between remote nodes on distributed memory architectures. This latency becomes 10-100ms between of a geographically distributed metacomputer; this drastic increase in latency explains why one cannot easily use such systems for parallel computing and we return to this point in Section 4.8. t_{comm} is time to communicate a single word and this is in range 0.1 to $0.01\mu s$ per word. For large enough messages (N_{comm} in range from 100 to 1000 or larger), the latency term can be ignored and we will set $t_{latency}=0$ in the following. We can generalize the above problem to N_{proc} processors arranged in an $\sqrt{N_{proc}}$ by $\sqrt{N_{proc}}$ grid with a total of N grid points and the grain size $n = N/N_{proc}$.

Then first considering load balance, we can write the sequential execution time

$T(1) = (\sqrt{N} - 2)^2 t_{calc}$, where this notes that boundary points are fixed and only the $(\sqrt{N} - 2)$ by $(\sqrt{N} - 2)$ array of internal points need to be updated. Further

$t_{calc} = 4 t_{float}$ is time to execute the basic update equation *(4.1)*

The parallel execution time is governed the "interior" processors with n points to be updated. Thus

$T(N_{proc}) = n t_{calc}$, and the speed-up is given by

$$S(N_{proc}) = T(1) / T(N_{proc}) = N_{proc} (1 - 2/(nN_{proc})^{1/2})^2$$

The speedup $S(N_{proc})$ is less than N_{proc} because not all the processors update the same number of points. However this load imbalance is a small edge effect decreasing when either n or N_{proc} becomes large.

More important in this case is the communication overhead, which adds to $T(N_{proc})$ a term $4\sqrt{n}\, t_{comm}$ illustrated in Figure 4.6 for the cases $n = 16$ and $n = 64$. The communication term is an edge effect proportional to $\sqrt{n}$ which decreases in importance as n increases compared with the computation term $4n\, t_{float}$. More precisely, we now find the full speed up formula:

$$S(N_{proc}) = N_{proc} (1 - 2/(nN_{proc})^{1/2})^2 / (1 + t_{comm}/(\sqrt{n}\, t_{float})) \qquad (4.2)$$

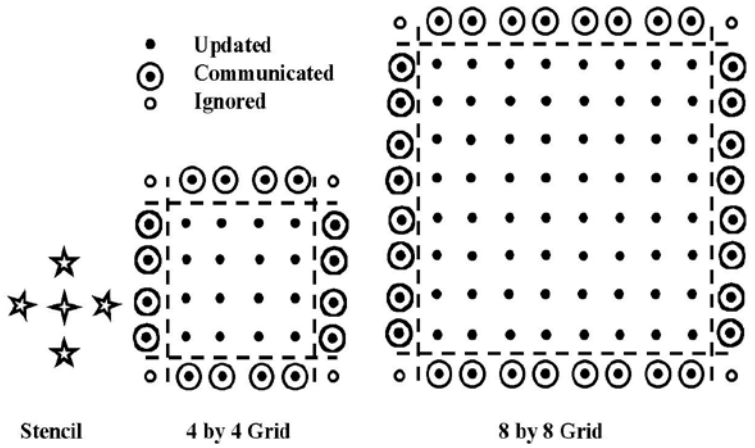

Figure 4.6: Communication Structure for a five-point stencil and two different grain sizes

Realistic values for t_{comm}/t_{float} are in range of 10 to 100, and so the communication overhead dominates in equation *(4.2)*. For an intermediate value $t_{comm}/ t_{float} = 50$, we need the grain size n to be greater than 250,000 grid points to reduce communication overhead below 0.1. We note that this analysis ignores the possibility available on some computers

of overlapping communication and computation. One can straightforwardly extend the analysis to include the effect of such strategies for performance enhancement. We can generalize the above formalism most conveniently using the notation that

$$S(N_{proc}) = \varepsilon N_{proc} = N_{proc}/(1 + f), \qquad (4.3)$$

which defines efficiency ε and overhead f. The communication part f_{comm} of the overhead f is given in equation *(4.4)* as

$$f_{comm} = t_{comm}/(\sqrt{n}\, t_{float}) \qquad (4.4)$$

Note that in many instances, f_{comm} can be thought of as simply the ratio of parallel communication to parallel computation. This equation can be generalized to essentially all problems we will later term loosely synchronous. Then in each coupled communicate-compute phases of such problems, one finds that the overhead takes the form:

$$f_{comm} = constant \,.\, t_{comm}/(n^{1/d}\, t_{float}) \qquad (4.5)$$

Here d is an appropriate (complexity or information) dimension, which is equal to the geometric dimension for partial differential based equations or other geometrically local algorithms. The most important case in practice is the 3D value $d=3$ when $n^{1/d}$ is just surface/volume in three dimensions. For full matrix problems, one finds the value $d=2$ for the best decompositions such as those used in SCALAPACK described in Chapter 20. Applying equation *(4.3)*, we find that $S(N_{proc})$ increases linearly with N_{proc} as long as N_{proc} is increased with fixed f_{comm} which implies fixed grain size n, while t_{comm} and t_{float} are naturally fixed. This is scaled speedup where the problem size $N = n\, N_{proc}$ also increases linearly with N_{proc}.

The continuing success of parallel computing even on very large machines can be considered as a consequence of equations *(4.3)* and *(4.5)*. Note that the formula for f_{comm} (whose numerical value we could aim to keep around 10% or lower) only depends on local node parameters and not on the number of processors. Here we consider the grain size n as reflecting the amount of local memory. Thus as we scale up the number of processors keeping the node hardware and application size n fixed, we will get scaling performance – speedup proportional to N_{proc}.

This simple problem is perhaps the one where the parallel issues are most obvious; however it is not the one where the parallel performance is easiest to obtain as the small computation load of the update equation *(4.1)* makes the communication overhead relatively more important. There is a fortunate general rule that as one increases the complexity of a problem, the computation needed grows faster than the communication overhead and we will illustrate this below. Jacobi iteration does have perhaps the smallest communication for problems of this class. However it has one of largest ratios of communication to computation and correspondingly high parallel overhead.

Note one sees the same effect on a hierarchical (cache) memory machine, where problems such as Jacobi Iteration for simple equations can perform poorly as the number of operations performed on each word fetched into cache is proportional to number of links per entity and this is small (four in the 2-D mesh considered above) for this problem class.

4.3 Communication Overheads for More General Update Stencils

It is instructive to consider in detail how the above analysis is altered as one changes the update formula of equation (4.1). Consider first using fourth order differencing to

approximate ∇^2 in Poisson's equation. Then as illustrated in Figure 4.7, we need to communicate twice as many points into halo cells. However the overhead f_{comm} is not changed significantly from its value of equation (4.4) for the computation needed to update each point is also doubled and the ratio of communication to computation is roughly unchanged.

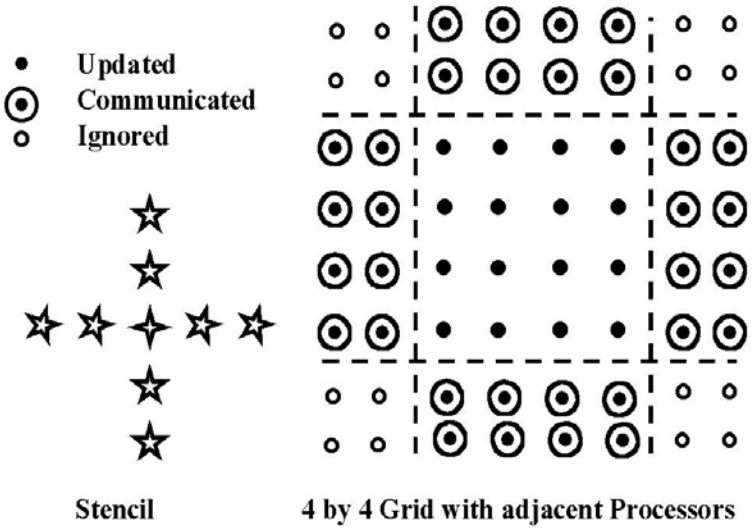

Figure 4.7: Communication Structure for nine-point stencil

We can now systematically increase the size of the stencil and find how f_{comm} changes. In the case explained below, where the grid points are replaced by particles, this corresponds to ratcheting up the range of force between the particles.

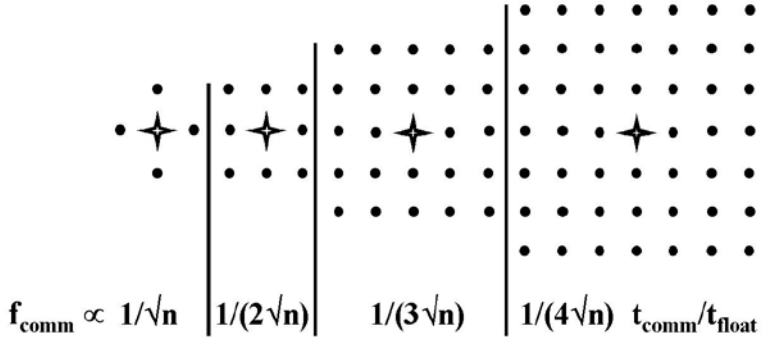

Figure 4.8: Communication Structure as a function of stencil size. The figure shows 4 stencils with from left to right, range l = 1,1,2,3.

We find that the communication overhead decreases systematically as shown in Figure 4.8 as the range of the force increases. For a range of l (measured in units of grid spacings), one finds in this 2D case,

$$f_{comm} \propto t_{comm}/(l\ \sqrt{n}\ t_{float})$$

This is valid for l which is large compared to 1 but smaller than the length scale corresponding to region stored in each processor. In the interesting limit of an infinite range ($l \to \infty$) force, the analysis needs to be redone and one finds a result that is independent of the geometric dimension

$$f_{comm} \propto t_{comm}/(n\ t_{float}) \qquad (4.6)$$

which is of the general form of equation (4.5) with complexity dimension *d=1*. This is the best-understood case where the geometric and complexity dimensions are different. The overhead formula of equation (4.6) corresponds to the computationally intense $O(N^2)$ algorithms for evolving N-body problems. The amount of computation is so large that the ratio of communication to computation is extremely small. This observation is at the heart of the success of special purpose machines such as GRAPE from the University of Tokyo [http://grape.c.u-tokyo.ac.jp/grape/]. The one teraflop GRAPE 4 won the Gordon Bell prize twice and the GRAPE 5 took the cost effectiveness award in 1999 (at $7 per megaflop). The 100 teraflop GRAPE 6 was completed in 2000 and won another Gordon Bell award! The modest memory and communication needs of the *N* body problem are some of the reasons enabling these powerful machines, which outperform on this problem any of the more general-purpose parallel computers. Of course the specialized GRAPE architecture limits the problems to which it is applicable.

4.4 Applications as Basic Complex Systems

Above we already showed how one could discuss the parallel issues for several different problems (here particle dynamics and local discretization for partial differential equations). This is generally true as the parallel issues depend not on the detailed science or numeric algorithm but on overall characteristics of the application.

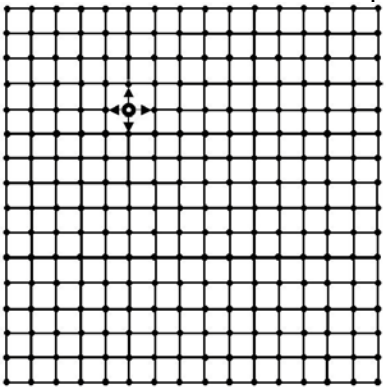

Figure 4.9: A Simple basic complex system with a set of entities with nearest neighbor linkage to at most four others.

So it makes sense to generalize the above discussion in terms of both general principles applicable to very many parallel computing problems and special features of the particular two-dimensional structure seen in Poisson's equation. It is useful to think of an application as a "complex system" or a linked set of entities and this way of thinking can relate the parallelelization strategy of seemingly very different problems. In particular, many other applications have a similar computational structures to the Laplace or Poisson equation of the previous sections.

Consider first the 2D Ising Model, where the mesh of Figure *4.9* is now not an array of discretized field values but a fixed grid of spins with a nearest neighbor connection for the interaction (forces) between them. The Ising model has a similar geometric structure to equation (4.1) but the physics and numerical procedure show many differences from Poisson's equation. The grid points in the Ising model are physically real spins where as in the Poisson case, the grid points are artifacts of the numerical procedure. The nearest

neighbor local connection in the Ising case corresponds to a physical force law while it follows from the differencing approximation to a partial derivative in Poisson's case. Further the usual numerical approach to the Ising Model uses a Monte Carlo method rather than a differential equation to express the dynamics of the system. Further the iterative process is not a perturbative solution to an exact Matrix problem as in Poisson's equation. Rather the iterator counts Monte Carlo sweeps as one accumulates integration points to decrease the statistical error, which is inversely proportional to the square root of the number of sweeps. However these differences which are so important to the underlying science do not affect very much many of the issues that come up in discussing appropriate parallelization strategies and the needed hardware and software systems. Even closer to our Poisson equation would be an application the solved a simple wave equation (or Maxwell's equations) in a two dimensional domain. Here we see an identical computational structure with the perturbative iteration in the sparse matrix solution replaced by stepping through a discretized time variable. Yet another rather similar structure can be found in cellular automata problems.

We can extend this very simple problem is several ways and some of these are explored in Chapter 16. For instance finite element problems have a similar mesh that can be quite irregular (compared to the uniform geometry of most finite difference problems) and this brings load balancing to the fore as an important issue. Particle dynamics problems with a short-range force can exhibit similar structure as discussed for Figure (4.8) but with a dynamic irregular structure and a variable number of links per entity. An obvious and important generalization of the Poisson structure is to higher dimensions with three and even four-dimensional structures. Here we have already stated in equation (4.5), a general form for the communication overhead. This equation applied for $d=3$ shows an overhead decreasing like $n^{1/3}$ – slower than for the 2D case discussed in detail above. This suggests values like $n \approx 10^6$ are needed in 3D in order to match as good a performance as the much smaller $n \approx 10^4$ in 2D. Note that in Chapter 8, we describe two physics examples – numerical relativity and computational QCD where the basic mesh is four-dimensional. The many partial differential equation applications in Chapters 5, 6, 7 and 8 also always have a richer structure at each grid or finite element mesh point than the single value of Poisson's equation. For instance QCD has 3 by 3 complex matrices representing "gluons" and vectors representing quarks. Computational fluid dynamics is usuallyformulated in terms of 5 degrees of freedom at each point. In these examples there is much more computation involved in the basic update replacing equation (4.1) but as we have explained that actually tends to reduce the parallel overheads as communication tend to scale like the number of degrees of freedom at each point. The computational update time complexity per point usually increases faster than this.

So we have seen that is helpful to consider many problems as linked entities arranged in one, two three or higher dimensional geometries. This linkage was "short-range" (a few links per entity) in the examples we discussed but one of course finds examples that span the gamut of possibilities. Particles interacting by a long-range gravitational force illustrate a case with many links per entity. This example using the simple $O(N^2_{particle})$ algorithm discussed at the end of Section 4.3 has very different properties from the short-range case. In particular the performance of this problem is excellent on both distributed and hierarchical memory machines. There are many (of

order $N_{particle}$) computations for any point stored in cache and even though the communication appears heavy in a distributed memory machine, the above analysis shows a low ratio of communication to computation. This type of long-range problem is found in a variety of fields, which are far from particle dynamics but still have the same computational structure. We give one interesting example of an $O(N^2_{particle})$ algorithm in Chapter 8 – the Greens function approach to the simulation of earthquakes. Such partial differential equation solvers become integral equations over the domain boundaries with the full linkage between the element mesh on the boundary. Some applications involving determination of correlation function also have this fully connected structure between the points in the computation.

The N body example can be used to illustrate another important point. Namely a given physical problem can look quite different depending on the numerical formulation. The natural $O(N^2_{particle})$ algorithm is often not the best approach to the simulation of gravitating particles, and for large problems, one usually adopts the so called fast multipole method with $O(N_{particle})$ or $O(N_{particle}\log N_{particle})$) behavior. This again shows that one needs to choose parallel algorithms carefully; the lowest communication or even the lowest communication to calculation ratio may not be the best choice. A simpler application illustrating the same issue is Poisson's equation, which can often be solved by either iterative local methods such as Jacobi or conjugate gradient, or by the FFT (Fast Fourier Transform). In both cases the obvious approach has a simpler complex system structure while the fast algorithm has a more complicated tree structure. This emphasizes that a computational scientists use their skill to convert a given application into a numerical system and it is the structure of the latter that determines the key parallel computing issues.

4.5 Time Stepped and Event Driven Simulations

Above we have noted the rich spatial or geometric structure of applications. Two rather distinct simulation methods, time-stepped and event-driven, correspond to different temporal structures. Most of the examples in this book correspond to the time stepped case where the entities in a complex system are evolved together and synchronized globally either by the concept of time or something essentially equivalent like an iteration or Monte Carlo sweep. This is of course very reasonable, as it is "how nature works". In the early days of parallel computing, there were concerns that the global synchronization implied by the time-stepped approach would lead to uncontrollable overheads. This is not true, for it can be seen that as described at the end of Section 4.1 for the simplest nearest neighbor Poisson equation, global time synchronization is implied by the local synchronization of neighboring nodes, either by exchanging messages or the equivalent shared memory mechanism. This synchronization mechanism is itself fully parallel (with no "hotspots" in proper implementations) and so introduces no serious parallel computing overheads. Such efficient synchronization is present in all such problems where there is a time or iteration count to provide algorithmic synchronization. Then correct implementation of such an algorithm with natural synchronization points, implies that the parallel program needs no special additional synchronization. Message passing systems like MPI naturally have such synchronization barriers built in but in other programming models (such as active messages and OpenMP) require explicit user attention to this issue.

The military makes substantial use of event-driven simulations in the field of Forces Modeling and we give an example if this in Section 8.11. Here one tends not to simulate systems in terms of their fundamental constructs (atoms, grid-points etc.) but rather in terms of macroscopic constructs such as vehicles, mines, battalions etc, in the war gaming example. These system components are naturally formulated in terms of objects interacting with events, which are queued (often in a distributed fashion) and executed either in real time (the natural case when there is "hardware in the loop") or according to a global virtual time. Here we do find potentially serious problems with the overhead of global synchronization and very ingenious techniques have been developed. One important strategy – termed Time Warp – involves simulating the system in terms of interacting time stamped events. Block data decomposition is typically used for parallelism just as in the synchronous and loosely synchronous case. Now however there are no straightforward ways to ensure all events have been received and so unambiguously decide to let the simulation proceed in any one processor. The Time Warp approach optimistically proceeds with the simulation marching forward in time in each processor with whatever events are available. Correctness is guaranteed by recording system state from time to time and “rolling back’ to this old (correct) state if an event arrives with an earlier time stamp than current processor simulation time. The particular minefield simulation CMS application described in Section 8.11 was successfully parallelized because the different entities in the simulation are largely independent and so there was essentially no synchronization difficulties.

Currently one of the most powerful parallel event-driven approaches is the SPEEDES system from Metron Corporation discussed in Section 8.11 and there are overall frameworks like HLA and RTI defined for this field. HLA (High Level Architecture) and RTI (Run Time Infrastructure) are object models similar to those described in Chapter 13. However no software system for event driven simulations enjoys the universal acceptance and relatively clear methodology for getting good performance shown by MPI in the time-stepped case. Some recent work at Los Alamos National Laboratory [http://www.lanl.gov/orgs/d/simulation.shtml] is potentially of great importance. These researchers have shown that some applications traditionally approached by event-driven simulations (such as a large scale traffic models) can be tackled as loosely synchronous problems with excellent scaling parallel performance.

Circuit simulation is an interesting application area, which can be tackled by either simulation technique. Obviously a circuit has a natural time, which can be iterated over with at each step every device component being updated. This approach can be inefficient as on most of iterations, only a tiny fraction of the components are active. The event-driven approach can then be the most effective approach to circuit simulations as one automatically only updates those devices affected by queued events. This analysis is clear for sequential machines but the difficult parallelism of event-based systems makes the parallel situation less clear.

4.6 Temporal Structure of Applications

It has been found useful to divide the temporal structure of numerical systems into four broad areas

- **Synchronous:** Here each point can be evolved in synchronous mode as is natural on a SIMD machine. The temporal synchronization is on a point-by-point basis. Most of the simple examples discussed above are of this type.
- **Loosely Synchronous:** Here the temporal synchronization is on a sub-domain basis and this is the natural form of SPMD (Single Program Multiple Data) implementations such are all HPF and most MPI programs. This is the dominant case for today's major applications as essentially any serious geometrical or other irregularity converts a problem, which is in its simplest mode synchronous to loosely synchronous form. In particular finite element problems, or finite difference codes with adaptive meshes are loosely synchronous. Domain decomposition in Chapter 6 has this structure, as does the fast multipole approach to particle dynamics discussed earlier. The simple $O(N^2_{particle})$ particle dynamics algorithm is however synchronous.
- **Asynchronous:** Event driven simulations fall into class, which include those problems, which are not formulated in terms of a stepped, time or iterator associated with each system entity. As discussed above, asynchronous problems can be very hard to parallelize whereas in principle loosely synchronous applications always run efficiently if they are large enough.
- **Pleasingly Parallel:** The time or iteration evolution structure of a problem can impact greatly the appropriate software and hardware architecture. However there is one important special case where this is not true -- namely cases where the entities in the system are essentially disconnected. Then each entity can be evolved more or less separately and there is no significant synchronization overhead whatever the temporal differences between the entities. One typically uses a "farm" architecture with worker nodes somehow getting given chunks of the simulation (entities) to do as they finish their previous assignment. This has very non-trivial application dependent implementation issues but such problems will always parallelize well if the problem is large enough. Good examples of this problem class come from the Internet where both large web servers and the backend of database search engines such as Inktomi and Google are of this type. Note this problem class was often termed "embarrassingly parallel" in the past.

4.7 Summary of Parallelization of Basic Complex Systems

So let us take stock of where we are. Problems are set up as computational or numerical systems and we have discussed one set of such systems, which consist of a space of linked entities. These we termed "basic complex systems" and characterized them by their possibly dynamic spatial (geometric) and temporal structure. We have noted the difference between the structure of the original problem and that of computational system derived from it. We can summarize much of the past experience in parallelizing applications by the conclusion

Synchronous and Loosely Synchronous problems perform well on large parallel machines as long as the problem is large enough. For a given machine, there is a typical sub-domain size (i.e. the grain size or size of that part of the problem stored on each node) above which one can expect to get good performance. There will be a roughly constant ratio of parallel speedup to N_{proc} if one scales the problem with fixed sub-domain size and total size proportional to N_{proc}.

Unfortunately although this assertion is probably true in most important cases, it has proven very difficult to design and implement productive programming environments that allow the user to realize this goal. That is why we need to write this book even though in principle success is often guaranteed …….

4.8 Metaproblems

Several applications can be solely discussed in terms of computational systems, which fall into the basic complex system type discussed above. However this description is often incomplete although it does properly describe key computational modules that are part if not all of the complete application. More generally, one finds metaproblems, which are built up from multiple modules that each can be classified as basic complex systems. Such metaproblems are particularly interesting today, as many of them are the natural applications for distributed systems such as computational grids. One tends to run basic complex systems on classic shared or distributed memory machines as these have the required low latency and high bandwidth communication. Separate modules in a metaproblem can often be run on geographically separated machines, as they tend to have much less stringent communication requirements that those needed in the simulation of basic complex systems. Important examples of metaproblems are:

- The 3-way linkage of data store, simulation and visualization subsystems forms one of the most generic metaproblems, which is seen in many different disciplines. Section 8.10 describes an application of this type with their synchrotron light source.
- Multidisciplinary Applications**:** As discussed in Chapter 22, there is a growing trend in modern engineering to sophisticated system-wide optimization. For aircraft design, one might simultaneously optimize over fluid flow, structural, acoustic and electromagnetic properties. Each of these corresponds to a separate module in the discussion above. The DoD initiative in SBA (Simulation Based Acquisition – Section 8.11) would need such metaproblems and we illustrate this type of application in Figure *4.10*.
- An early success of the CASA gigabit network was the simulation of a coupled ocean-atmosphere metaproblem and there is general understanding that such approaches are essential for reliable long-range climate forecasts.
- The forces modeling community often builds such metaproblems where each component is a separate focused simulation. In an example given in Section 8.11, one simulation engine is used to describe mine fields and another describes squads of vehicles. You can imagine that these simulations have interesting interactions. In this field, metaproblems are called federations and the basic simulations are termed federates. As mentioned above, this community has recently adopted sophisticated software standards (RTI for Run Time Infrastructure and HLA for High Level (object) Architecture) to support the federation of multiple event driven simulations.

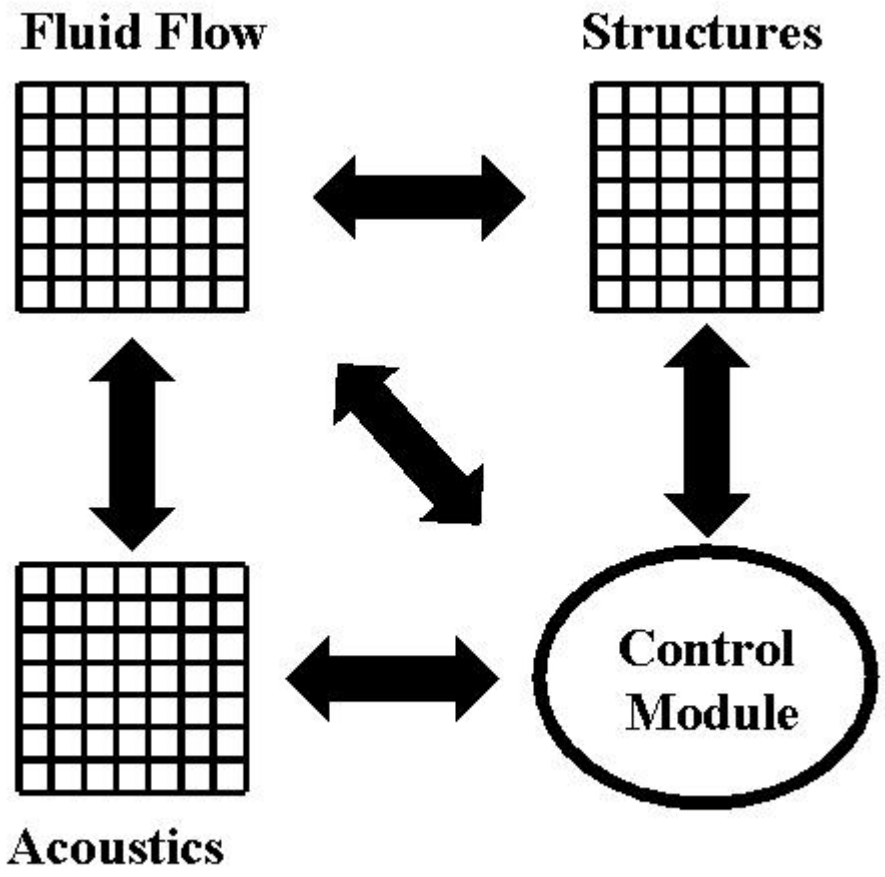

Figure 4.10. The linked modules in a typical metaproblem. We show three large-scale parallel modules which can be expected to execute individually on massively parallel systems. The control module is logically separate and may not need high performance computing.

Note that basic complex systems often have huge potential for parallelism with a complex 3D simulation perhaps exhibiting a billion independent degrees of freedom, which are candidates for data parallel systems. Metaproblems are different as one typically has but a few independent modules and further the linkage of these modules is often timed asynchronously and so naturally supported by different software concepts than the data parallel subcomponents. So this way one finds a metaproblem with each module using internally MPI, OpenMP, HPF or equivalent while the modules are linked together thorough channels using perhaps GridFTP (high performance Grid standard), Web Services and SOAP (W3C distributed object and message model), IIOP (CORBA) or RMI (Java). We discuss these different software models more completely in terms of object-based approaches and problem solving environments in Chapters 13 and 14.

4.9 Conclusion

At the start of this chapter, we posed the problem of understanding the principles governing the type of applications that could be parallelized. We addressed this by first

identifying basic (or “atomic”) complex systems. We discussed their parallelism in terms of their spatial and temporal structure which we summarized in Section 4.7 in terms of the application characteristics that govern the parallelism. The majority of large scale scientific and engineering codes can be parallelized. We illustrated these conclusions with examples and a simple performance model given in the earlier sections of this chapter. In the last section 4.8, we introduced metaproblems as the general application class defined in terms of loosely coupled aggregates of basic complex systems. We noted that this type of application was naturally suitable for distributed Grid architectures. This rather simplified discussion is complemented by the analysis of Section 8.12, which looks at some 50 particular applications and summarizes their computational structure. Chapters 5, 6 and 7 and sections 8.1 to 8.11 describe 14 application areas in detail.