Assessing and Comparing the Usability of Parallel Programming Systems

Gregory V. Wilson[1] R. Bruce Irvin[2]

Abstract

Parallel programming is widely acknowledged to be more difficult than sequential programming. One reason for this is that parallel programming systems are more difficult to use than their sequential counterparts. In particular, few parallel programming systems can support the software engineering requirements of large applications. We intend to assess and compare the usability of a variety of parallel programming systems using a small suite of chained applications called the Cowichan Problems.

1 Introduction

In their 1989 survey paper, Bal *et al.* listed more than 300 parallel programming systems [4]. A similar survey today could well uncover twice as many, but a poll of application programmers would probably find that the only ones in general use are data-parallel extensions to Fortran and procedural message-passing libraries of various flavours. It is therefore interesting to ask: why have so many parallel programming systems been developed, and why are most of them used so little?

The first question is the easiest to answer: parallelism has become a hot topic. All modern high-performance computers use parallelism to some degree, and the most cost-effective in any class use it a lot. Many of these machines require programmers to deal explicitly with communication and synchronization issues which pre-existing languages could ignore. There is also increasing interest in using networks of workstations as distributed supercomputers, so programming tools capable of handling both parallel and distributed systems are sorely needed.

[1]Dept. of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4 `gvw@cs.toronto.edu`

[2]Dept. of Computer Science, University of Wisconsin, Madison, Wisconsin, USA 53706 `rbi@cs.wisc.edu`

At the same time, language research during the past two decades has led to the notions of declarative programming and interacting agents. Languages based on these ideas, such as SISAL [13] and Concurrent Aggregates [7], seem naturally parallel, which is more than can be said for today's memory-overwriting languages. For those worried about dusty decks, hybrids such as Fortran-M [15], CC++ [28], and pC++ [6] hold out the hope that the transition from sequential to parallel programming might not have to be all that painful.

Finally, many parallel programming systems (PPSs) have been developed because they had to be[3]. In less than a decade, one of the authors has worked on two different bit-parallel SIMD arrays (the ICL DAP and Thinking Machines CM-200), a transputer-based Meiko Computing Surface, a Fujitsu AP1000 with three different sets of communication and synchronization hardware, and networks of workstations using message passing (PVM) and distributed shared memory (Orca). Each of these machines has created a niche for at least one idiosyncratic programming system.

The reasons why so many of these PPSs are only used by their developers (if at all) are murkier. Part of the blame undoubtedly lies with the diversity of hardware, and the attendant lack of portability, alluded to above. A second reason is that an academic researcher's principal interest is developing new languages, not supporting existing ones. By the time a system is 90% complete, its author is already thinking about its successor. Few application programmers are willing to invest effort porting codes to systems which are not properly supported, and liable to change or evaporate without notice. Third, many research groups have chosen to explore the potential of parallelism without consideration for compatibility with older languages. Application programmers must therefore face the prospect of completely re-writing their programs, rather than simply porting them. As the next section argues, this can obviate any performance that might be gained from parallelization. Finally, many PPSs simply cannot support the software engineering load of large applications. Many provide only a single global namespace, do not include I/O capabilities, require users to marshal and unmarshal data explicitly, or do not permit hardware to be timeshared.

The net result is that many PPSs are never used for anything more complicated than generating the Mandelbrot Set, simulating the Game of Life, multiplying dense matrices, or finding all solutions to the N-queens

[3] Note that we use the term "PPS" to include both languages and programming libraries.

problem for some small N. Lacking the feedback that comes from more demanding use, few systems are improvements over their predecessors in the way that successive versions of operating systems have improved on one another, or the languages in the Algol$\rightarrow$Modula-n family have become both more powerful and more secure. Most applications programmers therefore choose to play it safe and stick to parallel programming's lowest common denominators: data-parallel Fortran, and message-passing libraries.

The historical reasons for tolerating this situation are becoming less valid with each passing year. When only a few parallel computers existed, and these were of widely different types, software standardization simply didn't make sense. Similarly, when supercomputers were very expensive, the cost of software development seemed relatively smaller, and the requirement for maximum performance relatively greater, than in sequential computing. As [9] points out, however, these arguments are losing force. Now, as in sequential computing, the first question is not "How do I write programs for this machine?", but "Will this machine run my programs?" As microprocessor-based MIMD hardware comes to dominate massively-parallel computing, we see increasing emphasis on standards such as High-Performance Fortran (HPF) [23], and the Message-Passing Interface (MPI) [31].

But herein lies a danger. History shows that it is extremely difficult to displace a programming language or system once it becomes established. Cobol and MS-DOS are the most frequently cited examples, but many systems which should have been superceded long ago have managed not only to cling to life, but to grow in influence. (The fact that the mail signature "The last good thing written in C was Schubert's Ninth Symphony" can elicit smiles simply proves that even the truth can be funny.) Established systems evolve slowly, if at all. It would therefore seem that the best time to guarantee ourselves a good parallel programming paradigm is right now, before a large number of large applications are created and take on lives of their own.

We therefore propose to use a set of simple programming problems to compare the usability of parallel programming systems. Competent programmers, fluent in a specific system, will implement solutions to these problems and report their experiences in terms of development time, code size and clarity, and runtime efficiency. These problems are on the scale of the toy applications described several paragraphs ago both to keep implementation time low and because we feel that any PPS produced today ought to be able to support such simple programs. As the second author put it, "We're all sick of seeing these, but maybe if we all catch the disease at the same time we'll finally build up some immunity to it." One significant innovation in this work is that we also require implementations of our suite to chain problems together, so that the output of one problem is the input to another. This will allow us to assess how well each PPS supports modularization and code re-use (or, more realistically, how much extra effort programmers must invest to achieve these goals when using a particular PPS).

Section 2 motivates this work by showing how poor usability can limit the realizable parallelism in a program. Section 3 surveys previous work on the comparison of parallel programming systems, while Section 4 outlines our choice of methods, and the constraints on our work. Section 5 then presents the suite of problems we intend to use. In recognition of the original work done by Feo and others on the Salishan Problems [12], this suite is called the Cowichan Problems[4].

The authors wish to acknowledge contributions from the participants in the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems, particularly Alain Fagot, Frederic Guidec, Thomas Kunz, Jan Prins, and Julie Vachon. Help was also given by Peter Bailey, Henri Bal, Bruce Boghosian, Paul Lu, Neil MacDonald, Steve Moyer, Jonathan Schaeffer, and David Sitsky, who made many comments and improvements. The first author was supported by a grant from the European Commission's Human Capital and Mobility Programme, and later by the Computer Systems Research Institute at the University of Toronto. He is grateful to Henri Bal and the Vrije Universiteit, and to Ken Sevcik and the Universityr of Toronto, for their hospitality.

2 Quantifying the Importance of Usability

According to Amdahl's Law, the time t required to execute a program has two components: t_{seq}, representing sequential operations that cannot be parallelized, and t_{par}, representing those operations which can. If we let σ be the "serial fraction" of the program, so that $t_{seq} = \sigma t$ and $t_{par} = (1-\sigma)t$, then the speedup which can be achieved using P processors is:

$$s(P) \quad = \quad \frac{t(1)}{t(P)}$$

[4]Pronounced *Cow*-i-chun. Like Salishan, the word is a Northwest Coast Indian place name.

$$= \frac{t}{\sigma t + (1-\sigma)t/P}$$

As $P \to \infty$, the potential speedup is bounded by $1/\sigma$.

What is not usually acknowledged is the influence of programming system usability on the potential speedup of sequential programs. As stated above, Amdahl's Law applies only to a single run of a program. In the real world, programs are first developed, and then run many times. If we let T be the total running time of a program over its lifetime, D_{seq} be the time required to develop the sequential components of a program, and D_{par} be the time required to develop the parallel components (or parallelize the sequential components), then the lifetime speedup we can achieve is given by:

$$\begin{aligned} S(P) &= \frac{D_{seq} + T(1)}{D_{seq} + D_{par} + T(P)} \\ &= \frac{D_{seq} + T}{D_{seq} + D_{par} + \sigma T + (1-\sigma)T/P} \end{aligned}$$

Assume that we are parallelizing a "dusty deck", i.e. that the time D_{seq} has already been invested. If we let $\phi = D_{par}/T$ be the ratio of parallelization time to total program runtime, then achievable speedup is limited by:

$$\begin{aligned} S_{legacy}(P) &= \frac{T}{D_{par} + \sigma T} \\ &= \frac{1}{\phi + \sigma} \end{aligned}$$

The time required to parallelize a program can therefore be seen as increasing that program's effective serial fraction. Unless parallelization time can be substantially reduced, massively-parallel computing is likely to remain attractive only in those cases in which:

- the application is trivially parallel (e.g. task-farming a rendering calculation);
- the expected total program runtime is very large (i.e. the program is a package used extensively in a field such as computational chemistry or vehicle crash simulation); or
- cost is not an obstacle (i.e. the customer is the NSA).

3 Previous Work

In a 1981 survey article [26], B. A. Sheil observed that:

> As practiced by computer science, the study of programming is an unholy mixture of mathematics, literary criticism, and folklore.

Little has changed since. Computer scientists pay lip service to the importance of usability, but are strangely reluctant to try to quantify it, particularly with respect to the tools they use themselves. Religious debates about the "right" first-year teaching language or the "best" GUI toolkit are common; systematic use of the experimental method is not.

Most of the experiments which have been done have concentrated on "programming in the small" issues, such as readability and code-pattern recognition [17, 30, 33]. The little work that has been done on "programming in the large" has been anecdotal, such as [14]. This book presented various authors' impressions of Pascal, C, and Ada (before any complete Ada implementations existed). Its editors formulated a list of questions to use in comparing languages, but did not use it themselves. A modified version of this questionnaire, tailored for parallel programming languages, is given in Appendix B, and will be used as a guideline in our work.

Two comparative efforts in parallel programming are [1] and [12]. The first presented implementations of a simple numerical quadrature program in more than a dozen different parallel languages in use on mid-1980s hardware. The second presented implementations of the Salishan Problems—Hamming number generation, isomer enumeration, skyline matrix reduction, and a simple discrete event simulator—in C* [19], Occam [21], Ada [22], and a variety of dataflow and functional languages. Both of these books convey the flavor of the languages they describe, but neither made any effort to compare languages or problem implementations directly.

Inspired by the Salishan Problems, the first author developed a suite of seven problems [34] in order to test the limits of parallel programming systems. Individual problems were badly load-balanced, manipulated irregular pointer-based structures, and so on. Our experience implementing them was extremely valuable, as it uncovered many bugs in the programming system used, and revealed several shortcomings of the system's design. However, we also discovered that implementing each problem takes six to eight weeks, which we feel is prohibitive at this exploratory stage.

4 Methodology

The main aim of this work is to assess and compare the ease with which programs can be developed and/or ported using different parallel programming systems. Here, we discuss issues related to choosing the problems that make up the suite, assessing the complexity of particular implementations, and other purposes which implementations of these might serve. Recognizing that these problems are extremely simple, and easy to parallelize, we refer to them as "toys".

4.1 Criteria for Selection

Our criteria for including problems in this suite are as follows:

1. Each toy should require no more than an afternoon to write and test in a well-supported sequential language. We feel that making individual problems more complicated will only discourage implementation.

2. The correctness of each implementation must be relatively easy to verify. Toys whose output is easily visualized are therefore to be preferred, as are toys whose results are insensitive to floating-point discretization effects.

3. Each toy application should be amenable to an analysis of speedup or isoefficiency [16] to allow a comparison of theory and practice.

4. At least some toys should not be "infinitely scalable". Many real-world applications are not, and this suite should reflect such limitations.

5. At least some toys should require I/O, since this important aspect of real-world programming is often neglected by PPS designers.

6. There should be some potential redundancy in the type of parallelization done, so that re-use of software can be demonstrated.

7. Together, the toys in the suite must exercise a wide range of common parallel operations (Appendix A.1) and memory reference patterns (Appendix A.2).

8. The "obvious" implementations of these toys should span a wide range of parallel programming paradigms, such as task farming, geometric and functional decomposition, and speculative parallelism.

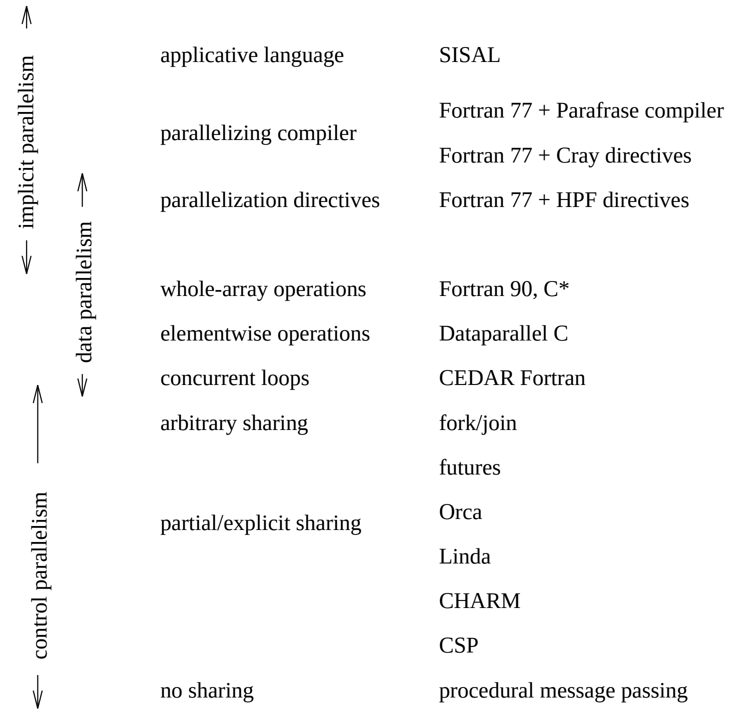

Figure 1: A Taxonomy of Parallel Programming Systems

9. Parallelizations for several (if not all) of the types of programming systems shown in Figure 1 should be known. This suite is not intended to be a collection of research problems; reasonably simple and efficient implementations in most environments should be straightforward to produce.

4.2 Specification

Toys should be specified in terms of inputs and outputs, rather than algorithmically. For example, "sort N integers in the range $1 \ldots R$, where R is not known in advance and there may be multiple instances of a given value" is to be preferred over "parallelize quicksort". As [18] argued, which parallel algorithm is best in a particular programming system depends critically on the architectural assumptions made by that system. Implementors should be free to employ those methods which they find natural and convenient. The only exception we make to this rule is in the two matrix equation solvers, where we require Gaussian elimination and successive over-relaxation. Given the enormous number of matrix solution algorithms in use, we felt we had to specify these particular algorithms in order to ensure that different implementations of this suite would be comparable.

4.3 Software Engineering Issues

These toys will exercise many different aspects of parallel systems. However, their "single algorithm per program" model is not representative of real applications. These usually contain several discrete (and sometimes overlapping) phases, each of which is qualitatively different. A full implementation of this suite will therefore have two parts. In the first, each toy will be implemented as a stand-alone program. In the second, toys will be chained together to create the kind of phase behaviour seen in real applications (Figure 2). This will test the ease with which heterogeneous parallelism can be mixed within a single program. It will also show how well the system supports code re-use and information hiding, which are crucial to the development of large programs.

A second reason for chaining toys is that many groups are now using algorithmic skeletons to structure parallel programs [8, 2, 25]. Such skeletons encapsulate the details of a particular style of parallelism, and hide low-level or platform-dependent issues. We hope that chaining will show the strengths and weaknesses of such systems.

Finally, chaining should be designed so that some toys can be executed concurrently. Many PPSs impose extraneous constraints on programs, e.g. require all processes to participate in every barrier synchronization, or require the same executable to be loaded onto each processor. These constraints can limit the exploitation of potential concurrency. Permitting, but not requiring, concurrent execution of several toys should uncover such limitations.

4.4 Sizing

One crucial aspect of the specification of toys is the way in which the sizes of problems are determined. In a completely frozen model, the actual size of each problem would be compiled into each toy. A fluid implementation, by contrast, would allow sizes to be specified at run-time, and would only then

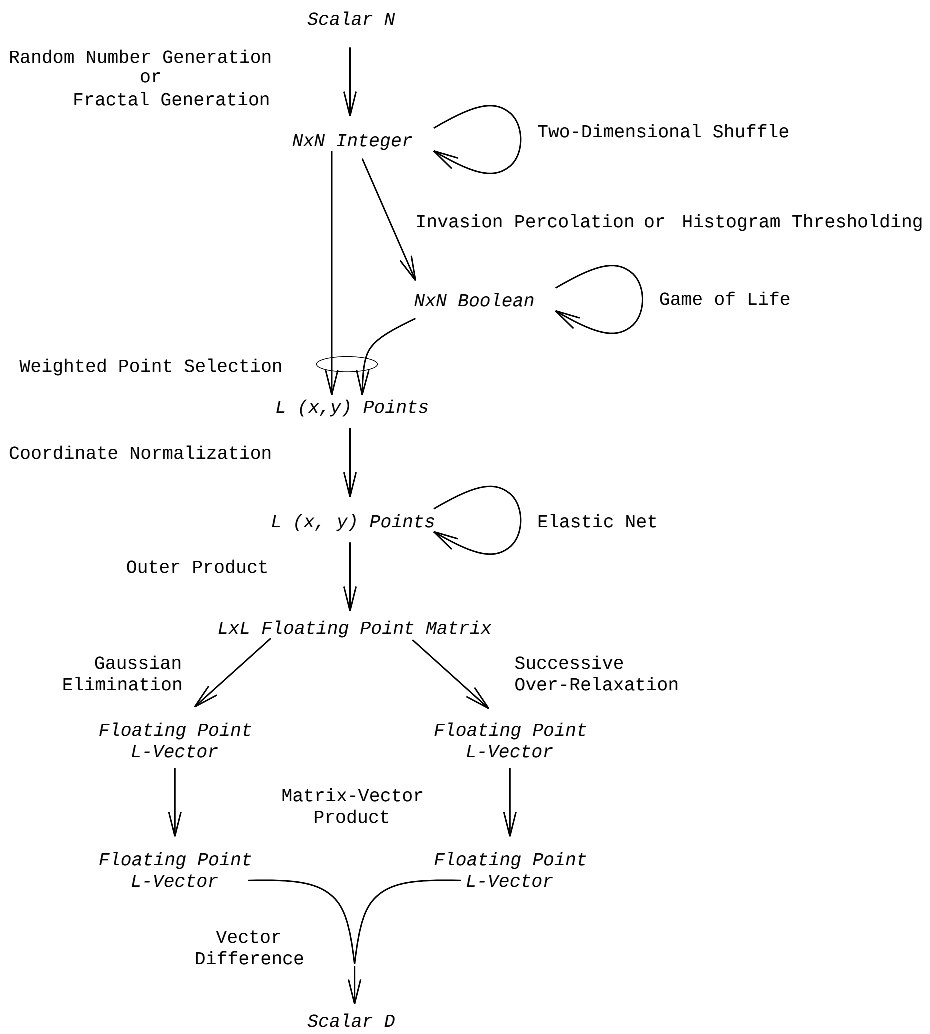

Figure 2: Chaining

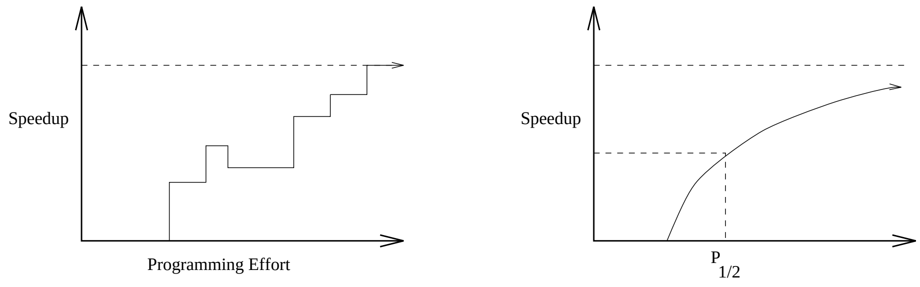

Figure 3: Performance vs. Time

allocate and partition data structures. We have chosen to use an intermediate model (which might be called "slushy"), in which the maximum size of individual problems is specified during compilation, but the actual size of a problem is only determined when the toy begins to execute. We hope that the use of this model will force implementors to deal with at least some of the software engineering issues involved in building flexible large-scale applications, without making implementation prohibitively difficult.

4.5 Assessing Complexity

There are at least two different ways in which usability might be compared. One would be to measure the performance achieved by an "average" programmer as a function of time on each of a range of problems. This would lead to graphs of the kind shown in Figure 3, where the left graph shows a particular result and the right graph a simplified abstraction of it. By analogy with Hockney's $n_{1/2}$ measure [20], which is the length of vector on which a pipelined architecture achieves half its theoretical peak performance, we could in principle find the value of $p_{1/2}$—the programming time required to achieve half of a machine's peak performance[5] for a particular combination of programming system and problem type.

A second option would be to measure the complexities of implementations of a single application using different programming systems. With this approach, comparative performance or speedup figures would be supplemented by complexity measures, as shown in Figure 4.

[5] Note that if performance was measured as a fraction of the figures quoted by manufacturers for their machines, it is unlikely that the halfway mark would ever be reached.

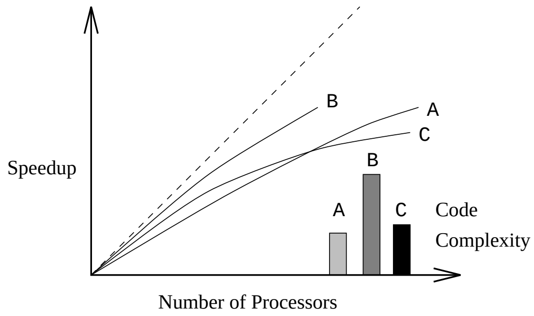

Figure 4: Comparative Code Complexity

The drawback of the first approach is that there are no "typical" problems, or "average" programmers. Meaningful measurement of $p_{1/2}$ would therefore require measurement of a large number of programmers working on many problems. Since this is too expensive to be practical, the second approach is the one we have chosen to adopt. It has drawbacks as well, most notably the fact that no-one has ever demonstrated any better measure of program complexity than a simple count of the number of lines in a program [10]. Since we anticipate a wide variation in the types of progamming systems that will be used in this work (ranging from declarative dataflow languages through functional languages with Lisp-like syntax to parallel dialects of Fortran and C), and in the indentation, commenting, and layout styles of particular programmers, we may measure complexity by counting the number of tokens in the source code which are keywords, intrinsic operators, user-defined functions, type or variable declarations, and miscellaneous punctuation.

4.6 Other Uses for Implementations

This suite is intended for assessing PPS usability, but we envisage at least three other uses for implementations of it:

Related Research: We hope that the existence of this suite will indicate to the developers of new parallel programming systems what a mature tool should be able to support. If sufficiently diverse, the suite could also be used to help test the correctness of new or improved systems. Finally, this suite could complement existing performance benchmarks, since performance figures for individual toys could be compared.

Tool Development: Requests from tool developers for existing parallel programs, or for event traces from them, are common on newsgroups such as `comp.parallel`. We hope that traces from implementations of this suite will be used to compare different debugging and performance monitoring tools, that their source code will be used to test new parallelizing compilation techniques or mapping and scheduling algorithms, and so on.

Teaching: Implementations will be suitable for use as classroom examples, since they will be small enough to be understood quickly. The toys themselves should also be suitable as classroom exercises in a senior undergraduate course on parallel computing.

4.7 Some Criticisms

Why not use an existing benchmark suite for this purpose?

Implementations of benchmark suites have usually been coded for absolute speed. They are not representative of "normal" practice, in which reasonable performance is the usual goal, and so are inappropriate for this work. In addition, most existing benchmark suites are too large to be implemented or ported quickly (e.g. SPEC [32]), over-specified (e.g. the Livermore Loops [11]), or concerned with a limited range of applications (e.g. the NAS Parallel Benchmarks [3]). Finally, most exist only in C or Fortran (or both); many interesting parallel programming systems are built on top of other languages, including Modula-2, Scheme, Prolog, and various dataflow languages. We do not want the effort required to translate several large programs to discourage researchers from participating in this exercise.

Won't the results depend primarily on programmer ability?

The short answer to this question is yes, but no more than the results from performance benchmarks. A longer answer is that since we intend to measure the combination of code complexity and performance achieved, rather than performance as a function of development time, programmers will be allowed to revise and improve their implementations when and as desired.

Code	Toy	Code	Toy
E	elastic net	O	outer product
G	Gaussian elimination	P	matrix-vector product
H	halving shuffle	R	random matrix generation
I	invasion percolation	S	successive over-relaxation
L	Game of Life	T	image thresholding
M	Mandelbrot Set	V	vector difference
N	point normalization	W	point value winnowing

Table 1: Single-Character Codes for Toys

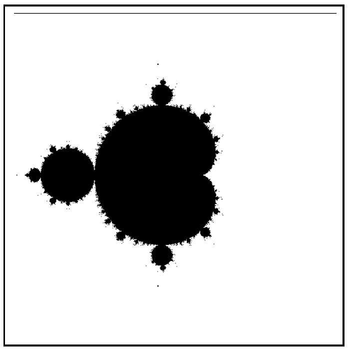

Figure 5: The Mandelbrot Set

5 The Cowichan Problems

This section describes the problems making up the full problem suite. As discussed above, the implementation of each is to be used both as a stand-alone program and as a module in the chained problem sequence described in Section 6.1. Table 1 lists the one-character codes sometimes used to identify toys.

5.1 `mandel`: Mandelbrot Set Generation

This module generates the Mandelbrot Set for a specified region of the complex plane (Figure 5). The inputs to this module are:

`nrows, ncols`: the number of rows and columns in the matrix.

`x0, y0`: the real coordinates of the lower-left corner of the region to be generated.

`dx, dy`: the extent of the region to be generated.

Its output is:

`matrix`: an integer matrix containing the iteration count at each point in the region.

Given initial coordinates (x_0, y_0), the Mandelbrot Set is generated by iterating the equation

$$\begin{aligned} x' &= x^2 - y^2 + y_0 \\ y' &= 2xy + x_0 \end{aligned}$$

until either an iteration limit is reached, or the values diverge. The iteration limit used in this module is 150 steps; divergence occurs when $x^2 + y^2$ becomes 2.0 or greater. The integer value of each element of the matrix is the number of iterations done. Note that, as in all problems, the output is required to be independent of the number of processors used. This requires programmers to be careful about floating-point roundoff effects.

5.2 `randmat`: Random Number Generation

This module fills a matrix with pseudo-random integers. The inputs to this module are:

`nrows, ncols`: the number of rows and columns in the matrix.

`s`: the random number generation seed.

Its output is:

`matrix`: an integer matrix filled with random values.

Note that, as in all problems, the output is required to be independent of the number of processors used. Generating new seed values from the given seed, and running one copy of the random number generator on each processor, is therefore unlikely to qualify as a solution.

5.3 `half`: Two-Dimensional Shuffle

This module divides the values in a rectangular two-dimensional integer matrix into two halves along one axis, shuffles them, and then repeats this operation along the other axis. Values in odd-numbered locations are collected at the low end of each row or column, while values in even-numbered locations are moved to the high end. An example transformation is:

$$\begin{matrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{matrix} \quad = \quad \begin{matrix} a & c & b & d \\ i & k & j & l \\ e & g & f & h \end{matrix}$$

Note that how an array element is moved depends only on whether its location index is odd or even, not on whether its value is odd or even.

The inputs to this module are:

`matrix`: an integer matrix.

`nrows, ncols`: the number of rows and columns in the matrix.

Its output is:

`matrix`: an integer matrix containing shuffled values.

5.4 `invperc`: Invasion Percolation

Invasion percolation models the displacement of one fluid (such as oil) by another (such as water) in fractured rock [29, 24]. In two dimensions, this can be simulated by generating an $N \times N$ grid of random numbers in the range $[1 \ldots R]$, and then marking the center cell of the grid as filled. In each iteration, one examines the four orthogonal neighbors of all filled cells, chooses the one with the lowest value (i.e. the one with the least resistance to filling), and fills it in. Figure 6 shows the first few steps in this process, while Figure 7 shows the evolution of the final fractal shape.

The simulation continues until some fixed percentage of cells have been filled, or until some other condition (such as the presence of trapped regions) is achieved. The fractal structure of the filled and unfilled regions is then examined to determine how much oil could be recovered. The naïve way to implement this is to repeatedly scan the array; a more sophisticated, and much faster, sequential technique is to maintain a priority queue of unfilled cells which are neighbors of filled cells. This latter technique is similar to the list-based methods used in some cellular automaton programs, and is very difficult to parallelize effectively.

The inputs to this module are:

26	12	72	45	38
10	38	39	92	38
44	29	$\circ$	29	77
61	26	90	35	11
83	84	18	56	52

26	12	72	45	38
10	38	39	92	38
44	$\star$	$\circ$	29	77
61	26	90	35	11
83	84	18	56	52

26	12	72	45	38
10	38	39	92	38
44	$\circ$	$\circ$	29	77
61	$\star$	90	35	11
83	84	18	56	52

26	12	72	45	38
10	38	39	92	38
44	$\circ$	$\circ$	$\star$	77
61	$\circ$	90	35	11
83	84	18	56	52

26	12	72	45	38
10	38	39	92	38
44	$\circ$	$\circ$	$\circ$	77
61	$\circ$	90	$\star$	11
83	84	18	56	52

26	12	72	45	38
10	38	39	92	38
44	$\circ$	$\circ$	$\circ$	77
61	$\circ$	90	$\circ$	$\star$
83	84	18	56	52

26	12	72	45	38
10	$\star$	39	92	38
44	$\circ$	$\circ$	$\circ$	77
61	$\circ$	90	35	11
83	84	18	56	52

26	12	72	45	38
$\star$	$\circ$	39	92	38
44	$\circ$	$\circ$	$\circ$	77
61	$\circ$	90	$\circ$	11
83	84	18	56	52

26	$\star$	72	45	38
$\circ$	$\circ$	39	92	38
44	$\circ$	$\circ$	$\circ$	77
61	$\circ$	90	$\circ$	$\circ$
83	84	18	56	52

Figure 6: Invasion Percolation

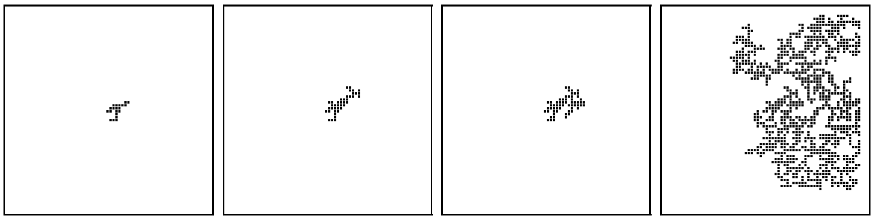

Figure 7: Fractal Generated by Invasion Percolation

`matrix`: an integer matrix.

`nrows, ncols`: the number of rows and columns in the matrix and mask.

`nfill`: the number of points to fill.

Its output is:

`mask`: a Boolean matrix filled with `true` (showing a filled cell) or `false` (showing an unfilled cell).

Filling begins at the central cell of the matrix (rounding down for even-sized axes).

5.5 `thresh`: Histogram Thresholding

This module performs histogram thresholding on an image. Given an integer image I and a target percentage p, it constructs a binary image B such that $B_{i,j}$ is set if no more than p percent of the pixels in I are brighter than $I_{i,j}$. The general idea is that an image's histogram should have 2 peaks, one centered around the average foreground intensity, and one centered around the average background intensity. This program attempts to set a threshold between the two peaks in the histogram and select the pixels above the threshold. This module's inputs are:

`matrix`: the integer matrix to be thresholded.

`nrows, ncols`: the number of rows and columns in the matrix and mask.

`percent`: the minimum percentage of cells to retain.

Its output is:

`mask`: a Boolean matrix filled with `true` (showing a cell that is kept) or `false` (showing a cell that is discarded).

5.6 `life`: Game of Life

This module simulates the evolution of Conway's Game of Life, a two-dimensional cellular automaton (Figure 8). Its inputs are:

`matrix`: a Boolean matrix representing the Life world.

`nrows, ncols`: the number of rows and columns in the matrix.

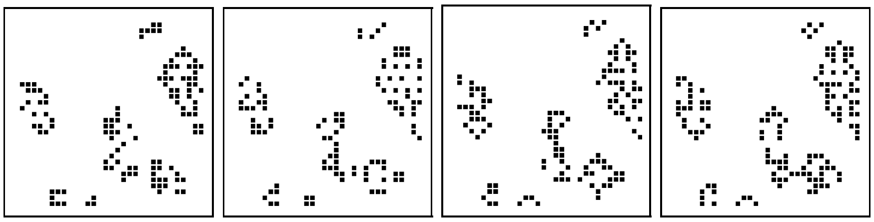

Figure 8: Snapshots from the Game of Life

`numgen`: the number of generations to simulate.

Its output is:

`matrix`: a Boolean matrix representing the world after simulation.

At each time step, this module must count the number of live (`true`) neighbors of each cell, using both orthogonal and diagonal connectivity and circular boundary conditions. The update rule is simple: if a cell has 3 live neighbors, or has 2 live neighbors and is already alive, it is alive in the next generation. In any other situation, the cell becomes, or stays, dead.

5.7 `winnow`: Weighted Point Selection

This module converts a matrix of integer values to a vector of points, represented as x and y coordinates. Its inputs are:

`matrix`: an integer matrix, whose values are used as masses.

`mask`: a Boolean matrix showing which points are eligible for consideration.

`nrows, ncols`: the number of rows and columns in the matrix.

`nelts`: the number of points to select.

Its output is:

`points`: a vector of (x, y) points.

Each location where `mask` is `true` becomes a candidate point, with a weight equal to the integer value in `matrix` at that location and x and y coordinates equal to its row and column indices. These candidate points are then sorted into increasing order by weight, and `nelts` evenly-spaced points selected to create the result vector.

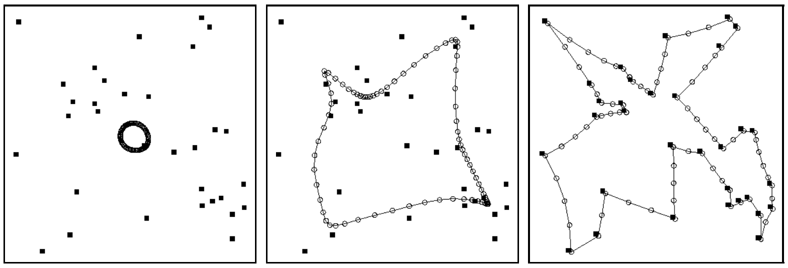

Figure 9: Relaxation of an Elastic Net

5.8 `norm`: Point Location Normalization

This module normalizes point coordinates so that all points lie within the unit square $[0...1]\times[0...1]$. Its inputs are:

`points`: a vector of point locations.

`nelts`: the number of elements in the point vector.

Its output is:

`points`: a vector of normalized point locations.

If x_{min} and x_{max} are the minimum and maximum x coordinate values in the input vector, then the normalization equation is:

$$x'_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

y coordinates are normalized in the same fashion.

5.9 `elastic`: Elastic Net Simulation

This module uses the elastic net algorithm [27] to find an approximate solution to the travelling salesentity problem (TSP). A closed, circular loop, modelled by a large number of doubly-connected points, is placed in the plane. This path is slowly deformed until it connects the points representing the cities to be visited (Figure 9). The rate at which the loop is deformed determines the quality of the final path.

Let the (fixed) location of city i be c_i, and the (changeable) location of point j on the elastic net be p_j. The change in the location of point j during each iteration of the algorithm is given by:

$$\delta p_j = \alpha \sum_i w_{ij}(c_i - p_j) \; + \; K\beta(p_{j+1} - 2p_j + p_{j-1}) \tag{1}$$

α and β are time-invariant factors relating the strengths of the city-to-point and point-to-point forces, and K is a time-dependent factor which controls the rate of annealing. The strength of the force w_{ij} between city c_i and point p_j is given by:

$$w_{ij} = \phi(|c_i - p_j|, K) / \sum_{\ell} \phi(|c_i - p_\ell|, K) \tag{2}$$

where $\phi(d, K)$ is the Gaussian function e^{-d^2/K^2}. Note the normalization of each w_{ij}; in order for this algorithm to work properly, the total force being exerted by each city must be the same. This is accomplished by summing the forces exerted by each city, and then normalizing the city's forces by that total.

The inputs to this module are:

`cities`: a vector of points representing the locations of the cities to be toured. The (x, y) coordinates of these cities must be scaled to lie in the unit square $[0...1]\times[0...1]$.

`n_cities`: the number of cities in `cities`.

`n_net`: the number of elastic net points to use.

`n_iters`: the total number of iterations to perform.

`n_relax`: the number of iterations to perform with each discrete value of K. After each `n_relax` iterations, K is reduced by a (constant) pre-defined scaling factor.

Its output is:

`net`: a vector of `n_net` points representing the tour found.

5.10 `outer`: Outer Product

This module turns a vector containing point positions into a dense, symmetric, diagonally dominant matrix by calculating the distances between each pair of points. It also constructs a real vector whose values are the distance of each point from the origin. Inputs are:

`points`: a vector of (x, y) points, where x and y are the point's position.

`nelts`: the number of points in the vector, and the size of the matrix along each axis.

Its outputs are:

`matrix`: a real matrix, whose values are filled with inter-point distances.

`vector`: a real vector, whose values are filled with origin-to-point distances.

Each matrix element $M_{i,j}$ such that $i \neq j$ is given the value $d_{i,j}$, the Euclidean distance between point i and point j. The diagonal values $M_{i,i}$ are then set to `nelts` times the maximum off-diagonal value to ensure that the matrix is diagonally dominant. The value of the vector element v_i is set to the distance of point i from the origin, which is given by $\sqrt{{x_i}^2 + {y_i}^2}$.

5.11 `gauss`: Gaussian Elimination

This module solves a matrix equation $AX = V$ for a dense, symmetric, diagonally dominant matrix A and an arbitrary vector non-zero V using explicit reduction. (Matrices are required to be symmetric and diagonally dominant in order to guarantee that there is a well-formed solution to the equation.) Inputs are:

`matrix`: the real matrix A.

`target`: the real vector V.

`nelts`: the number of values in each vector, and the size of the matrix along each axis.

Its output is:

`solution`: a real vector containing the solution X.

5.12 `sor`: Successive Over-Relaxation

This module solves a matrix equation $AX = V$ for a dense, symmetric, diagonally dominant matrix A and an arbitrary vector non-zero V using successive over-relaxation. Its inputs are:

`matrix`: the real matrix A.

`target`: the real vector V.

`nelts`: the number of values in each vector, and the size of the matrix along each axis.

`tolerance`: the solution tolerance, e.g. 10^{-6}.

Its output is:

`solution`: a real vector containing the solution X.

5.13 `product`: Matrix-Vector Product

Given a matrix A, a vector V, and an assumed solution X to the equation $AX = V$, this module calculates the actual product $AX = V'$, and then finds the magnitude of the error. Inputs are:

`matrix`: the real matrix A.

`actual`: the real vector V.

`candidate`: a real vector containing the supposed solution.

`nelts`: the number of values in each vector, and the size of the matrix along each axis.

The output of this function is:

`e`: the largest absolute value in the element-wise difference of V and V'.

6 Other Issues

As described in Section 4.3, implementors are required to chain modules together in the manner shown in Figure 2. This section describes the flow of control and data between these chained problems, and discusses several other important aspects of this suite, including I/O and visualization.

6.1 Chained Problem Sequences

The chained implementation of these problems executes individual toys in the following order:

1. An integer matrix I with r rows and c columns is created either by using the Mandelbrot Set algorithm (Section 5.1) or by filling locations with random values (Section 5.2).

2. The integer matrix I is shuffled in both dimensions (Section 5.3).

3. Either invasion percolation (Section 5.4) or histogram thresholding (Section 5.5) is used to generate a Boolean mask B from I in which the (minimum) percentage of `true` locations is P. Like I, B has r rows and c columns.

4. The Game of Life (Section 5.6) is simulated for G generations, using B as an initial configuration. This step overwrites the Boolean matrix B.

5. A vector of L (m, x, y) points is created using the integer matrix I and the Boolean matrix B as described in Section 5.7.

6. The elastic net algorithm is simulated using the points created in the previous step as cities.

7. The final locations of the net points from the prevoius step are normalized, as described in Section 5.8.

8. An $L \times L$ matrix A and an L vector V are created using the normalized point locations from the previous step, as described in Section 5.10.

9. The matrix equation $AX = V$ is solved using Gaussian elimination and using successive over-relaxation to generate two solution vectors X_{gauss} and X_{sor}. In a parallel system, these two problems should if possible be executed concurrently.

10. The checking vectors $V_{gauss} = AX_{gauss}$ and $V_{sor} = AX_{sor}$ are calculated. In a parallel system, these two problems should if possible be executed concurrently.

11. The norm-1 distance between V_{gauss} and V_{sor}, i.e. the greatest element-wise absolute difference, is calculated. This measures the agreement between the solutions found by the two methods.

Because of choices in steps 1 and 3, there are four possible chained sequences. A minimal implementation will provide one of these, while a full implementation will be a single program which can execute any of the four sequences. Intermediate implementations providing all four sequences as separate programs are also acceptable.

6.2 Input and Output

I/O is an important part of programming, but is often treated as being of secondary importance by language designers. This suite requires all stand-alone toys to read input values from files, and write results back; the chained version must be able to checkpoint intermediate results between toys. Finally, implementors are strongly encouraged to include some means of visualizing the output or evolution of individual toys.

The file formats used in the Cowichan problems are shown in Figure 10. A file containing a vector begins with 1 positive integer, which specifies the number of elements in the vector; a file containing a matrix begins with 2 positive integers, which specify the number of rows and columns in the matrix respectively. Elements of the vector or matrix then appear one per line, from least-indexed to greatest-indexed (and in row-major order for matrices).

The formats described above are required to be human-readable. Implementations may also include I/O using "raw" files, i.e. binary data dumps rather than formatted ASCII, using whatever file formats are convenient. This will allow programmers to demonstrate the "natural" I/O capabilities of particular systems which most probably be used for checkpointing intermediate results in real programs.

6.3 Visualization

Visualization is an increasingly important part of scientific and numerical computing. Including it as an optional part of this problem set is also a good way to exercise both the use of libraries within a parallel programming system, and the ease with which a programming system can support variant versions of a program.

Our requirements for the graphics interface are:

1. Visualization for individual toys should be as simple as possible, and should require as few changes to the basic toy code as possible.

```
integer matrix        real matrix        real vector
  rows cols             rows cols          length
  v(1, 1)               v(1, 1)            v(1)
  v(1, 2)               v(1, 2)            v(2)
    ⋮                     ⋮                ⋮
  v(1, cols)            v(1, cols)         v(length)
  v(2, 1)               v(2, 1)
  v(2, 2)               v(2, 2)
    ⋮                     ⋮
  v(rows, cols)         v(rows, cols)
```

Figure 10: Cowichan File Formats

2. The same code should be used for both the stand-alone versions of the toys and for the chained versions. The latter should create all images in a single window.

3. A single source file should implement both the graphical and non-graphical versions of each toy.

The first two points require the system to distinguish between the actual drawing code (e.g. drawing a vector of points as circles) and the place where the drawing has to be done in the window (called the *viewport*). While the stand-alone toys use a whole window for drawing, the chained versions may only draw in some portion of the window, e.g. the upper left corner. At present, this is implemented by using a call counter in the interfaces for the individual toys and in the chained module. The actual drawing routines called from within each toy could then be the same.

The present visualization routines were originally written by Anil Sukul, a student at the Vrije Universiteit, Amsterdam. The package uses the public domain graphics library VOGLE, available by anonymous FTP from `gondwana.ecr.mu.oz.au::/pub`. VOGLE includes functions for drawing points, lines, polygons, and text in a device-independent fashion, as well as screen control routines (such as double buffering). VOGLE was chosen because of its simplicity, and because it includes Pascal and Fortran interfaces.

6.4 Reproducibility

Reproducibility is an important issue for parallel programming systems. While constraining the order of operations in a parallel system to guarantee reproducibility makes programs in that system easier to debug, it can also reduce the expressiveness or performance of the system.

In this problem set, irreproducibility can appear in two forms: numerical and algorithmic. The first arises in toys such as `gauss`, `sor`, and `elastic`, which use floating-point numbers. Precisely how round-off errors occur in these calculations can depend on the distribution of work among processors, or the order in which those processors carry out particular operations.

Irreproducibility also arises in toys which only use exact numbers, such as `invperc` and `randmat`. In the former, the percolation region is grown by repeatedly filling the lowest-valued cell on its perimeter. If several cells have this value, implementations may choose one arbitrarily. Thus, different implementations may produce very different fractal shapes. In the case of random number generation, the simplest thing to do is to run the same generator independently on each processor, although the values in the resulting matrix then depend on the number of processors used.

7 Conclusion

A full ANSI C version of this problem suite has been written, and is available by anonymous FTP from:

`ftp.cs.toronto.edu ::  pub/gvw/cowichan/src.tar.gz`

This release includes two parallel versions, both based on the POSIX threads package. The first uses repeated fork-and-join parallelism, while the second creates a fixed number of threads at the beginning of the program, then synchronizes them repeatedly using barriers.

We would like to invite interested groups to participate in this project by parallelizing the problem suite using their favourite language or tool. Our aim is to make both the implementations and their descriptions available as a hypertext document through the World-Wide Web, and, if there is sufficient interest, to publish this material in book form. If you would like to contribute to this effort, please contact either of the first two authors.

References

[1] Robert G. Babb, editor. *Programming Parallel Processors*. Addison-Wesley, 1988.

[2] B. Bacci, M. Danelutto, and S. Pelagatti. Resource Optimisation via Structured Parallel Programming. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems.* Birkhäuser Verlag AG, April 1994.

[3] David H. Bailey, Eric Barszcz, Leonardo Dagum, and Horst D. Simon. NAS Parallel Benchmark Results. *IEEE Parallel & Distributed Technology*, February 1993.

[4] Henri E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3), 1989.

[5] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the 1990 Conference on Principles and Practice of Parallel Programming.* ACM Press, 1990.

[6] François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2:7–22, 1994.

[7] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs.* MIT Press, 1993.

[8] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[9] Boelie Elzen and Donald MacKenzie. The Social Limits of Speed: The Development and Use of Supercomputers. *IEEE Annals of the History of Computing*, 16(1), 1994.

[10] Norman E. Fenton. *Software Metrics: A Rigorous Approach.* Chapman and Hall, 1991.

[11] John T. Feo. An Analysis of the Computational and Parallel Complexity of the Livermore Loops. *Parallel Computing*, 7(2):163–85, 1988.

[12] John T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems.* North-Holland, 1992.

[13] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, 10:349–66, 1990.

[14] Alan R. Feuer and Narain H. Gehani, editors. *Comparing & Assessing Programming Languages: Ada, C, Pascal.* Prentice-Hall, 1984.

[15] I. Foster and K. M. Chandy. FORTRAN M: A Language for Modular Parallel Programming. Technical Report MCS-P327-0992, Argonne National Laboratory, 1992.

[16] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel and Distributed Technology*, 1(3), August 1993.

[17] T. R. G. Green, R. K. E. Bellamy, and J. M. Parker. Parsing and Gnisrap: A Model of Device Use. In Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, editors, *Empirical Studies of Programmers: Second Workshop*. Ablex, 1987.

[18] John Gustafson, Diane Rover, Stephen Elbert, and Michael Carter. The Design of a Scalable, Fixed-Time Computer Benchmark. *Journal of Parallel and Distributed Computing*, 12:388–401, 1991.

[19] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[20] Roger Hockney. Performance Parameters and Benchmarking of Supercomputers. *Parallel Computing*, 17:1111–30, 1991.

[21] Inmos Ltd. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[22] Do-While Jones. *Ada in Action*. Wiley, 1989.

[23] David B. Loveman. High Performance Fortran. *IEEE Parallel & Distributed Technology*, 1(1), February 1993.

[24] Evelyn Sander, Leonard M. Sander, and Robert M. Ziff. Fractals and Fractal Correlations. *Computer in Physics*, 8(4):420–25, 1994.

[25] Jonathan Schaeffer, Duane Szafron, Greg Lobe, and Ian Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel & Distributed Technology*, 1(3):85–96, 1993.

[26] B. A. Sheil. The Psychological Study of Programming. *ACM Computing Surveys*, 13(1), March 1981.

[27] Martin W. Simmen. Parameter Sensitivity of the Elastic Net Approach to the Travelling Salesman Problem. *Neural Computation*, 3:363–74, 1991.

[28] Paolo A. G. Sivilotti. A Verified Integration of Parallel Programming Paradigms in CC++. In Howard Jay Siegel, editor, *Proc. Eighth International Parallel Processing Symposium*. IEEE Computer Society Press, April 1994.

[29] D. Stauffer and A. Aharony. *Introduction to Percolation Theory*. Taylor and Francis, 1992.

[30] Mark Thomas and Stuart Zweben. The Effects of Program-Dependent and Program-Independent Deletions on Software Cloze Tests. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers (Proceedings of the First Workshop on Empirical Studies of Programmers)*. Ablex, 1986.

[31] David W. Walker. The Design of a Standard Message-Passing Interface for Distributed Memory Concurrent Computers. *Parallel Computing*, 20(4), April 1994.

[32] Reinhold P. Weicker. An Overview of Common Benchmarks. *IEEE Computer*, December 1990.

[33] Susan Wiedenbeck. Processes in Computer Program Comprehension. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers (Proceedings of the First Workshop on Empirical Studies of Programmers)*. Ablex, 1986.

[34] Gregory V. Wilson. Assessing the Usability of Parallel Programming Systems: The Cowichan Problems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*. Birkhäuser Verlag AG, April 1994.

A Parallel Clichés

A.1 Operations

This list details some operations which are supported by many parallel programming systems. The toys described in Section 5 provide opportunities for exercising many of these, and implementors are encouraged to phrase discussion of their work in terms of these operations where appropriate.

- elementwise operations on arrays (unary, binary, and scalar promotion)
- cumulative operations on arrays (reduction and parallel prefix)
- array re-shaping and re-sizing (e.g. sub-array extraction)
- partial (conditionally masked) versions of all of the above
- regular data motion (e.g. circular and planar shifting)
- irregular data motion (e.g. 1-to-1, 1-to-many, and many-to-1 permutation)
- scalar and non-scalar value location (e.g. finding a scalar or record value in an array)
- differential local addressing (i.e. subscripting an array with an array)
- full or partial replication of shared read-only values
- full or partial replication of shared read-mostly values with automatic consistency management
- structures with rotating ownership, suitable for migration
- producer-consumer structures
- partitionable structures
- pre-determined run-time re-partitioning (i.e. re-distributing an array)
- dynamic re-partitioning (e.g. for load balancing)
- committed mutual exclusion (e.g. waiting on a semaphore)
- uncommitted mutual exclusion (e.g. lock or fail)
- barrier synchronization
- multiple concurrent barriers used by non-overlapping groups of processes
- fetch-and-add, and other fetch-and-operate functions
- pre-scheduled receipt of messages of known composition
- variable-length message receipt
- message selection by type
- message selection by source
- message selection by contents
- guarded message selection (e.g. CSP's `alt` construct)
- broadcast and partial broadcast

- split-phase (non-blocking) operations
- data marshalling and unmarshalling of "flat" structures (e.g. arrays of scalars)
- data marshalling and unmarshalling of nested or linked structures
- heterogeneous parallelism (i.e. running different applications concurrently in one machine)
- pipelining
- distributed linked structures in systems without physically-shared memory
- indexing of distributed shared arrays
- collective I/O
- uniform-size interleaved I/O
- heterogeneous-size interleaved I/O
- independent I/O operations on a single shared file

A.2 Memory Reference Patterns

These are inspired by the categorization used in the Munin system [5]. Again, implementors are encouraged to phrase discussion of their work in these terms where appropriate.

- Write-once: variables which are assigned a value once, and only read thereafter. These can be supported through replication.
- Write-many: variables which are written and read many times. If several processes write to the variable concurrently, they typically write to different portions of it.
- Producer-consumer: variables which are written to by one or more objects, and read by one or more other objects. Entries in a shared queue, or a mailbox used for communication between two processes, are examples.
- Private: variables which are potentially shared, but actually private. The interior points of a mesh in a program which uses geometric decomposition fall into this category, while boundary points belong to the previous class.
- Migratory: variables which are read and written many times in succession by a single process before ownership passes to another process. The structures representing particles in an N-body simulation are the clearest example of this category.
- Result: Accumulators which are updated many times by different processes, but thereafter read.
- Read-mostly: Any variable which is read much more often than it is written to. The best score so far in a search problem is a typical instance of this class: while any process might update it, in practice processes read its value much more often than they change it.

- Synchronization: Variables used to force explicit synchronization in a program, such as semaphores and locks. These are typically ignored for long periods, and then the subject of intense bursts of access activity.
- General read-write: Any variable which cannot be put in one of the above categories.

B Comparing Parallel Programming Systems

This list of questions is a modified version of the one presented in [14]. It is presented here as an organizational aid; when describing the PPS they have used, implementors are encouraged to use the order and terminology of this list where appropriate. We acknowledge that many of these questions will not apply to particular systems, and encourage implementors to ask questions which we have left out.

1. History and Philosophy
   (a) For whom and for what purpose was the language designed?
2. Syntax
   (a) Model implemented (Figure 1)
   (b) Class
      i. library called from existing sequential language?
      ii. syntactic extensions to existing sequential language?
      iii. new language in Backus (Fortran/Algol/C) tradition?
      iv. new (non-Backus) language?
   (c) Concurrency mechanisms adhere to same philosophy as original language (where appropriate)?
   (d) Readability? Syntactic consistency? Conciseness? (on an APL-to-Modula scale)
   (e) Common errors (e.g. confusing = and == in C)?
3. Type Philosophy and Data Types
   (a) What primitive types does the language support?
   (b) How are new types created?
   (c) What intrinsic operations are defined on user-defined types?
   (d) How strict is typing? How is type conversion done?
   (e) Can type information be inspected at run-time?
   (f) What concurrent operations on aggregates (such as lists and arrays) are intrinsic? What kinds of concurrent aggregate operations can users define?
   (g) How are aggregates defined (e.g. by index set, by shape)? Can new kinds of aggregates (e.g. templates for arrays) be defined?
   (h) How strict is typing during communication?

- (i) What support is there for communication or manipulation of data of non-scalar or non-intrinsic types?
- (j) Can structural information about aggregates (e.g. number of elements in a set, dimension of an array) be inspected at run-time?
- (k) What built-in support is there for object classes and inheritance? For polymorphism? For class (as opposed to instance) data?
- (l) Are functions first-class?
- (m) Are continuations supported? Are they first-class?
- (n) Is dynamically-allocated memory supported? Is it done automatically? Is garbage-collection done automatically? Can a program ever create a dangling pointer? An alias?

4. Operators
   - (a) What intrinsic operators are provided (brief list)? How strongly do these type-check their arguments?
   - (b) How are new operators defined?
   - (c) Are the precedence rules straightforward?
   - (d) Are side-effects/mutation allowed inside non-call expressions (e.g. C's side-effecting `i++` construct)?
   - (e) Are their results platform-independent?
   - (f) Are the results of intrinsic parallel operations reproducible? Are they platform-independent?
   - (g) Are heterogeneous operations on aggregates allowed (e.g. every element of a list of polygons calculates its center of mass)? If so, how are these described, and how are they implemented?
   - (h) Which of the operators in Section A.1 are supported?
5. Control Flow
   - (a) What synchronization is automatically imposed during execution (i.e. at what points can definite assertions about the state of the running program be made)?
   - (b) What control constructs does the language support? Encourage?
   - (c) How is concurrency expressed? How structured is this mechanism?
   - (d) Can the degree of concurrency be throttled (limited)? Is this done by the system, or by the user?
   - (e) How are exceptions handled/reported? Is it different for intrinsic operations and user-defined operations?
6. Subroutines, Scope, and Modularization

(a) How efficient is subroutine call compared to in-line execution (i.e. how expensive are parameter passing and non-local control flow)?

(b) How much information about parameters is available (required) inside called subroutine (e.g. whether object is shared with other processes or private to executing process)?

(c) Are side-effecting procedures allowed?

(d) Can generic routines be written (e.g. to work on different types or array shapes)?

(e) Is there a restriction on return values (e.g. can a function return any object which could be declared)?

(f) Are there levels of scope? If multiple levels are permitted, how are references to stale scopes handled or prevented? Can information for declarations (such as size) be inherited (dynamically) from higher scopes?

(g) What synchronization requirements (if any) are imposed during calls (i.e. do all processes (in a group?) have to enter/exit the same subroutine at the same time)?

(h) What support is there for modularization? For hiding implementation details? For sharing such information? For detecting and resolving name clashes?

7. Concurrent Programming Facilities

(a) At what level(s) is the system concurrent?

    i. implicit (the user shouldn't ask such things)
    ii. operational (e.g. data-parallel addition)
    iii. program counter (single executable, multiple states)
    iv. procedural (heterogeneous control parallelism)
        A. one execution thread per address space
        B. multiple threads per address space

(b) What mutual exclusion and synchronization primitives are provided?

(c) At what level(s) is data sharing visible to the programmer?

    i. sequential consistency
    ii. sequential consistency within processor clusters
    iii. object-level consistency
    iv. there is no sharing

(d) What is the conceptual granularity of operations?

    i. synchronization after atomic instruction
    ii. homogeneous operations with large-grain synchronization
    iii. heterogeneous operations with large-grain synchronization

iv. completely asynchronous execution

(e) How do processes communicate data values? How do they synchronize? Are non-blocking operations (ones in which buffers might be in a visibly volatile state) allowed? How are they checked for termination?

(f) If data are communicated explicitly, how does the receiver choose what to accept?

   i. by address (e.g. channel or port)
   ii. by pattern matching on contents
   iii. by explicit message tagging
   iv. by provision of a typed buffer
   v. by provision of an untyped (byte-block) buffer
   vi. in FIFO order

(g) How do users specify the number of processors on which to run? Must this be built into the program, or can it be decided at run-time? Is there support for executing on heterogeneous processors?

(h) Can the user specify the mapping of processes to processors? If so, how? How tightly coupled are the mapping of processes to processors, and the mapping of distributed data structures (if supported)?

(i) Can the user specify the mapping of data structures to processors? Can data structures be decomposed or partitioned?

(j) What support is there for operations on groups of processes, such as:

   i. broadcast
   ii. multicast (partial broadcast)
   iii. barrier synchronization

(k) Can arbitrary process groups be created, or must they be structured in some way (e.g. as logical mesh)? Can operations involving disjoint groups be executed concurrently (e.g. a broadcast within one group, and a barrier within another)?

8. Input/Output

   (a) Is simple sequential (single-process) I/O simple to write?

   (b) Is concurrent I/O supported? If so, which flavours:

      i. single-reader-broadcast/collect-single-writer
      ii. homogeneously-sized or -strided segments
      iii. heterogeneously-sized or -strided segments
      iv. arbitrary independent operations

   (c) What types can be read/written intrinsically? E.g. can an entire record (or array, or list) be read or written in a single operation?

9. Access to Routines in Other Languages and to the Hardware
   (a) Can subroutines written in other languages be called? Can subroutines in this language be called from other languages?
   (b) Are there ways to access the run-time system directly, or similar low-level facilities?
   (c) Is there high-level access to physical addresses and devices?
10. Practical Considerations
    (a) Are both batch and interactive processing supported?
    (b) How reliable and efficient are existing implementations? How fast is it (i.e. how quickly does code compile and load)? Is it supported (who fixes it if it breaks)?
    (c) Is there a standard for the system (from some standards organization)?
    (d) Is the documentation adequate? For whom (CS graduates, computer-literate users, the general population)? Is training available? Consulting?
    (e) How much code has been written (lines of code or weeks of effort)? By whom (computer science undergraduates, professional engineers, etc.)?
    (f) How long does it take the compiler to generate a "small" executable (such as `"hello, world"`)?
    (g) How large is a "small" executable? How long does it take to load such an executable?
11. Supporting Tools
    (a) What is the development environment like? Can code be developed on a workstation and then moved to a parallel platform?
    (b) Are there debugging aids, performance analysers, and management tools?
    (c) How much help do these really provide?

C A Critique of the C Implementation

This section is a critique of the ANSI C reference implementation of the problem suite. It is intended to serve as a model for (self-)criticism of other programming systems.

- C's support for multi-dimensional arrays (MDAs) is very weak. There is no way to dynamically allocate an MDA in a single go—one must either allocate a block of the same size as the desired array, and then do indexing calculations by hand, or allocate a vector of pointers to allocated vectors of pointers to...to vectors holding data. MDAs do not carry dimension information with them, so it is impossible to determine the size of an array parameter within a function. Finally, C does not treat all axes of an array equally: while it is trivial to take a slice out of a 2-dimensional array along the most-significant axis, it is impossible to slice it along the other axis.

- C does not distinguish between Boolean and integer types. As a result, the `matrix` and `mask` arguments to the invasion percolation problem can be passed in reverse order without a type error. Using `typedef` to create a Boolean type name does not solve this (at least, not in `gcc` V2.5.8).

- Union types cannot be safely initialized. This complicated the implementation of the graphics interface, where it would have been much more elegant to define a union type, each of whose variants held parameter specifications for a single toy. The code in the graphics module `gfx.c` relies instead on arrays of `int`s and `float`s, initializing some and filling other with don't-care values. This is neither safe nor elegant.

- Parameter values cannot be used in the initialization of local variables inside functions. In particular, it is not possible to create a local vector with a length specified by an input parameter. Such a facility would be useful in the `winnow` toy, where we have instead allocated local temporaries of the maximum possible size.

- The automatic conversion of floats to doubles across function calls can quite often be a nuisance. For example, the "fail()" error-handling routine was buggy because all `real` arguments were being automatically converted to doubles, but were being taken off the stack as floats.

- no intrinsic notion of group ID/self ID in threads
- packaging parameters for threads
- pointers to array vs. arrays themselves (semantics of definition depends upon context)