Complex Systems Research on Parallel Computers

Russell K. Standish
High Performance Computing Support Unit
University of New South Wales
Sydney, 2052
Australia
R.Standish@unsw.edu.au
http://parallel.hpc.unsw.edu.au/rks

August 10, 1999

Abstract

Complex Systems research is a multidisciplinary field covering the gamut from Physics, Mathematics, Computer Science, Biology to Economics. Whilst there is no good agreement on what the term "Complexity" might mean, there are a number of threads or concepts linking the different disciplines together into a coherent field of research. Traditional mathematical techniques such as Analysis, Linear Algebra and Differential Equations have been largely ineffective at tackling the problems in Complex Systems; it is only in the last twenty years or so that computer modeling has been able to shed light in this field of research.

The field is maturing, and the complex systems models are getting more sophisticated. It is now becoming apparent that high performance computing and in particular parallel computing will need to be exploited in the next generation of models. Unfortunately, the structure of typical complex systems models have acted as an impediment to effctive expoitation of their inherent parallelism. This paper discusses some of the issues, and suggests a way forward.

1 Complex Systems

The idea of what is a complex system is not well defined, however people intuitively feel that a biological organism is complex, whereas a cup of water is not. There are a number of themes which run through studies of complex systems, one of the more important being that of emergence. Whilst the notion of emergence is as notoriously difficult to pin down rigourously as the concept

of complexity, it refers to the idea that a system's properties arise (or emerge) out of the interactions of the components of that system, rather than being a property of the components themselves i.e. "the whole is more than the sum of its parts". Examples of systems studied include: ecology, evolution, economics, artificial life, genetic algorithms, neural networks, cellular automata and agent-based modeling.

Cellular automata is a perhaps the clearest example application of parallel computing. As the interactions are between nearest neighbours, and are typically homogeneous, traditional data parallel techniques can be used to great effect. Even inhomogenous CAs can be efficiently computed with MIMD computational model. The natural extension to cellular automata is that of the *random boolean network*[10]. In this the coupling between lattice sites occurs randomly throughout the entire lattice, destroying data locality. One of the most prominent RBNs is Kauffman's NK model[2], which has been applied to a large number of different situations.

The idea that network structure is one of the prime determinants of complex systems properties is well argued in Green[1]. Unfortunately, this network structure is vary rarely locally connected, giving rise to models that are dominated by communication costs on parallel computers. Frequently the situation is so bad, that the model runs fastest on a single processor, particularly if it can fit within the cache. This is in spite of a high degree of parallelism in the underlying algorithm.

In this paper I look at two different complex systems modeling systems in particular to illustrate the sorts of difficulties faced in using parallel computers for this type of modeling.

2 Ecolab

The Ecolab model was first presented in Standish[8]. It consists of a first order differential equation in a n_{sp} dimensional space:

$$\dot{n}_i = \alpha_i n_i + \sum_{j=0}^{n_{\mathrm{sp}}} \beta_{ij} n_i n_j \tag{1}$$

where i ranges over $0 \ldots n_{\mathrm{sp}}$ (the number of species in the ecology). One of the key features of the system is that n_{sp} varies over time, due to introduction of new species to the system, and extinction of others. This is not especially germane to the current discussion, as the most time consuming portion of the calculation is performing the sparse matrix-vector multiplication $\sum_j \beta_{ij} n_j$. However, it does rule out any optimisation based on fixed communication patterns (e.g. Connection Machine's *Communication Compiler*), as the pattern of nonzeros in $\boldsymbol{\beta}$ will have changed before the optimisation costs have been fully amortised. Sparse matrix-vector multiplication is a perfectly parallel operation if the matrix

n	SGI	CM5
10^2	0.27	1.52
10^3	1.08	4.97
10^4	3.73	9.79
10^5	6.36	15.6
10^6		17.6

Notes:

1. SGI Power Challenge f90 compiler using auto parallelisation over 10 threads, vs single thread performance.
2. CM Fortran over 64 nodes. Scalar comparison time done by using `-node` argument, which runs an identical copy of the code on each processor, rather than distributing them.
3. Each processor on the SGI system is about six times faster than the CM5 processors.

Table 1: Speedups for a sparse matrix multiplication with 10 nonzero elements per row, random column positions. n refers to the system size. The test code was written in Fortran 90, with HPF and CMF compiler directives to control data layout. As can be seen, very large system sizes are required to achieve payoff from parallel systems.

is stored in *jagged diagonal* format. Table 1 shows the typical sparse matrix multiplication performance that occurs for sparse matrix-vector computation on parallel systems, showing that adequate performance only occurs for very large networks.

Whilst this appears rather gloomy for this model, two very obvious (and trivial) parallel exploits have been used — parameter space exploration, and ensemble collection of statistics used in computing the lifetime distributions in [5]. The latter method is of limited use, as the behaviour over long runs (i.e. the 'tail' of the lifetime distribution) is also important.

Recently, I have turned my attention to a spatially dependent Ecolab model[6]. Since one expects migrations of individuals between neighbouring patches to be relatively slow compared with the typical generation time of the system, and that these migrations will be of a "nearest neighbour" type, one can parallelise the system over the spatial dimension.

Ecolab is designed as a series of TCL[4] commands written in C++ that execute basic atomic operations of the model, such iterate eq1, mutate the parameters, perform migration etc. This allows the experimenter a great deal of flexibility in scripting a typical experimental run, without having to recompile. Some of operations are specialised visualisation widgets that may be inserted into the experimental script to gain insight into the model's behaviour. This type of model construction is typical in complex systems research, as serendipity proves to be a powerful method for understanding these systems. Other

examples include Wuensche's *DDLab*[1], and the Santa Fe Institute's *Swarm*[2] system[3].

This program structure, along with the requirement that the same code be executable on a single processor system for visualisation purposes leads to unusal requirements from a parallel programming system. In this case, MPI was adopted as the programming environment for the implementation. Each MPI call is protected by `#ifdef MPI`, and the code semantics are such that `MPI` undefined is equivalent to running the code on one thread.

- The experimental script was run on thread 0 (master thread). If the script had been run on all threads, then the possibility exists for individual threads to be executing different atomic operations (`migrate` and `connect_plot` (plot of non-zero elements of $\boldsymbol{\beta}$) say) that have different cooperative communication requirements (regular grid communication vs gather). This could lead to deadlock.

- Each slave thread waits for a command sent from the master thread, which is then passed to the slave TCL interpreter. An operation to be executed in parallel has the `PARALLEL` statement placed as the first line of the code. This macro (if executed on the master thread) broadcasts the name of the current routine, plus any arguments to the slave threads.

- Any TCL variable from the environment used in the C++ code is accessed via a `tclvar` class. This allows the TCL variable to be mixed in normal C++ arithmetic expressions. In the parallel environment, the `tclvar` declaration causes a broadcast of the TCL variable's value to all the slave processors.

- Routines that need to access the whole model (rather than just that subset residing on the current thread) will need to use the `GLOBAL` directive, which arranges for the data on all of the slave threads to be gathered onto the master thread.

- Routines that perform gridded communication (eg `migrate`) use a directive to set up the neighbourhood (`SET_NBHOOD(coords)`), where coords is a matrix of the relative coordinates of the neighbouring points. For example, the 2D Moore and von Neumann neighbourhoods are defined by

  $$\begin{matrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{matrix} \qquad \begin{matrix} 1 & 1 & 0 & -1 & -1 & -1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & -1 & -1 & -1 \end{matrix}$$

  respectively. This function organises for the edges of the cells to transferred to the neighbouring cells. This then adjusts the model's data to allow the code running on that thread to access (read-only) data outside the normal range.

[1] http://www.santafe.edu/~wuensch/ddlab.html
[2] http://www.santafe.edu/projects/swarm

These directives are pretty much sufficient to express all the parallel programming needs, apart from the occasional debugging statement to work out where things go wrong. What is more, it depends only on standard C++, TCL and MPI implementations, although the C preprocessor is given a fair workout, which can make debugging a chore.

As reported in [6], this code shows excellent scalability up to 12 processors, and even shows superlinear speedup on small numbers of processor (due to cache effects). The Ecolab code and documentation is available from the Ecolab Web site[3].

3 Swarm

Swarm is an object oriented simulation environment designed primarily to simplify coding of *agent based models*. Agent based modeling consists of modeling the individual components of the system as separate objects, each with its own behaviour as it interacts with the other agents of the system. The global dynamics of the system emerges out of these interactions. This contrasts with *collective* models, such as Ecolab, which model an entire ecosystem as a differential equation (eq. (1)). The individual components of the system (species in this case) are too simple to considered 'agents'.

Swarm consists of a toolbox of techniques, such as a random number library, a 2D grid and a number of visualisation tools such as graphs and histograms. It is not scripted as such, although there are some add-on components which allow scripting, however it makes use of a powerful technique called probes. A probe allows the user to examine the contents of an object, or to pass some aspect of that object into a visualisation tool. Probes can be instantiated dynamically, so the researcher can interact with a running simulation, selecting what objects to view, changing the values here and there as appropriate. This sort of interactivity is not currently possible in Ecolab.

In order to implement this dynamic probing ability, the object oriented programming language must have dynamic binding of the data types and object methods. C++ performs this statically at compile time, and typically throws away the information required. As such, Swarm is implemented in Objective C, which has this dynamic binding.

All this complexity and richness poses a question of how parallel computing might help improve the performance of models written in Swarm. It is clear that each object could potentially be run on a separate thread, and as the way in which they interact is via *messages* (akin to class methods of C++), we can consider each object is a miniature interpreter, like the TCL interpreters used in the Ecolab system.

Consider the heatbugs example application in Swarm. The heatbug lives in a 2D world, with a property called *temperature* obeying the usual heat diffusion

[3] http://parallel.hpc.unsw.edu.au/rks/ecolab.html

equation. Each individual heatbug has an ideal temperature that it likes to be at, if it is too cold, it moves towards higher temperatures, if it is too hot, it moves away. Also, each individual is putting out a certain constant amount of heat per unit time.

This should be a good example to try an parallelise. The diffusion equation is something that scales well on parallel computers, and this portion of the simulation accounts for most of the CPU cycles. The heatbugs themselves independently measure the temperature, make a move and then add heat into the environment.

3.1 Proxies and Packaging

Whilst the object nature of Objective C appears to offer a simple form of concurrency, a problem occurs in that each object is referred to by a pointer. This implies a single address space in which all threads run, limiting parallelism to the shared-memory (or SMP) variety. Another problem to address is that of the return value. In order to exploit parallelism, the message call must return immediately, either returning a void value, or a bland object value such as "self" (pointer to object receiving message). Similar considerations apply to arguments passed by reference, that are modified by the routine. This problem leads to the concept of a *proxy* — in this instance a placeholder for the return value. One ought to be able to query this proxy to determine if its value has been calculated. Any subsequent use of the return value ought to block until the routine computing the return value has finshed.

This concept of a proxy allows one to escape the need to have single address space for the objects to reside. The proxy can refer to an object residing in a different thread's address space. When a message is sent to the proxy, it will forward the message onto the remote object that it is a proxy for. This uses the Objective C `forward` mechanism — if no method exists for a particular message, prior to an error being raised, a `forward` message is sent to the object, which then allows the proxy object to forward the message onto its remote client.

A message in Objective C is implemented as a function call, with the arguments passed on the stack. Since stacks are thread local, the argument frame will need to be packed into compact binary representation to be passed to the remote thread. The packing and unpacking operation is recursive, because any objects contained within the argframe need to be packed as well into memory space independent form (as objects are represented by pointers to structures). Objects may also contain other objects. One suitable binary representation is to start with run length encoded binary string containing the contents of the `struct` underlying the object. This is sufficient for all simple types (`int`s, `float`s etc.), as well fixed sized complex types, such as `struct`s, `union`s and fixed length arrays. The remaining data types include classes, selectors, objects and pointers. Both classes and selectors (an object rather like an ANSI C prototype representing a class method) are compile time constants, and so exist on all

threads. All that is needed is to pass the unique string identifying these objects. Any object needs its class as its first field, followed by the binary representation of the data, followed by a list of the more complex types. Pointers as such can not be meaningfully passed between threads, and so should be disallowed from parallel messages, with the possible exception of `char *`, which often refers to character strings.

Proxies and binary object packages (which appear to be known as codings) are features of the *distributed objects*[9] capability of the OPENSTEP system. It appears that plans are under way to implement a type of distributed objects in gcc, however, to date it is neither implemented, nor documented.

3.2 Distribution of Data

Distributing agents in parallel should occur at creation time. One might issue a message of the form

```
p = [parallel create class create ...];
```

This call will create an object of class *class* on a remote thread, call its `create` method, and return a proxy `p` to the remote object. The objects will be allocated to the threads on a round-robin basis by default. This behaviour could be changed by a call to set a blocking factor.

These objects should be allowed to move as well. A message `alignWith` sent to an object's proxy will cause the object to be moved to the thread containing the target object. In order for objects to be movable, all references to that object must be via proxy, even when the reference is on the same thread as the actual object. Since there may be more than one proxy to each object, the objects new coordinates must be broadcast to all proxies, in order to allow the proxies to update the location of their target objects. Another requirement for an object to be movable is that any objects contained within it are proxies.

Returning to our heatbugs application, the heatbugs themselves become the above style of proxied parallel object. The world array that contains the heatbugs could still be the normal 2D array from the Swarm space library, but containing proxies rather than heatbugs. However, the heat array is an array of integers, on which the diffusion operator operates. It would be woefully inefficient to store proxies to integers, and one would miss out on the regular communication pattern. Instead, one should create a new type of proxy (which I call a distributed object). The full array will be broken up into a number of subarrays, one per thread. The proxy will actually understand the coordinates being passed to messages, and ship the message onto the subarray that needs to handle it. Each thread will have its own proxy to the distributed object.

As for implementing the diffusion equation, a similar mechanism to the `SET_NBHOOD` method used in Ecolab should be employed. One of the methods of the proxy will be to move data according to a template, and cache the

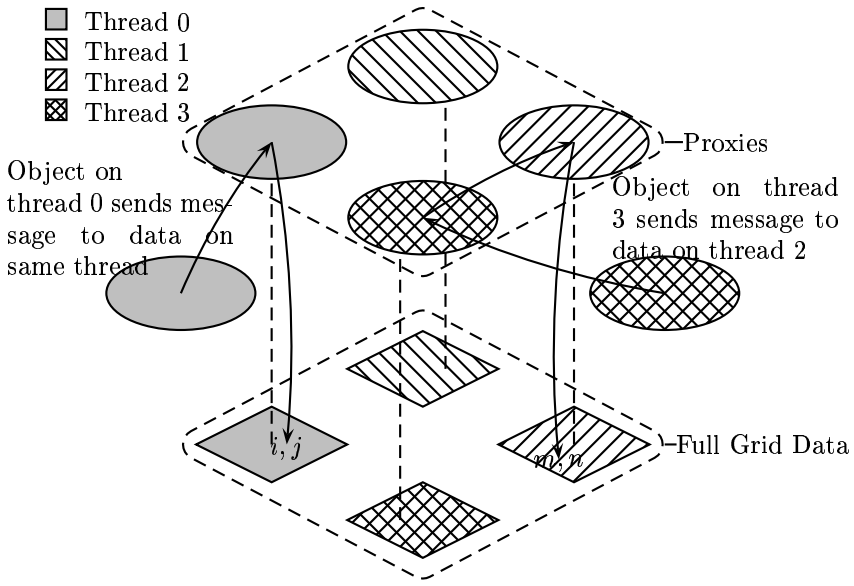

Figure 1: Schematic of Dsitributed Grid Object

boundary elements of neighbouring cells. Then the proxy will respond with data out of the cache when one of the boundaries is requested.

Figure 1 shows a schematic of the distributed object. If the message sent to the distributed proxy is destined for the current thread, it is simply passed to the local subgrid. If the message is destined to another thread, then it is forwarded to the proxy on the other thread. In order to prevent deadlocks, everytime a proxy receives a message, it will call a routine to process all incoming messages, and pass them onto relevant local objects. If a routine is blocking on getting a return value from a proxy, then incoming messages are continuously polled for. This is a form of non-preemptive multitasking, similar to that used in the Windows or MacOS operating systems. At this stage, the added complexity of preemptive multitasking (as used in Unix and NT based systems) shouldn't be needed.

4 Conclusion

The study of complex systems provides many challenges to the parallel computing community. Traditional compiler and message passing techniques are not going to be generally useful to the field, but by adding some appropriate parallel computing techniques to the standard toolboxes of Complex Systems research

(such as Swarm), the potential for parallel computing to be applied to this field can be realised.

References

[1] David G. Green. Towards a mathematics of complexity. In *Complex Systems: From Local Interactions to Global Phenomena*, pages 98–105. IOS Press, Amsterdam, 1996. also *Complexity International*, **3**, http://www.csu.edu.au/ci.

[2] Stuart A. Kauffman. *The Origins of Order: Self Organization and Selection in Evolution.* Oxford UP, Oxford, 1993.

[3] Nelson Minar and Roger Burkhart. The Swarm simulation system: A toolkit for building multi-agent simulations. Technical Report 96-06-042, Santa Fe Institute, 1996. http://www.santafe.edu/projects/swarm/swarmdoc/swarmdoc.html.

[4] J. K. Ousterhout. *TCL and the Tk Toolkit.* Addison-Wesley, 1994.

[5] R. K. Standish. Statistics of certain models of evolution. *Phys. Rev. E*, 1998.

[6] Russell Standish. Cellular Ecolab. In Standish et al. [7], page 80. also in *Complexity International*, **6** http://www.csu.edu.au/ci.

[7] Russell Standish, Bruce Henry, Simon Watt, Robert marks, Robert Stocker, David Green, Steve Keen, and Terry Bossomaier, editors. *Complex Systems '98 — Complexity Between the Ecos: From Ecology to Economics.* Complexity Online, http://life.csu.edu.au/complex, 1998. also in *Complexity International*, **6** http://www.csu.edu.au/ci.

[8] Russell K. Standish. Population models with random embryologies as a paradigm for evolution. In *Complex Systems: Mechanism of Adaption.* IOS Press, Amsterdam, 1994. also *Complexity International*, **2**, http://www.csu.edu.au/ci.

[9] Unknown. *Object Oriented Programming and the Objective-C Language.* Apple Computer, 1996. http://developer.apple.com/techpubs/macosxserver/ObjectiveC/index.html.

[10] Andrew Wuensche. Discrete dynamical networks and their attractor basins. In Standish et al. [7], page 3. also in *Complexity International*, **6** http://www.csu.edu.au/ci.