The Purdue Parallel Benchmarks in FP

Randolph Bentson

Technical Report CS-93-126

October 19, 1993

The Purdue Parallel Benchmarks in FP

Randolph Bentson

Technical Report CS-93-126

October 16, 1993

Abstract

In [Ric80], John Rice proposed a list of sixteen problems to be used to test the expressive power and performance of languages in parallel computational environments. Since then, there have been several followup reports. One from Purdue, [MR87], examines the Fortran extensions on various architectures. Two CSU reports, [HHRO88] and [AHO90], have looked at the language SISAL. A recent report from Purdue, [RJ90], adds a seventeenth problem to the set.

This report takes up the challenge by looking at how well an implementation of Backus' FP, TFP [Ben92], can express a solution and how quickly that solution can be computed. Because of the fine-grain parallelism available in the graph reduction method used by TFP, dramatic parallelism is available for complex programs. Modest sized problems, such as computing polynomial interpolant values over 32 points (problem 7) saw an average parallelism of over 500 and a peak of ten times that. Simple programs did not fare as well, due mostly to the nearly linear nature of some of the functions and PFOs. Certain functions and PFOs, such as `trans`, `append`, and `\/`, carry significant serialization in their implementation, and therefore can slow down programs that use them.

This suggests that the linked-list implementation of sequences can be a handicap for simple programs, but this is masked whenever the program complexity becomes sufficiently large.

1 Language Description

The computational model expressed in Fortran is often referred to as von Neuman style. This language style makes explicit references to memory. The state of the computation, as it proceeds, is described in the language as changes in values of variables. It is incumbent upon the programmer and the compiler to access variables in the appropriate sequence. The language forces one to specify the order of operations to a degree far beyond the minimum needed to arrive at a solution. This over-specification actually interferes with attempts by the compiler to discover parallelism. Most will acknowledge that this style of programming makes it difficult to identify the maximum parallelism allowed by the problem.

Languages of this style have been extended to make it easier for the programmer to code a solution and for the compiler to discover parallelism. One common extension is the provision of objects such as vectors and arrays that can be operated on via simple operators. The compiler can then determine the best way to implement such an operator. Another extension is to allow the programmer to specify many distinct tasks which work co-operatively.

Another style of language, called functional or applicative, extended this idea that parallel tasks should be easily derived from the program. In these languages, the tasks are not specified directly by the programmer, but rather are derived directly from the property of the languages that state the order of the evaluation of expressions does not yield different results. The compiler and run-time can then create tasks out of the evaluation of expressions and functions whenever there is benefit in doing so.

SISAL is the result of one of the efforts to expose more parallelism to the compiler without burdening the programmer. It does so, in part, by removing the issue of changing variable values. Value holding identifiers in SISAL have a single value assigned to them. Until that assignment, the value can be thought of as a special value "not yet known". Programs are expressed in terms of expressions built up from functions and identifiers. Additional language constructs support aggregrate operations on arrays, etc. It appears that SISAL is reasonably concise in expression and efficient in execution.

FP is an even more dramatic departure from the von Neuman language style. It is purely functional, with no names other than function names. All functions take a single object as an argument and return a single object as a result. This is not a hardship because an object is an atom or a sequence of atoms and/or sequences, and can describe arbitrarily complex data. In fact, one often speaks of "input values" when one should say "values in the input sequence". Because this mis-statement is understood, it normally goes unremarked. Any expression of state must be explicit, i.e., the state of computation is passed as an argument to a function and the function returns the value of the new state.

FP has several functional forms or program forming operations (PFOs) that are used to combine functions into more complex functions. Unlike functions which operate only on objects, these PFOs operate on functions as well as objects. The standard PFOs are condition, composition, construction, while-iteration, insertion (a reduction operation), application-to-all, and binding. These are not a minimal orthogonal set, but rather a large, strong set of operations likely to be sufficient to conveniently program any function desired. This is in contrast to many other functional languages which allow the programmer to define new program forming operations. By limiting the program forming operations, it is possible and reasonable to develop an algebra of programs. This enhances the ability to analyze an FP program.

FP was defined as a strict language, where all inputs to functions are evaluated before the function started. In addition, functions (and all PFOs except condition) return a special value, called "bottom", shown as `?` on displays or as $\perp$ when typeset, if their input contains bottom or if their results are undefined. The implementation considered in this report is not strict. As a consequence all programs that return a non-bottom result in a strict implementation will return the same non-bottom value in this version, but because a function need not return bottom as its value whenever bottom appears in its input sequence, additional non-bottom results can appear in a program's range.

The difference becomes apparent when one considers the function `1 o [f1,f2]:x`. In a strict system, if `f2:x` returns bottom, then `[f1,f2]:x` always returns bottom, and then `1:?` returns bottom.

If the construction PFO is non-strict, `1 o [f1,f2]:x` can have different values under different evaluation schemes. If `f2:x` is evaluated early, `[f1,f2]:x` is evaluated as bottom, and `1:?` is then evaluated as bottom. But if composition is applied before `f2:x` has been evaluated, the value `f1:x` would be selected from the partially evaluated sequence.

The TFP implementation allows function results to be represented by "continuations", or incompletely evaluated function applications. By this method, the composition PFO could return a sequence of continuations and the selection function could be evaluated before it `f2:x` is evaluated (and its results would therefore be discarded). But if we are to have results that are invariant under execution timing, we must always discard the results of `f2:x`, regardless of its status as a continuation or bottom. By this we see that non-strictness is a practical consequence of allowing continuations, and that continuations are a convenient way to develop non-strict implementations.

Lazy evaluation follows the principle that no function is evaluated until its results are needed. In a single processor system this reduces execution time by avoiding unneeded computation (provided the management overhead is small enough). The lazy evaluation of a strict language would save no time because all the parameters would be evaluated; therefore as a practical matter, all languages implemented with lazy evaluation are non-strict. But this does not mean that all non-strict languages use lazy evaluation.

TFP's use of non-strictness offers improved parallelism because operations such as composition and selection can proceed before all of the elements of their input sequence have been evaluated. In addition one can proceed with the parallel evaluation of as-yet-unneeded continuations, even while necessary continuations are being evaluated. This non-strict, not-fully-lazy evaluation scheme is herein called lenient evaluation.

2 Language Idioms

The body of FP code is small. Other functional languages with more conventional syntax are more commonly used. The comments here are based on experiences implementing solutions to the Purdue problem set.

Sometimes a solution was trivially derived from the problem statement. This usually occurred because a PFO captured the essence of the problem. In other cases more effort, such as writing non-trivial functions, was required. It is possible to in-line all non-recursive functions into a single large program. Because of restrictions on the application of higher-order functions on user defined functions, this was a common style in writing APL programs. Fortunately FP has no such restrictions, and because there is no performance cost in using functions, the final program is developed from many smaller functions. As a bonus, this provides for enhanced readability.

In general, one builds an initial value object and then pushes that object through successive functions (using the composition PFO) until a final value object is determined. Sometimes the same function is applied several times, in which case the use of the while-iteration PFO or recursion is appropriate.

Functions can be applied to all elements of the sequence using the `aa` PFO. This provides significant parallelism, especially when the functions are complex in nature. This leads to the style of moving the data about until one has a sequence of independent datasets that can be operated upon. In Problem 9 the input matrix is moved about until each element is replaced by a sequence containing that element and its neighbors. The averaging function is then applied to each of these sequences.

Often the last function sums the elements of the sequence. The PFOs `\`, `/`, and `\/` insert a binary operator such as `+` or `*` between terms of the series. Their use acts as a classic reduction operator.

The FP implementation being tested applies functions to objects as soon as the source object is sufficiently defined. In many cases the result object becomes defined in stages. For instance, `f1 o [f2,f3]:y` has an intermediate object that is known to be a sequence `<f2:y,f3:y>`, but the values of the elements may not be known. It is possible to apply `f1` to this sequence before the values are known and `f1` is delayed only as necessary.

Unfortunately this implementation isn't as lenient as it could be. Even if the values of the elements in a sequence may be represented by continuations, the number of elements in a sequence must be known before that sequence is made available as an output. PFOs such as `apndl` and `apndr` cannot be used in recursions to provide infinitely long sequences. This implementation restriction prevents one from using streams of unbounded data in programming.

Another shortcoming with this implementation is its failure to recognize common sub-expressions. As a consequence, the programmer is obliged to "hoist" those sub-expressions to outer levels if the extra computation is to be avoided.

3 Basis for Timing

This FP implementation is a multi-threaded graph reduction interpreter. The interpreter has been designed with respect to a possible implementation on a shared memory multiprocessor with multiple operations per instruction and low-overhead scheduling primitives, where each processor supports a number of execution threads. Although it is currently executed on a uniprocessor, it simulates parallel performance by way of timing simulated parallel execution. Command line arguments control the number of parallel processors being simulated, as well as the maximum number of threads on each processor.

Each reduction step computes the number of machine ticks that would be required if the reduction had been done on a multiprocessor. This timing information is used to keep track of when the function's results are to be made available for further processing, i.e., when the next function can be applied. Until this delay is satisfied, that function application counts as one of the active threads. When the number of active threads reaches the per-processor limit, additional continuations are placed in a ready queue. When an active thread completes, i.e., when enough time has elapsed so that the results are made available, another continuation thread is removed from the ready queue and is made active. The simulator moves from one simulated processor to the next, extracting threads from ready queues, computing function results, moving the thread to an active queue, advancing the simulated clock, etc. In this way, the system can determine how long the FP program would take to execute on a multiprocessor.

Although the architecture being modeled supports 256 processors $\times$ 128 threads/processor, tests can simulate more or fewer processors or threads/processor. In addition to varying the data set size, problems are examined with respect to the maximum number of available processors and threads per processor. In this way we can examine how effectively parallelism can be exploited. One subtle limit to performance is that although one can have many threads on a processor, performance does not increase linearly as threads are added. The first thread on a processor achieves about 1/12 of the processor's capability. This is because no thread may have more than one instruction in the pipeline at any moment. In a very busy system this utilization may be smaller because of memory access delays, but with a good compiler, the effect should be insignificant. As more active threads are added to a processor, utilization and aggregate performance increases until the pipeline is full. Adding more threads beyond this point actually reduces the per-thread performance, but because the thread scheduling is performed by hardware, there is no additional overhead to hurt aggregate performance. This is simulated by applying a slow down factor to all timing. The factor is computed as `slow_down = active_tasks < PipeLineSize ? 1.0 : active_tasks / PipeLineSize`; where PipeLineSize is normally 12.

Fourteen of the seventeen problems have been programmed in FP. The performance of each program is examined with respect to various data set sizes. The main dependent variable is the number of machine ticks required for solution. Additional dependent variables include the average and maximum number of active tasks, etc. Tests of of increasingly larger data sets shows that for any given data set size, adding processors and threads reduces execution time to a limit. As predicted by Amdahl's Law [Amd67], there is always a threshold below which execution time cannot be reduced, regardless of the available parallel hardware. In "good" parallel programs, the degree of parallelism grows with the problem set size, and the effect of the sequential component is lessened. In "bad" parallel programs, the sequential component grows with the problem set size, or the degree of parallelism does not grow, and thus the sequential component stays visible, or even dominates the computation.

One significant assumption in computing time in the TFP simulator is that the allocation and deallocation of cells (which contain objects and code definitions) takes some small, fixed amount of time. If all threads accessed a common pool of cells, this assumption would not be realistic due to blocking. If a (mostly) non-blocking allocation scheme is used, (such as a pool per processor) this assumption is reasonable.

4 Test Methods

Each program is run under the control of a test framework. The outer loop in the framework generally starts at one and doubles the problem set size until limits to parallelism are found, excessive paging occurs on a 16Mbyte system, or the simulation time becomes unbearable. An inner loop controls the number of processors being simulated and the threads on each. It generally starts at one and doubles until it equals the dataset size or shows the limits to parallelism.

In some cases the program uses the dataset size directly and in other cases the framework generates the test dataset via a test generation program. (And in the case of problem 16, the number of rows and columns is set to $\lg N$.) This dataset is then read as data to the program under test. If the test dataset generation were done within the program under test, the cost of that generation would be charged against the program, confusing performance issues.

The dataset generation is rather naive – the data are not randomly created and only slight effort is put into domain limiting.

The study has been largely restricted to nearly asymptotic performance, i.e., how much parallelism is available given unlimited processors (or at least all the processors in the system). As noted in the Performance sections of many of the problems, the underlying time complexity of any TFP program is $\mathcal{O}(N)$, or at best $\mathcal{O}(\lg N)$, because of limitations in accessing sequence elements. Since this time is masked if the processing of the elements takes long enough, or if the sequences are short enough, the "actual" performance reported is based on inspection of these limited results. This is what is meant by the "nearly asymptotic performance" mentioned earlier.

Another issue is how effectively fewer processors can speed-up the solution. When all goes well, one sees that speed-up grows as $P \times T$ while the number of threads/processor is less than or equal to the size of the pipeline, and beyond that point speed-up grows with the number of processors (because the processor is fully loaded).

4.1 Problem 1

4.1.1 Problem Description and Algorithm

The problem is to estimate the value of the integral of *f(x)* in the interval [a, b]. The formula is

$$T_N = h * (\frac{f(a) + f(b)}{2} + \sum_{i=1}^{N-1} f(a + i * h))$$

where N is the number of intervals in for the estimate and, $h = (b-a)/N$. Increasing N improves the accuracy of the estimate.[1]

4.1.2 Inherent Parallelism

The serial solution grows as $\mathcal{O}(N)$. The summation of all the terms can be done in parallel in $\mathcal{O}(\lg N)$ time using $\mathcal{O}(N/2)$ processors.

4.1.3 Characteristics of Translation

First a sequence (in this case the x_i values) is built, one then applies the function, and reduces the results (in this case by summation). The parallel algorithm contains extra multiplications to avoid some serial bottlenecks.

4.1.4 Performance

Although the single processor, single thread/processor execution time is $\mathcal{O}(N)$ as predicted, the parallel execution time grows nearly as fast, with speed-up of less than $\mathcal{O}(\lg N)$. Detailed analysis shows that the `distl` and `appndr` functions, and the `aa` PFO are a major bottleneck. Over one third of the execution time was spent distributing `<h,initial_value>` over the `iota o size` sequence. Since the current implementation does not make the resultant sequence available until the entire distribution is done, the computation of the segments is delayed until `distl` has completed. The `aa` and `\/` PFOs are similarly handicapped. In addition, the sequences are

[1]Thanks to the authors of [AHO90] for permission to copy the text of the "Problem Description and Algorithm" sub-sections from their report.

operated on via a linear scan instead of a data-parallel or tree structured manner that would yield constant time or logarithmic time performance.

These conjectures are tested by using the "OneTick" debugging option. This special feature of TFP allows certain suspect functions to be timed as if they took only one tick to execute. This was added because these functions take $\mathcal{O}(N)$ time due to the implementation of sequences as linked lists. Had sequences been implemented as trees, these operations could be performed in $\mathcal{O}(\lg N)$ time.

When this option is applied to this problem we see execution time drop dramatically, as much as twenty to one for $N = 256$.

4.2 Problem 2

4.2.1 Problem Description and Algorithm

This problem computes e^* by:

$$e^* = \sum_{i=1}^{n} \prod_{j=1}^{m} (1 + e^{(-|i-j|)})$$

Two nested loops with reduction operators solve this problem easily. The algorithm follows the formula above.

4.2.2 Inherent Parallelism

The serial complexity of this is simply $\mathcal{O}(N \times M)$. By using $N \times M$ processors, each of the terms can be computed in constant time. Each of the N sets of M terms can then be multiplied in $\mathcal{O}(\lg M)$ time using $\mathcal{O}(N)$ processors. The N results can then be added in $\mathcal{O}(\lg N)$ time using $\mathcal{O}(N)$ processors. As tested, M and N are given the same value, so the parallel time complexity simplifies to $\mathcal{O}(\lg N)$ time using $\mathcal{O}(N^2)$ processors.

4.2.3 Characteristics of Translation

Both the Σ and Π operations are easy to express in FP. The only shortcoming in the language is that one has to know that one has to pass the values of n and m to the inner function. As a consequence, one builds a sequence of pairs to which the function is applied. This requires some care, but it is not especially tricky.

4.2.4 Performance

Empirical results showing the time taken by a single processor matches nicely with theory.

However, when maximal parallelism is allowed, the time grows faster than $\mathcal{O}(\lg^2 N)$, but slower than $\mathcal{O}(N)$, while using fewer than N^2 processors. This performance was so disappointing that further analysis was done to find the reason. When the tests were re-run with `distl` and `distr` running in "OneTick", the times grew slower longer. This suggests that the data structure for sequences has again contributed to the low performance. (And it's worth noting that the `iota` function appears several times in this program, and it is also linear in its implementation.)

4.3 Problem 3

4.3.1 Problem Description and Algorithm

Compute the value of

$$S = \sum_{i=1}^{n} \prod_{j=1}^{m} a_{ij}$$

This is a straightforward problem that multiplies elements of each row and adds each of the products to a final sum. The algorithm strips off each row for product formation in an inner loop and then sums them in an outer loop.

4.3.2 Inherent Parallelism

The serial complexity is $\mathcal{O}(N \cdot M)$, but with $N \cdot M/2$ processors one can compute all the terms to the summation in $\mathcal{O}(\lg M)$ time, and with $N/2$ processors one can sum in $\mathcal{O}(\lg N)$ time.

4.3.3 Characteristics of Translation

This is close to Problem 2 in expression, but the programming is trivial. This is because the issue of passing parameters to an inner function is missing. In addition, because the input matrix is completely specified, one needn't generate a sequence of values for the internal function.

4.3.4 Performance

Empirical results showing the time taken by a single processor are slightly greater than theory predicted, but less than $\mathcal{O}(N^2 \lg N)$.

When maximal parallelism is allowed, the time grows faster $\mathcal{O}(\lg^2 N)$. In this case, the PFO `\/` is suspect. When the function being inserted is simple, as is the case here, the behavior of the `\/` PFO becomes visible. When more complex functions are being inserted, or when the evaluation of the terms is complex, this delay is masked.

4.4 Problem 4

4.4.1 Problem Description and Algorithm

Compute the value of

$$R = \sum_{\substack{i=1 \\ x_i \neq 0}}^{n} \frac{1}{x_i}$$

This problem sums operands whose denominators are nonzero.

4.4.2 Inherent Parallelism

The serial complexity is $\mathcal{O}(N)$, but when using N processors, the N terms can be computed in parallel, and then $N/2$ processors can be used to sum the terms in $\mathcal{O}(\lg N)$ time.

4.4.3 Characteristics of Translation

An input sequence of the x_i is generated. The function is then applied to all terms in parallel and the sum taken.

4.4.4 Performance

While the single processor/single thread execution times grow only slightly faster than N, the maximally parallel performance is very poor. For reasons given with previous problems, the rather serial implementation of the \/ PFO comes to dominate the program. With the simple function being applied, parallelism grows at an abysmal $\mathcal{O}(\lg N)$ rate.

4.5 Problem 5

4.5.1 Problem Description and Algorithm

Given a table of the i^{th} student's scores on the u^{th} test, perform the following computations:

(a) list the top score for each student
(b) give the number of scores above the average
(c) increase all the above average scores by 10%
(d) give the lowest score that is above average
(e) note whether any student has all scores above average

The algorithm for this problem takes a two-dimensional array of grades and computes a new matrix of grades whose above average scores have been increased by 10% (c). I assumed that the intent of the computation of the lowest score above the average (d) was to be done prior to increasing the scores by 10%.

4.5.2 Inherent Parallelism

There are a number of reduction operations on arrays, compute average, find top, test against average, etc., suggesting the problem's parallel complexity is $\mathcal{O}(\lg N)$. While the operations on the array elements are highly parallel, there are a number of sequential restrictions throughout the computations. For example, the average score for all tests and students must be computed prior to computing the number of scores above the average (b) or increasing all scores above the average by 10% (c). Thus this problem is a good example of several highly parallel operations separated by barrier points at which the execution must be sequential. Fortunately these barriers are fixed in number, so the complexity measure is unchanged.[2]

4.5.3 Characteristics of Translation

The problem statement was ambiguous in many ways, some of which could affect the program structure, e.g. is the top score from the original dataset, or after step (c) has been applied. Once these were resolved, the programming was straightforward.

I often found myself drawing pictures of the current sequence just to keep track of the where the data were. This small handicap in the language is largely compensated by the fact that once it's right, changing input data sizes doesn't affect the solution.

4.5.4 Performance

The single processor/single thread execution time grows as N^2 – this is not surprising as the test dataset had the number of scores grow at the same rate as the number of students. The maximally parallel execution times grow faster than expected – far beyond $\mathcal{O}(\lg N)$ or even $\mathcal{O}(\lg^2 N)$. Again, the processing of sequences in PFOs and functions, especially `trans`, are suspect.

4.6 Problem 6

4.6.1 Problem Description and Algorithm

The description of the algorithm was unavailable beyond its implementation in FORTRAN.

[2]Again, thanks to the authors of [AHO90] for permission to copy much of this analysis from their report.

4.6.2 Inherent Parallelism

While it is possible to solve a tridiagonal system, the "vector oriented algorithm of Jordan" contains trickery that involves violation of array bounds and some assumptions about storage that are wholely inappropriate in a language such as FP.

While it is suspected that the available parallelism falls somewhere between elimination methods and iterative methods, no formal analysis has been done.

4.6.3 Characteristics of Translation

This problem has not been implemented in FP.

4.7 Problem 7

4.7.1 Problem Description and Algorithm

The problem is to compute polynomial interpolent values of *f(x)* using Lagrange interpolation formulae, given by:

$$p(x) = \sum_{i=1}^{N} f(x_i) l_i(x) \text{ with } l_i(x) = \prod_{\substack{j=1 \\ i \neq j}}^{N} (x - x_j) / \prod_{\substack{j=1 \\ i \neq j}}^{N} (x_i - x_j)$$

4.7.2 Inherent Parallelism

This has $\mathcal{O}(N^2)$ serial complexity, but with $\mathcal{O}(N^2)$ processors the $l_i(x)$ terms can be computed in $\mathcal{O}(\lg N)$ time and then summed in $\mathcal{O}(\lg N)$ time.

4.7.3 Characteristics of Translation

The recurring idiom of building a vector or a matrix of indices and then computing the function at those points, is used in this program.

An initial program version was derived directly from the formula. It did not, however, retain the value of the $f(x_i)/(x_i - x_j)$ term. The code in the final program does, but it doesn't yet exploit this in computing the results for the multiple input values.

It's interesting to note that the two program versions return slightly different values because of the sum–then–div in one and the div–then–sum in the other.

4.7.4 Performance

Serial performance agrees with theory. The execution times for parallel execution grew as $\mathcal{O}(N)$ (as parallelism grew only sightly better than N). Using the OneTick option for a number of the functions and PFOs kept the parallel execution times in the $\mathcal{O}(\lg N)$ realm longer, but the system eventually succumbs to the access time for linear sequences.

4.8 Problem 8

4.8.1 Problem Description and Algorithm

These formulae define the divided difference table for a set of data x_i, y_i = f(x_i):

$$f[x_i] = y_i$$

$$f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_{i+k-1}}$$

The problem is to compute the first M columns of the divided difference table

$$D_{ik} = f[x_i, x_{i+1}, \ldots, x_{i+k-1}]$$

4.8.2 Inherent Parallelism

Each new column of the divided difference table is derived from the previous column of the difference table. It can be computed in constant time using N processors. Even if each element of that column were to be computed directly from the first column (the y_i values), the computation performance would be limited by the number of stages needed in the computation. No speed-up would be found in this approach, so we compute each column after the previous one is done.

The maximum parallelism is found in computing the first divided difference column. Remaining columns have less available parallelism.

4.8.3 Characteristics of Translation

Unlike FORTRAN or SISAL, it's rather difficult in FP to index into a sequence representing a column or row. The FP solution is to extract the previous column twice, shift one of the two columns, transpose, and subtract. The rest of the code is "glue" to hold the program together.

4.8.4 Performance

Empirical results show the single processor /single thread execution time grows much faster than theory suggests, $\mathcal{O}(N^2)$ rather than $\mathcal{O}(N)$.

The parallel performance is similarly sluggish. Average parallelism and execution times grow by $\mathcal{O}(N)$. The continued manipulation of the intermediate results by the `append` function costs dearly. Although this program produces the correct result, it is a good candidate for optimization such as computing all the columns simultaneously. Such a version would avoid almost all the copying that is being performed by the `append` function.

4.9 Problem 9

4.9.1 Problem Description and Algorithm

This problem is to smooth a matrix of real values. The computation is useful in image processing, solving differential equations, and geometric modeling and therefore is typical of much scientific computing. The matrix is smoothed by taking the average of each cell in the matrix with all its neighbors. That is,

$$u_{ij} = (\sum_{Neighbors} u_{ij})/(Number\ of\ neighbors)$$

4.9.2 Inherent Parallelism

For an $N \times N$ matrix, serial computation is $\mathcal{O}(N^2)$ and parallel computation takes constant time with N^2 processors.

4.9.3 Characteristics of Translation

As mentioned in Section 2, Language Idioms, this matrix is replicated nine times and each copy is shifted appropriately so that an average across the Z axis yields the appropriate result.

The boundaries caused a few problems in that the problem statement only implied what was to be done. The programming would have been a little easier if the `trans` function allowed irregular shapes to be transposed. As it was, significant effort is expended to "fill in" the irregularities.

4.9.4 Performance

Because of the boundary problems mentioned above, functions such as `tl` and `length` were needed to adjust sequences. These functions prevented the program from achieving its full potential. In spite of this, parallelism still grew much faster than $\mathcal{O}(N \lg N)$. N.B. This is a deceptive improvement because of the tremendous overhead in the shifting. The modified `trans` function mentioned above would be a much cleaner implementation.

4.10 Problem 10

4.10.1 Problem Description and Algorithm

The problem is LU factorization of an $N \times N$ matrix $A = a_{ij}$ using Gaussian elimination with partial pivoting.

4.10.2 Inherent Parallelism

Although finding the pivot element (maximum element in a column) can be done in parallel, and adjusting the remaining rows can be done in parallel, This problem has a fundamentally serial nature. One cannot find a pivot row K until the processing of row $K-1$ has been completed.

In spite of these recurring barriers, there still would be sufficient parallelism, but for the costs of copying matrices as the rows are shifted. A shuffle function that pays careful attention to reference counts, so it could "copy in place", would prove very useful.

4.10.3 Characteristics of Translation

This problem has not been implemented in FP.

4.11 Problem 11

4.11.1 Problem Description and Algorithm

The problem is to read a set of N real numbers n_i, trim the numbers to fall within the range [0,1000], and apply a logarithmic transformation $d_i = log(1+n_i)$, and then compute the first four Fourier moments according to the formula:

$$fourier - moment(k) = (\sum_{i=1}^{N} d_i \cos\,(\pi i/(k+1)))/N$$

where k is 1,2,3 or 4.

4.11.2 Inherent Parallelism

This problem contains significant inherent parallelism. With $\mathcal{O}(N)$ processors, each term of the sum in each Fourier moment can be computed in parallel in constant time. The summation then takes $\mathcal{O}(\lg N)$ time.

4.11.3 Characteristics of Translation

The problem was easily divided into three phases: one to limit ranges, the next to distribute vector length and index across vector values, and finally one to compute moments. The only problem is in the large degree of indentation that shows up in the program listing.

4.11.4 Performance

Although it was possible to construct the test data with the index and array size bound to each value, it wouldn't have been a proper use of the expressive power of the language. Instead the vector is first built and is then processed.

Serial execution times grew slightly faster than the $\mathcal{O}(N)$ predicted by theory. Parallel execution times grew slightly slower than $\mathcal{O}(N)$. The average parallelism grew a bit faster than $\mathcal{O}(\lg N)$. A brief check showed that the shuffle function (shown in the program listing) uses up over half the time (due to the linear nature of `trans`).

4.12 Problem 12

4.12.1 Problem Description and Algorithm

This problem is similar to Problem 9 in that a matrix is built from a previously existing matrix. But here the new matrix results from a smaller matrix, two vectors, and a scalar. Rice states this problem as:

Given the $m \times m$ matrix A, the $1 \times m$ vector R, the $m \times 1$ vector C, and a number a, build the array

$$ABIG = \begin{bmatrix} AC \\ Ra \end{bmatrix}$$

4.12.2 Inherent Parallelism

In serial execution, $\mathcal{O}(N^2)$ time is needed to copy the N^2 elements. In a data parallel system, this could be done in constant time. This implementation's parallelism is limited only by the ability to build a sequence and then populate its elements. The `apndr` and `append` functions are linear in this version, but could be $\mathcal{O}(\lg N)$ if the sequences were tree structured.

4.12.3 Characteristics of Translation

The code so easily captures the solution that the program is shorter than the description. It seems almost too easy, but it works!

4.12.4 Performance

Serial execution runs in $\mathcal{O}(N^2)$ time as predicted. Parallel execution times show only a modest 10 to 1 improvement over serial execution. The serialization in `append`, `trans`, and `apndr` is the culprit. The OneTick option only pulls the times down to $\mathcal{O}(N)$ because the `trans` function really isn't speeded up that much due to its internal use of selection functions.

4.13 Problem 13

4.13.1 Problem Description and Algorithm

For vectors a, b, c, and d, recompute a with the procedure:

$$a_i = a_i^{sin(b_i)}$$

If $(a_i < cos(b_i))$ then $a_i = a_i + c_i$
else $a_i = a_i - d_i$

and compute

$$e = \sum_{j=1}^{ndim} a_j^2$$

The algorithm reads the a_i array, creating a new value for each instantiation of a_i and finally returns a new vector and a sum. N.B. See the comments about documented versus actual formulae in the section "Characteristics of Translation."

4.13.2 Inherent Parallelism

Each of the new a_i elements can be computed in constant time using $\mathcal{O}(N)$ processors. The summation then takes $\mathcal{O}(\lg N)$ time.

4.13.3 Characteristics of Translation

As with many of these problems, the dataset is "turned sideways" and the operations are then applied to all of the tuples in parallel.

The original Rice paper said `ai = ai*sin(bi)`, but the Fortran code says `ai = ai**sin(bi)`. Later the papers test `ai<cos(ci)`, but the code tests `sin(ai)<cos(ci)`. This program matches the operations of the actual code.

A "DOMAIN ERROR" forced the insertion of `abs o` in the computation of `ai**sin(bi)` to ensure that the power function works properly.

4.13.4 Performance

Serial execution times grew a little faster than $\mathcal{O}(N)$ Unfortunately parallel execution times grew nearly as fast as $\mathcal{O}(N)$. Average parallelism grew as $\mathcal{O}(\lg N)$. As with Problem 11, a brief check of `trans` showed it to cause much of the delay.

I tried pulling the `[id,id]` into the conditional as `[+,+]` and `[-,-]` to little effect. This suggests that the predicate is known soon enough to let the `[id,id]` pick up a conditional. Or possibly the single tick improvement was lost in the noise.

4.14 Problem 14

4.14.1 Problem Description and Algorithm

This problem carries out a test of three methods to integrate three separate functions with 50 different levels of accuracies. The original problem used four methods; trapezoidal integration, Simpson's rule, 3-point Gaussian integration, and the DCADRE specialized algorithm. I decided that the first two were enough for testing parallelism in this type of environment.

4.14.2 Inherent Parallelism

In this problem, the problem set size is the number of intervals over which the functions are to be integrated.

Full parallelism is found at the outer level because each test is independent. Within each test, each term of the numerical integration can be computed in constant time using N parallel tasks. The terms can then be added in $\mathcal{O}(\lg N)$ time.

4.14.3 Characteristics of Translation

This shows one major limitation of FP as a language. The decision to prohibit user defined PFOs prevents one from defining an integration PFO that is applied to a function and object. Instead, one must re-write the integration function for each of the test functions.

Another approach would be to "pass the function" via the name binding mechanism, invoking the program once for each of the user functions.

4.14.4 Performance

Serial execution times grew as $\mathcal{O}(N)$. Parallel execution times grew nearly as fast, but the average parallelism grew as $\mathcal{O}(\lg N)$.

4.15 Problem 15

4.15.1 Problem Description and Algorithm

The problem is to compare two types of interpolation points (equispaced and Chebyshev spaced) for Hermite interpolation using piece-wise polynomials. The interpolant's value v at y is

$$v(y) = \sum_{j=1}^{N} f(x_j)h_{1j}(y) + f'(x_j)h_{2j}(y)$$

where $h_{1j}(x)$ and $h_{2j}(x)$ are suitable basis functions that depend on the N interpolation points x_j.

4.15.2 Inherent Parallelism

At the outermost levels there is modest parallelism across the types of interpolation points. Within that, there is the evaluation to be performed at each of the interpolation points: $\mathcal{O}(k)$ time using $\mathcal{O}(N)$ processors. These points are then summed in $\mathcal{O}(\lg N)$ time using $\mathcal{O}(N)$ processors.

4.15.3 Characteristics of Translation

This problem has not been implemented in FP, due only to lack of time.

4.16 Problem 16

4.16.1 Problem Description and Algorithm

This problem is to solve the matrix equation $Ax = B$ where A is an $N \times N$ Hilbert matrix and B is an $N \times 4$ matrix. The matrix order N takes on the values 4, 9, 16, 25, 36, ... and the B column-vectors are, respectively, the first column of the identity matrix, all 1's, a random perturbation of 1.0 ± 0.01 in all elements, and alternating $+1, -1$.

4.16.2 Inherent Parallelism

The problem statement leaves open the method to be used to solve the problem. I chose an iterative method to compute the matrix inverse, C, which can then be multiplied by B to yield the answer. This method has tremendous parallelism and serial time complexity comparable to direct methods for most matrices.

The serialization is limited only to the test at each iteration, and it is possible to proceed in parallel with the next iteration using the `while` PFO.

4.16.3 Characteristics of Translation

The problem can be restated again as "Compute C, an $N \times N$ inverse of an $N \times N$ matrix, such that the product $(C \times A)$ is a $N \times N$ matrix "close" to the Identity matrix. Then multiply this inverse times the B column-vectors."

Houseman's method [Hou64] is similar in form to Newton's method of finding a root (but operates on matrices).

Each iteration step computes C_{n+1} is $2C_n - C_n \times A \times C_n$ (derived from $B \times (2 \times I - A \times C_n)$ or from $(2I - C_n \times A) \times C_n$) Matrix multiplication is easily expressed in FP: one just shuffles rows and columns about until an `aa` PFO can be used to compute output terms in parallel.

An adequate initial inverse, C_1, is the transpose divided by the product of the row and column maximums.

4.16.4 Performance

Although this technique appears to be robust in terms of many near-singular matrices, it converges painfully slowly with the Hilbert matrix. For this reason, tests have been limited to other, less difficult matrixes.

The matrix size is $\lg(N) \times \lg(N)$, where N is given in these tables. The serial execution times grew as $\mathcal{O}(N^3)$ as would be expected for a naive matrix inversion. Parallelism grew as $\mathcal{O}(N)$, so that parallel execution times grew approximately $\mathcal{O}(N)$. A new implementation of `trans` and tree-structured sequences should offer significant improvement.

4.17 Problem 17

4.17.1 Problem Description and Algorithm

The last Purdue technical report [RJ90] added a 17th problem – adaptive quadrature. The report did not formally specify what was to be done, rather it referred to textbooks that discussed various implementations of AQ [Ric75],[Ric76].

4.17.2 Inherent Parallelism

Given truly independent parallelism – not that of many SIMD systems – AQ offers large, but irregular parallelism.

One does not know *a priori* the degree of parallelism. The degree of parallelism is instead derived from the function being evaluated. Parallelism grows exponentially and the interval is divided into finer and finer pieces; it then decays exponentially as the sub-interval integrals are summed.

There are two controlling factors: the function being evaluated and the error term demanded of the solution.

4.17.3 Characteristics of Translation

A simple trapezoidal rule is used to evaluate the integral for an interval and for two half intervals. If the difference is small enough, the sum of the half intervals is used, if not, the sum of the results of a recursive evaluation is used.

Programming this was puzzling until a data-flow graph was laid out. It was then possible to recognize the several phases in each level. These phases were described in individual functions that passed some parameters through unchanged while changing or creating new parameters for the function in the next phase. Because of the recursive doubling that can occur, it may not qualify as a systolic algorithm, but it has that feel to it.

4.17.4 Performance

Up until this problem, the idea of "problem set size" has been a simple concept. One just counts the elements to be summed, sorted, averaged, etc. Adaptive quadrature specifically avoids an external specification of how many intervals are to be used. As a consequence one has to shift the problem in subtle ways to evoke greater parallelism.

In this test, parallelism was adjusted by controlling the error term. Reasonable results were found by setting the error term to the reciprocal of what we have heretofore called the "problem set size", N.

One sees the parallelism initially grows with N, but even while the maximum parallelism keeps growing, the average seems to be approaching an asymptote of about 100. I suspect that the exponential growth (due to the recursive doubling) keeps the maximum parallelism high, while the average parallelism is dominated by the function itself.

5 Conclusion

Table 1 summarizes the theoretical and actual parallel performance found in the problems.

This implementation of FP has some fundamental limitations in exploiting the full parallelism available in the language. These limitations generally center on the linked list used to represent sequences and the functions and PFOs that operate upon them. In spite of this, many of these programs still show surprising, and gratifying to the author, parallel performance. In five cases

Table 1: Complexity and performance of problems

	Theoretical		Actual	
Prob	Serial	Parallel	Serial	Parallel
1	$\mathcal{O}(N)$	$\mathcal{O}(\lg N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
2	$\mathcal{O}(N^2)$	$\mathcal{O}(\lg N)$	$\mathcal{O}(N^2)$	$\mathcal{O}(\lg^2 N) < \mathcal{O}(N)$
3	$\mathcal{O}(N^2)$	$\mathcal{O}(\lg N)$	$\mathcal{O}(N^2 \lg N)$	$\mathcal{O}(\lg^2 N)$
4	$\mathcal{O}(N)$	$\mathcal{O}(\lg N)$	$\mathcal{O}(N)$	
5	$\mathcal{O}(N^2)$	$\mathcal{O}(\lg N)$	$\mathcal{O}(N^2)$	$\gg \mathcal{O}(\lg^2 N)$
6	not calculated		not implemented in FP	
7	$\mathcal{O}(N^2)$	$\mathcal{O}(\lg N)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$
8	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$
9	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$
10	not calculated		not implemented in FP	
11	$\mathcal{O}(N)$	$\mathcal{O}(\lg N)$	$> \mathcal{O}(N)$	$< \mathcal{O}(N)$
12	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
13	$\mathcal{O}(N)$	$\mathcal{O}(\lg N)$	$> \mathcal{O}(N)$	$< \mathcal{O}(N)$
14	$\mathcal{O}(N)$	$\mathcal{O}(\lg N)$	$\mathcal{O}(N)$	$< \mathcal{O}(N)$
15	not calculated		not implemented in FP	
16			$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
17	Inappropriate measure	Not known *a priori*	not reported	

(problems 2, 7, 9, 16, and 17) the speedups were greater than 50 for problem set sizes of only 32. In at least one case (problem 7) the *speedup* grew faster than $\mathcal{O}(N)$!

It is not surprising to note that while a PFO such as / is the most efficient in a single processor /single thread environment, the \/ variation is better designed to identify and exploit greater parallelism. This suggests that an advanced implementation would dynamically choose the appropriate evaluation strategy for a generalized insert PFO based on system load.

References

[AHO90] K. Aziz, M. Haines, and R. R. Oldehoeft. Purdue Parallel Benchmarks in SISAL (Revised). Technical Report CS-90-101, Department of Computer Science, Colorado State University, 1990.

[Amd67] Gene Amdahl. The validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the 1967 AFIPS National Computer Conference*, pages 483–485. AFIPS, 1967.

[Ben92] Randolph Bentson. The implementation of an FP system with parallel execution. Technical Report CS-92-108, Department of Computer Science, Colorado State University, 1992.

[HHRO88] T. Hanson, S. Harikrishnan, T. Richert, and R. Oldehoeft. The Purdue Parallel Benchmarks in SISAL. Technical Report CS-88-114, Department of Computer Science, Colorado State University, 1988.

[Hou64] A. S. Householder. *The Theory of Matrices in Numerical Analysis.* Blaisdell Publishing Company, New York, 1964.

[MR87] H. S. McFaddin and J. R. Rice. Parallel and Vector Problems on the FLEX/32. Technical Report CSD-TR-661, Department of Computer Science, Purdue University, 1987.

[Ric75] J. R. Rice. A metalgorithm for adaptive quadrature. *Journal of the Association for Computing Machinery*, 22, 1975.

[Ric76] J. R. Rice. Parallel algorithms for adaptive quadrature III – program correctness. *ACM Transactions on Mathematical Software*, 2, 1976.

[Ric80] John R. Rice. Problems to Test Parallel and Vector Languages. Technical Report CSD-TR 516, Department of Computer Science, Purdue University, 1980.

[RJ90] John R. Rice and Jin Jing. Problems to Test Parallel and Vector Languages-II. Technical Report CSD-TR-1016, Department of Computer Science, Purdue University, 1990.

Appendix A Program Performance

In these tables, "N" is the problem set size, "P" is the number of processors, "T" is the number of threads per processor, "max" is the maximum number of concurrently executing tasks, "avg" is the average number of concurrently executing tasks, and "t-o-d" is the number of simulated clock ticks to execute the program.

For the sake of brevity, the number of threads per processor were increased at the same rate as the number of processors (P = T) up to a maximum number of threads per processor of 128.

.1 Execution timing of Problem 1

N	P	T	max	avg	t-o-d
1	1	1	1	0.99	1221
1	2	2	4	2.73	523
2	1	1	1	0.99	1696
2	2	2	4	2.38	777
2	4	4	10	3.59	500
4	1	1	1	1.00	2615
4	2	2	4	2.45	1171
4	4	4	10	3.68	765
4	8	8	10	3.66	765
8	1	1	1	1.00	4525
8	2	2	4	2.48	1967
8	4	4	14	3.82	1238
8	8	8	18	3.88	1224
8	16	16	18	3.88	1224
16	1	1	1	1.00	8465
16	2	2	4	2.46	3692
16	4	4	16	4.21	2062
16	8	8	26	4.25	2062
16	16	16	26	4.25	2062
16	32	32	26	4.25	2062
32	1	1	1	1.00	16561
32	2	2	4	2.49	7298
32	4	4	15	4.30	3946
32	8	8	26	4.47	3829
32	16	16	26	4.47	3829
32	32	32	26	4.47	3829
32	64	64	26	4.47	3829
64	1	1	1	1.00	33161
64	2	2	4	2.55	14194
64	4	4	15	4.48	7610
64	8	8	26	4.62	7427
64	16	16	26	4.70	7276
64	32	32	26	4.70	7276
64	64	64	26	4.70	7276
64	128	128	26	4.70	7276
128	1	1	1	1.00	67153
128	2	2	4	2.59	27957
128	4	4	16	4.61	14960
128	8	8	30	4.83	14322
128	16	16	46	4.93	14121
128	32	32	48	4.95	14125
128	64	64	48	4.95	14125
128	128	128	48	4.95	14125
128	256	128	48	4.95	14125
256	1	1	1	1.00	136697
256	2	2	4	2.58	58344
256	4	4	16	4.78	29348
256	8	8	44	4.98	28305
256	16	16	70	5.12	27801
256	32	32	85	5.30	27863
256	64	64	85	5.30	27863
256	128	128	85	5.30	27863
256	256	128	85	5.30	27863
256	512	128	85	5.30	27863

In the following, the "OneTick" option was used to give the distribution, appending, insertion, and application functions and PFOs a simulated duration of only one tick each. This allows us to examine the effects of more efficient implementations of these functions and PFOs.

N	P	T	max	avg	t-o-d
1	1	1	1	0.99	1214
2	1	1	1	0.99	1579

4	1	1	1	1.00	2274
8	1	1	1	1.00	3736
16	1	1	1	1.00	6780
32	1	1	1	1.00	13084
64	1	1	1	1.00	26100
128	1	1	1	1.00	52924
256	1	1	1	1.00	108132
512	1	1	1	1.00	221644
1	2	2	4	2.75	516
2	4	4	10	4.18	406
4	8	8	13	4.38	561
8	16	16	18	4.93	800
16	32	32	22	5.75	1232
32	64	64	22	6.43	2119
64	128	128	23	7.13	3806
128	256	256	48	7.81	7135
256	512	512	85	8.62	13833
512	1024	1024	128	9.39	27062

.2 Execution timing of Problem 2

N	P	T	max	avg	t-o-d
1	1	1	1	0.99	906
1	2	2	4	2.02	536
2	1	1	1	1.00	2294
2	2	2	4	2.54	1054
2	4	4	11	3.25	701
4	1	1	1	1.00	8101
4	2	2	4	3.02	2875
4	4	4	16	6.10	1337
4	8	8	37	6.24	1251
8	1	1	1	1.00	30983
8	2	2	4	3.17	10616
8	4	4	16	9.77	3206
8	8	8	63	13.86	2231
8	16	16	109	13.82	2168
16	1	1	1	1.00	122875
16	2	2	4	3.26	40426
16	4	4	16	12.25	10194
16	8	8	64	26.46	4639
16	16	16	253	33.15	3936
16	32	32	350	33.25	3831
32	1	1	1	1.00	495587
32	2	2	4	3.35	161063
32	4	4	16	13.28	37783
32	8	8	64	40.28	12325
32	16	16	256	70.32	7982
32	32	32	919	91.56	7633
32	64	64	854	73.17	8463
64	1	1	1	1.00	2014771
64	2	2	4	3.36	648523
64	4	4	16	13.97	146534
64	8	8	64	51.52	39191
64	16	16	256	123.31	19020
64	32	32	1024	231.50	16379
64	64	64	3048	258.90	14993
64	128	128	2032	175.97	17240
128	1	1	1	1.00	8221139
128	2	2	4	3.40	2601872
128	4	4	16	14.15	590054
128	8	8	64	55.75	147833
128	32	32	1024	440.36	38075
128	64	64	4094	651.67	34773
128	128	128	10002	659.64	31875
128	256	128	5641	402.62	33987

In the following, the "OneTick" option was used to give `distl` and `distr` a simulated duration of only one tick each. This allows us to examine the effects of more efficient implementations of these functions.

N	P	T	max	avg	t-o-d
1	1	1	1	0.99	786
1	1	1	1	0.99	786
2	1	1	1	0.99	1949
2	2	2	4	2.70	796
4	1	1	1	1.00	6976
4	4	4	16	7.65	931
8	1	1	1	1.00	26978
8	8	8	63	20.07	1341
16	1	1	1	1.00	107830

16	16	16	253	53.29	2166
32	1	1	1	1.00	437342
32	32	32	919	156.13	4103
64	1	1	1	1.00	1785646
64	64	64	3048	459.85	7943
128	1	1	1	1.00	7312334
128	128	128	10002	1131.14	17785

.3 Execution timing of Problem 3

N	P	T	max	avg	t-o-d
2	1	1	1	0.92	130
2	2	2	3	1.89	71
2	4	4	3	1.89	71
4	1	1	1	0.99	1137
4	2	2	3	2.24	705
4	4	4	7	3.50	396
4	8	8	8	3.84	345
8	1	1	1	1.00	5827
8	2	2	4	2.64	2475
8	4	4	12	6.37	978
8	8	8	18	8.02	764
8	16	16	24	8.69	710
16	1	1	1	1.00	26967
16	2	2	4	3.04	9634
16	4	4	16	10.29	2707
16	8	8	41	16.29	1680
16	16	16	62	18.00	1632
16	32	32	61	23.47	1559
32	1	1	1	1.00	120703
32	2	2	4	3.12	41590
32	4	4	16	12.50	9850
32	8	8	63	31.23	3880
32	16	16	128	43.49	3015
32	32	32	140	56.13	2962
32	64	64	141	56.86	2962
64	1	1	1	1.00	532047
64	2	2	4	3.24	173386
64	4	4	16	13.73	39347
64	8	8	64	42.63	12524
64	16	16	195	75.46	7751
64	32	32	221	97.79	6959
64	64	64	217	109.07	6759
64	128	128	216	109.40	6759
128	1	1	1	1.00	2323183
128	4	4	16	14.23	165321
128	8	8	64	52.19	44713
128	16	16	228	109.57	23450
128	32	32	338	142.44	21474
128	64	64	356	155.77	20846
128	128	128	362	168.46	21009
128	256	128	362	168.46	21009

.4 Execution timing of Problem 4

N	P	T	max	avg	t-o-d
2	1	1	1	1.00	596
2	2	2	4	2.18	273
2	4	4	7	4.08	146
4	1	1	1	1.00	1295
4	2	2	4	2.34	554
4	4	4	10	4.01	323
4	8	8	13	4.68	277
8	1	1	1	1.00	2767
8	2	2	4	2.71	1023
8	4	4	12	5.41	511
8	8	8	22	5.95	465
8	16	16	23	5.95	465
16	1	1	1	1.00	5807
16	2	2	4	2.87	2021
16	4	4	16	6.74	863
16	8	8	26	7.02	827
16	16	16	24	7.02	827
16	32	32	24	7.02	827
32	1	1	1	1.00	12079
32	2	2	4	2.85	4237
32	4	4	16	7.29	1657
32	8	8	26	8.11	1490
32	16	16	24	8.11	1490
32	32	32	24	8.11	1490
32	64	64	24	8.11	1490
64	1	1	1	1.00	25007

64	2	2	4	2.97	8420
64	4	4	16	8.10	3087
64	8	8	26	8.81	2837
64	16	16	28	9.09	2760
64	32	32	27	9.09	2760
64	64	64	27	9.09	2760
64	128	128	27	9.09	2760
128	1	1	1	1.00	51631
128	2	2	4	2.87	18007
128	4	4	16	8.78	5880
128	8	8	30	9.68	5336
128	16	16	35	10.00	5236
128	32	32	48	10.06	5306
128	64	64	48	10.06	5306
128	128	128	48	10.06	5306
128	256	128	48	10.06	5306
256	1	1	1	1.00	106415
256	2	2	4	3.11	34170
256	4	4	16	9.43	11280
256	8	8	38	10.25	10387
256	16	16	53	10.72	10147
256	32	32	60	11.33	10482
256	64	64	71	11.78	10383
256	128	128	71	11.78	10383
256	256	128	71	11.78	10383
256	512	128	71	11.78	10383
512	1	1	1	1.00	219055
512	2	2	4	3.00	73016
512	4	4	16	9.75	22479
512	8	8	53	10.90	20089
512	16	16	63	11.31	19908
512	32	32	109	12.43	19989
512	64	64	96	12.90	19918
512	128	128	96	12.90	19918
512	256	128	96	12.90	19918
512	512	128	96	12.90	19918
512	1024	128	96	12.90	19918

.5 Execution timing of Problem 5

N	P	T	max	avg	t-o-d
1	1	1	1	1.00	2330
1	2	2	4	2.85	889
2	1	1	1	1.00	5782
2	2	2	4	3.13	2095
2	4	4	16	7.72	765
4	1	1	1	1.00	22407
4	2	2	4	3.19	7719
4	4	4	16	11.09	2031
4	8	8	63	17.83	1252
8	1	1	1	1.00	89047
8	2	2	4	3.28	29945
8	4	4	16	13.54	6716
8	8	8	64	34.26	2549
8	16	16	226	41.07	2258
16	1	1	1	1.00	369589
16	2	2	4	3.31	124323
16	4	4	16	14.00	26934
16	8	8	64	49.93	7279
16	16	16	256	93.87	4286
16	32	32	771	110.38	4508
32	1	1	1	1.00	1562859
32	2	2	4	3.34	525594
32	4	4	16	14.05	113569
32	8	8	64	56.52	27747
32	16	16	256	158.22	11180
32	32	32	927	195.71	10747
32	64	64	1256	226.14	9451
64	1	1	1	1.00	6826755
64	2	2	4	3.31	2359995
64	4	4	16	14.24	490094
64	8	8	64	58.48	117172
64	16	16	256	210.19	38036
64	32	32	1007	321.53	30341
64	64	64	1018	298.89	30647
64	128	128	1688	353.48	28096

In the following, the “OneTick” option was used to give `distl` and `distr` a simulated duration of only one tick each. This allows us to examine the effects of more efficient implementations of these functions.

N	P	T	max	avg	t-o-d
2	1	1	1	1.00	5782
2	4	4	16	7.72	765
4	1	1	1	1.00	22407
4	8	8	63	17.83	1252
8	1	1	1	1.00	89047
8	16	16	226	41.07	2258
16	1	1	1	1.00	369589
16	32	32	771	110.38	4508
32	1	1	1	1.00	1562859
32	64	64	1256	226.14	9451
64	1	1	1	1.00	6826755
64	128	128	1688	353.48	28096

.6 Execution timing of Problem 6

No tests were performed on this problem.

.7 Execution timing of Problem 7

N	P	T	max	avg	t-o-d
1	1	1	1	1.00	10326
1	2	2	4	3.11	3474
1	4	4	16	8.65	1184
1	8	8	41	10.50	947
2	1	1	1	1.00	27431
2	2	2	4	3.35	8894
2	4	4	16	11.57	2385
2	8	8	64	21.05	1272
2	16	16	143	21.82	1177
2	32	32	143	21.80	1177
4	1	1	1	1.00	97956
4	2	2	4	3.33	32078
4	4	4	16	13.27	7495
4	8	8	64	36.03	2716
4	16	16	256	51.88	1958
4	32	32	419	48.86	1947
4	64	64	443	49.42	1857
8	1	1	1	1.00	378686
8	2	2	4	3.37	123574
8	4	4	16	14.02	27618
8	8	8	64	48.46	7883
8	16	16	256	115.39	3851
8	32	32	710	143.44	3398
8	64	64	830	157.01	3274
8	128	128	964	165.67	3268
16	1	1	1	1.00	1509906
16	2	2	4	3.38	493593
16	4	4	16	14.27	108422
16	8	8	64	54.96	27785
16	16	16	256	160.25	11344
16	32	32	1015	391.99	7822
16	64	64	2959	275.51	8909
16	128	128	2976	340.95	6670
16	256	128	2995	345.36	6670
16	512	128	2995	345.36	6670
32	1	1	1	1.00	6096506
32	2	2	4	3.37	1993742
32	4	4	16	14.37	435623
32	8	8	64	57.99	106428
32	16	16	256	210.02	35520
32	32	32	1019	688.15	21025
32	64	64	3797	662.77	17792
32	128	128	4399	492.19	17791
32	256	256	4220	722.37	12334
32	512	256	4220	722.37	12334

.8 Execution timing of Problem 8

N	P	T	max	avg	t-o-d
8	1	1	1	1.00	15430
8	2	2	4	2.75	6507
8	4	4	11	3.55	4542
8	8	8	11	3.55	4542
8	16	16	11	3.55	4542
16	1	1	1	1.00	32486
16	2	2	4	3.00	12348
16	4	4	14	4.64	7326
16	8	8	20	4.61	7326
16	16	16	21	4.60	7326
16	32	32	21	4.60	7326

32	1	1	1	1.00	79270
32	2	2	4	3.16	29081
32	4	4	15	6.39	12894
32	8	8	23	6.40	12894
32	16	16	38	6.35	12894
32	32	32	42	6.34	12894
32	64	64	42	6.34	12894
64	1	1	1	1.00	223526
64	2	2	4	3.19	86797
64	4	4	16	8.94	25748
64	8	8	30	9.57	24030
64	16	16	44	9.60	24030
64	32	32	74	9.81	24030
64	64	64	80	9.81	24030
64	128	128	80	9.81	24030
128	1	1	1	1.00	714790
128	2	2	4	3.07	309451
128	4	4	16	11.85	62747
128	8	8	44	15.73	46302
128	16	16	56	15.75	46302
128	32	32	88	17.03	46302
128	64	64	147	17.87	46302
128	128	128	159	17.89	46302
128	256	128	159	17.89	46302
256	1	1	1	1.00	2508326
256	2	2	4	3.03	1135282
256	4	4	16	12.53	211175
256	8	8	64	26.64	95373
256	16	16	125	27.95	90846
256	32	32	113	31.71	90846
256	64	64	176	36.71	90846
256	128	128	297	39.86	90846
256	256	128	297	39.86	90846
256	512	128	297	39.86	90846
512	1	1	1	1.00	9339430
512	2	2	4	2.84	4841957
512	4	4	16	12.52	794663
512	8	8	64	44.57	211583
512	16	16	152	52.25	179934
512	32	32	185	61.13	179934
512	64	64	245	75.88	179934
512	128	128	363	95.11	179934
512	256	128	363	95.11	179934
512	512	128	363	95.11	179934
512	1024	128	363	95.11	179934

In the following, the “OneTick” option was used to give `distl` and `distr` a simulated duration of only one tick each. This allows us to examine the effects of more efficient implementations of these functions.

N	P	T	max	avg	t-o-d
8	1	1	1	1.00	12906
8	16	16	27	4.16	3170
16	1	1	1	1.00	25914
16	32	32	51	6.96	3829
32	1	1	1	1.00	64602
32	64	64	100	10.61	8235
64	1	1	1	1.00	192666
64	128	128	232	16.30	28981
128	1	1	1	1.00	651546
128	256	256	375	25.45	114731
256	1	1	1	1.00	2380314
256	512	512	760	46.93	461877
512	1	1	1	1.00	9081882
512	1024	1024	1527	89.99	1861675

.9 Execution timing of Problem 9

N	P	T	max	avg	t-o-d
4	1	1	1	1.00	55076
4	2	2	4	3.35	18125
4	4	4	16	13.42	4179
4	8	8	59	22.90	2394
8	1	1	1	1.00	213070
8	2	2	4	3.43	66157
8	4	4	16	14.44	14988
8	8	8	64	50.07	4260
8	16	16	197	69.90	3050
16	1	1	1	1.00	864098

16	2	2	4	3.38	277720
16	4	4	16	14.56	60159
16	8	8	64	59.09	14667
16	16	16	255	165.89	6020
16	32	32	654	217.06	4691
32	1	1	1	1.00	3651466
32	2	2	4	3.37	1187869
32	4	4	16	14.50	256043
32	8	8	64	60.51	60552
32	16	16	256	217.43	19808
32	32	32	1019	647.24	12387
32	64	64	1652	614.31	9303
64	1	1	1	1.00	16328666
64	2	2	4	3.34	5486594
64	4	4	16	14.40	1156684
64	8	8	64	60.18	272811
64	16	16	256	237.34	80497
64	32	32	1022	864.25	44822
64	64	64	4077	2401.62	27081
64	128	128	3022	961.63	24074

.10 Execution timing of Problem 10

No tests were performed on this problem.

.11 Execution timing of Problem 11

N	P	T	max	avg	t-o-d
1	1	1	1	1.00	3242
1	2	2	4	2.80	1157
2	1	1	1	1.00	5485
2	2	2	4	3.09	1777
2	4	4	15	8.63	636
4	1	1	1	1.00	10313
4	2	2	4	3.12	3368
4	4	4	16	10.06	1022
4	8	8	41	12.12	858
8	1	1	1	1.00	20553
8	2	2	4	3.15	6613
8	4	4	16	10.44	1986
8	8	8	52	16.72	1240
8	16	16	79	16.74	1240
16	1	1	1	1.00	41921
16	2	2	4	3.16	13429
16	4	4	16	11.50	3683
16	8	8	62	19.81	2130
16	16	16	112	21.44	1981
16	32	32	117	21.39	1981
32	1	1	1	1.00	86145
32	2	2	4	3.12	27890
32	4	4	16	11.91	7324
32	8	8	64	23.54	3688
32	16	16	141	23.98	3688
32	32	32	145	23.82	3771
32	64	64	145	23.69	3794
64	1	1	1	1.00	180953
64	2	2	4	3.03	60171
64	4	4	16	12.77	14305
64	8	8	64	25.11	7248
64	16	16	151	26.74	6958
64	32	32	144	27.55	7129
64	64	64	142	27.76	7188
64	128	128	142	27.76	7188
128	1	1	1	1.00	391857
128	2	2	4	2.96	133204
128	4	4	16	13.47	29428
128	8	8	64	28.15	14005
128	16	16	150	30.06	13416
128	32	32	139	31.64	13489
128	64	64	141	32.17	13511
128	128	128	141	32.17	13511
128	256	128	141	32.17	13511
256	1	1	1	1.00	893645
256	2	2	4	2.86	315049
256	4	4	16	13.45	67049
256	8	8	64	32.91	27315
256	16	16	160	35.34	26045
256	32	32	160	37.66	26063
256	64	64	155	38.96	26063
256	128	128	155	39.33	26085
256	256	128	155	39.33	26085
256	512	128	155	39.33	26085

512	1	1	1	1.00	2204421
512	2	2	4	2.71	819935
512	4	4	16	13.29	167162
512	8	8	64	40.63	54525
512	16	16	197	44.32	51079
512	32	32	216	44.05	61701
512	64	64	221	54.46	63137
512	128	128	285	73.77	60370
512	256	128	285	73.77	60370
512	512	128	285	73.77	60370
512	1024	128	285	73.77	60370
1024	1	1	1	1.00	6030089
1024	2	2	4	2.26	2680529
1024	4	4	16	13.24	458021
1024	8	8	64	46.36	130575
1024	16	16	252	60.97	101178
1024	32	32	259	47.22	165083
1024	64	64	262	51.95	237600
1024	128	128	262	82.67	236282
1024	256	128	262	82.67	236282
1024	512	128	262	82.67	236282
1024	1024	128			

In the following, the "OneTick" option was used to give `distl` and `distr` a simulated duration of only one tick each. This allows us to examine the effects of more efficient implementations of these functions.

N	P	T	max	avg	t-o-d
1	1	1	1	1.00	3047
1	2	2	4	3.06	997
2	1	1	1	1.00	5135
2	4	4	15	9.25	555
4	1	1	1	1.00	9117
4	8	8	40	18.67	493
8	1	1	1	1.00	17377
8	16	16	75	33.88	530
16	1	1	1	1.00	34401
16	32	32	188	66.01	619
32	1	1	1	1.00	69169
32	64	64	306	61.72	1758
64	1	1	1	1.00	143529
64	128	128	537	42.63	6465
128	1	1	1	1.00	310465
128	256	256	996	36.74	24205
256	1	1	1	1.00	718173
256	512	512	1875	45.44	94325
512	1	1	1	1.00	1828501
512	1024	1024	3239	72.02	373974
1024	1	1	1	1.00	5228697

.12 Execution timing of Problem 12

N	P	T	max	avg	t-o-d
1	1	1	1	0.88	786
1	2	2	4	1.92	461
2	1	1	1	0.89	1117
2	2	2	4	1.93	704
2	4	4	6	2.05	425
4	1	1	1	0.91	1959
4	2	2	4	1.94	1352
4	4	4	6	2.43	770
4	8	8	6	2.12	623
8	1	1	1	0.93	4363
8	2	2	4	1.99	3296
8	4	4	6	2.79	1529
8	8	8	8	3.34	1226
8	16	16	10	2.42	1019
16	1	1	1	0.96	12051
16	2	2	4	2.03	9370
16	4	4	6	3.33	4166
16	8	8	8	4.46	2657
16	16	16	16	4.86	2248
16	32	32	18	2.87	1994
32	1	1	1	0.98	38947
32	2	2	4	2.03	33657
32	4	4	7	3.71	12425
32	8	8	8	5.69	6815
32	16	16	16	7.73	5326
32	32	32	32	6.27	5380
32	64	64	34	3.11	5017

64	1	1	1	0.99	138819
64	2	2	4	2.05	125333
64	4	4	11	3.98	43706
64	8	8	12	6.91	22034
64	16	16	16	10.71	14650
64	32	32	32	14.85	14891
64	64	64	64	7.63	15215
64	128	128	66	3.41	14414
128	1	1	1	0.99	522883
128	2	2	4	2.05	481742
128	4	4	16	4.21	156140
128	8	8	20	7.69	74945
128	16	16	24	13.24	45970
128	32	32	32	21.98	46969
128	64	64	64	30.14	48257
128	128	128	128	8.83	48508
128	256	128	128	8.83	48508

In the following, the "OneTick" option was used to give `distl` and `distr` a simulated duration of only one tick each. This allows us to examine the effects of more efficient implementations of these functions.

N	P	T	max	avg	t-o-d
1	1	1	1	1.00	328
1	2	2	4	2.06	159
2	1	1	1	1.00	366
2	4	4	8	3.85	95
4	1	1	1	1.00	460
4	8	8	10	4.55	101
8	1	1	1	1.00	720
8	16	16	17	6.37	113
16	1	1	1	1.00	1528
16	32	32	33	11.00	142
32	1	1	1	1.00	4296
32	64	64	65	17.37	350
64	1	1	1	1.00	14440
64	128	128	129	32.79	1150
128	1	1	1	1.00	53160
128	256	256	257	64.62	4286

.13 Execution timing of Problem 13

N	P	T	max	avg	t-o-d
1	1	1	1	1.00	1152
1	2	2	4	2.39	472
2	1	1	1	1.00	2249
2	2	2	4	2.79	799
2	4	4	15	5.45	408
4	1	1	1	1.00	4614
4	2	2	4	2.94	1569
4	4	4	16	6.41	714
4	8	8	30	7.74	590
8	1	1	1	1.00	9562
8	2	2	4	2.95	3246
8	4	4	16	7.70	1242
8	8	8	55	9.66	977
8	16	16	61	9.82	962
16	1	1	1	1.00	20130
16	2	2	4	2.89	6962
16	4	4	16	8.82	2282
16	8	8	63	12.28	1626
16	16	16	77	12.39	1609
16	32	32	77	12.39	1609
32	1	1	1	1.00	43762
32	2	2	4	2.86	15321
32	4	4	16	10.11	4329
32	8	8	63	14.54	2974
32	16	16	80	14.60	2974
32	32	32	80	14.60	2974
32	64	64	80	14.60	2974
64	1	1	1	1.00	100626
64	2	2	4	2.80	35961
64	4	4	16	11.34	8874
64	8	8	64	17.24	5808
64	16	16	84	17.58	5685
64	32	32	80	17.57	5685
64	64	64	80	17.57	5685
64	128	128	80	17.57	5685
128	1	1	1	1.00	251986
128	2	2	4	2.56	98440

128	4	4	16	11.47	21971
128	8	8	64	22.19	11345
128	16	16	97	22.55	11136
128	32	32	93	22.57	11154
128	64	64	93	22.57	11154
128	128	128	93	22.57	11154
128	256	128	93	22.57	11154
256	1	1	1	1.00	703698
256	2	2	4	2.44	288779
256	4	4	16	12.03	58485
256	8	8	64	31.24	22523
256	16	16	133	32.05	21935
256	32	32	127	32.20	22076
256	64	64	125	32.32	22086
256	128	128	125	32.32	22086
256	256	128	125	32.32	22086
256	512	128	125	32.32	22086
512	1	1	1	1.00	2200018
512	2	2	4	2.29	962686
512	4	4	16	12.13	181432
512	8	8	64	39.63	55515
512	16	16	192	50.68	43433
512	32	32	191	51.26	43303
512	64	64	187	51.31	43444
512	128	128	187	51.31	43444
512	256	128	187	51.31	43444
512	512	128	187	51.31	43444
512	1024	128	187	51.31	43444

In the following, the "OneTick" option was used to give `distl` and `distr` a simulated duration of only one tick each. This allows us to examine the effects of more efficient implementations of these functions.

N	P	T	max	avg	t-o-d
1	2	2	3	2.56	61
4	8	8	5	3.17	208
16	32	32	8	4.72	796
64	128	128	20	10.62	3148
256	512	512	67	34.13	12556

.14 Execution timing of Problem 14

N	P	T	max	avg	t-o-d
8	1	1	1	1.00	45381
8	2	2	4	3.32	14614
8	4	4	16	12.69	3613
8	8	8	59	22.52	2009
16	1	1	1	1.00	91077
16	2	2	4	3.40	28450
16	4	4	16	13.04	7060
16	8	8	64	26.36	3450
16	16	16	164	27.24	3363
32	1	1	1	1.00	183837
32	2	2	4	3.42	57226
32	4	4	16	12.77	14518
32	8	8	64	28.76	6400
32	16	16	216	30.75	6026
32	32	32	248	30.81	6032
64	1	1	1	1.00	371877
64	2	2	4	3.39	116824
64	4	4	16	12.58	29853
64	8	8	64	29.79	12511
64	16	16	252	31.71	12300
64	32	32	260	32.64	11799
64	64	64	269	32.75	11799
128	1	1	1	1.00	752781
128	2	2	4	3.42	233702
128	4	4	16	12.74	59551
128	8	8	64	31.15	24221
128	16	16	255	33.20	24616
128	32	32	264	34.25	23034
128	64	64	268	34.28	23025
128	128	128	267	34.27	23025
256	1	1	1	1.00	1524021
256	2	2	4	3.38	482517
256	4	4	16	12.91	118989
256	8	8	64	31.90	47861
256	16	16	255	34.36	48808
256	32	32	269	35.08	45510
256	64	64	275	35.06	45495

256	128	128	266	34.98	45495
256	256	128	266	34.98	45495
512	1	1	1	1.00	3085149
512	2	2	4	3.37	977309
512	4	4	16	12.96	240086
512	8	8	64	32.60	94842
512	16	16	255	35.19	97505
512	32	32	275	35.90	90274
512	64	64	275	35.74	90391
512	128	128	278	35.68	90391
512	256	128	278	35.68	90391
512	512	128	278	35.68	90391

.15 Execution timing of Problem 15

No tests were performed on this problem.

.16 Execution timing of Problem 16

N	P	T	max	avg	t-o-d
4	1	1	1	1.00	96046
4	2	2	4	3.34	32422
4	4	4	16	13.39	7348
4	8	8	61	26.22	3640
8	1	1	1	1.00	396273
8	2	2	4	3.38	124511
8	4	4	16	14.08	27888
8	8	8	64	42.40	9313
8	16	16	133	46.05	8410
16	1	1	1	1.00	840389
16	2	2	4	3.37	274478
16	4	4	16	14.30	59480
16	8	8	64	50.68	17146
16	16	16	199	68.43	12008
16	32	32	214	70.32	11966
32	1	1	1	1.00	2069506
32	2	2	4	3.37	675396
32	4	4	16	14.38	144493
32	8	8	64	56.35	36219
32	16	16	240	103.97	20908
32	32	32	361	103.25	21199
32	64	64	389	105.40	20970
64	1	1	1	1.00	3240819
64	2	2	4	3.37	1068158
64	4	4	16	14.38	225387
64	8	8	64	58.25	55836
64	16	16	256	139.01	25154
64	32	32	496	149.21	24150
64	64	64	514	146.17	24692
64	128	128	554	145.39	24273
128	1	1	1	1.00	5454477
128	2	2	4	3.37	1837396
128	4	4	16	14.38	380407
128	8	8	64	59.10	94568
128	16	16	255	166.73	38635
128	32	32	643	190.74	33668
128	64	64	640	181.23	35236
128	128	128	639	181.14	34980
128	256	128	647	187.10	32900
256	1	1	1	1.00	8728229
256	2	2	4	3.36	2961945
256	4	4	16	14.39	619983
256	8	8	64	59.66	150517
256	16	16	256	190.32	54740
256	32	32	876	232.13	48402
256	64	64	985	212.64	47646
256	128	128	971	234.64	47567
256	256	128	997	242.42	45383
256	512	128	997	242.42	45383
512	1	1	1	1.00	13338322
512	2	2	4	3.36	4461349
512	4	4	16	14.40	957216
512	8	8	64	59.96	221050
512	16	16	256	199.22	77913
512	32	32	984	299.65	58792
512	64	64	1202	256.37	67228
512	128	128	1298	232.64	67940
512	256	128	1369	278.34	60950
512	512	128	1369	278.34	60950
512	1024	128	1369	278.34	60950

.17 Execution timing of Problem 17

N	P	T	max	avg	t-o-d
1	1	1	1	1.00	26123
1	2	2	4	3.36	8647
2	1	1	1	1.00	26123
2	2	2	4	3.36	8647
2	4	4	16	11.38	1229
4	1	1	1	1.00	53929
4	2	2	4	3.34	17085
4	4	4	16	12.98	4572
4	8	8	59	21.53	1127
8	1	1	1	1.00	53929
8	2	2	4	3.34	17085
8	4	4	16	12.98	4572
8	8	8	59	21.53	1127
8	16	16	85	21.87	1066
16	1	1	1	1.00	81735
16	2	2	4	3.35	27588
16	4	4	16	13.74	5717
16	8	8	60	25.10	1461
16	16	16	85	27.13	1330
16	32	32	94	27.53	1313
32	1	1	1	1.00	109541
32	2	2	4	3.35	38730
32	4	4	16	13.95	8086
32	8	8	64	37.47	1891
32	16	16	143	36.70	1381
32	32	32	147	37.22	1387
32	64	64	162	38.13	1364
64	1	1	1	1.00	192959
64	2	2	4	3.34	69739
64	4	4	16	14.15	14663
64	8	8	64	46.30	3217
64	16	16	195	54.28	1689
64	32	32	225	53.26	1710
64	64	64	232	53.07	1774
64	128	128	232	53.07	1774
128	1	1	1	1.00	192959
128	2	2	4	3.34	69739
128	4	4	16	14.15	14663
128	8	8	64	46.30	3217
128	16	16	195	54.28	1689
128	32	32	225	53.26	1710
128	64	64	232	53.07	1774
128	128	128	232	53.07	1774
128	256	128	232	53.07	1774
256	1	1	1	1.00	359795
256	2	2	4	3.35	131063
256	4	4	16	14.27	27067
256	8	8	64	53.72	6526
256	16	16	250	88.18	2095
256	32	32	340	89.25	2026
256	64	64	314	84.34	2056
256	128	128	324	84.58	2056
256	256	128	324	84.58	2056
256	512	128	324	84.58	2056
512	1	1	1	1.00	415407
512	2	2	4	3.32	153577
512	4	4	16	14.35	31251
512	8	8	64	54.72	7594
512	16	16	254	113.39	2392
512	32	32	507	107.53	2019
512	64	64	476	100.99	2056
512	128	128	484	97.73	2094
512	256	128	484	97.73	2094
512	512	128	484	97.73	2094
512	1024	128	484	97.73	2094

B FP Program Listings

.1 Source Listing of Problem 1

```
# Problem 1
# Given three arguments X1, Xn, and N, compute integral
# of function from X1 to Xn by trapezoidal rule using
# N-1 segments and the two end points.

# It appears that 'h' is computed twice.  I don't think
# it's worth the effort to avoid this.
Def h
  div
  o [ -
      o [2,1]
    , *
      o [3,_1.0]
    ]

Def Tn
  *
  o [ h
    , +
      o [ bur div 2
  o +
  o aa TestFunc
  o [ 1 , 2 ]
, \/+
  o aa ( TestFunc
 # Instead of summing 'h' to get each term (and
 # doing less arithmetic over longer time), we
 # choose to use the quicker form.
o + o [ * o [ 1 , 3 ] , 2 ]
o apndr
 # The "apndr" has the effect
 # of changing the sequence to
 # <<h,iv,x1>,<h,iv,x2>,...<h,iv,xn>>
)
  # At this point the object is the sequence
  # <<<h,iv>,x1>,<<h,iv>,x2>,...<<h,iv>,xn>>
  o distl
  o [ [ h
```

```
      , 1
      ]
    , iota
      o -
      o [3,_1]
    ]
]
    ]

Def TestFunc exp
```

.2 Source Listing of Problem 2

```
# Problem 2
# Compute  *    n   m     (-|i-j|)
#         e = Sum Pi  (1+e        )
#             i=1 j=1

Def estar
    ( \/+
    o (aa prod)
    o distr
    o [iota o 1, 2]
    )

Def prod
    ( \/*
    o (aa fun)
    o distl
    o [1, iota o 2]
    )

Def fun
    ( (bu + 1)
    o exp
    o neg
    o abs
    o -
    o [1,2]
    )
```

.3 Source Listing of Problem 3

```
# Problem 3
# Compute  *    n   m
#         e = Sum Pi (a   )
#             i=1 j=1  i,j

Def s
      \/+ o (aa \/*)
```

.4 Source Listing of Problem 4

```
# Problem 4
# Compute       n 1
#         R = Sum --
#             i=1 xi  (xi!=0)

Def r
    \/+
    o (aa inv)

Def inv
    (bu = 0)
    -> _0
    ; (bu div 1.0)
```

.5 Source Listing of Problem 5

```
# Problem 5
# The data structure is a sequence of test results,
# each containing a sequence of student scores.
#        Student 1  Student 2 ... Student m
# Test 1 << a, b, c>,
# Test 2 ,< d, e, f>,
# ...
# Test n ,< g, h, i>>
#

# The top score for each student.
Def top
      (aa \/max) o trans
```

```
# The total number of scores above average.
Def nabove
      \/+ o
      (aa (\/+ o (aa (2->_1;_0) )) )

# Boosts above average scores by 10%.
Def addten
      (aa aa (2->(* o [1,_1.1]);1))

# Finds the lowest score above average.
Def lowestabove
      ( (null->_<>;\/min)
      o ( aa ( (null->_<>;\/min)
     o append
     o (aa (2->[1];_<>) )) )
      )

# Determine if any student is completely
# above average.
Def genius
      ( \/or
      o ( aa ( \/or o aa 2))
      o trans
      )

Def above
      (aa aa ([1,>->_T;_F]))

Def avg
      div o
      (aa \/+) o
      trans o
      (aa [\/+,length])

Def test
      apndl
      o
      [top
      , [nabove,addten,lowestabove,genius]
        o above
        o (aa distr)
        o distr
        o [id,avg]
      ]
```

.6 Source Listing of Problem 6

No program was written for this problem.

.7 Source Listing of Problem 7

```
# Problem 7
# Compute polynomial interpolant values of f(x) at
# five points using Lagrange interpolation formulas....
#
# The formulation immediately below is derived
# directly from the formula. It doesn't, however,
# retain the value of the "f(xi)/(xi-xj)" term.
# The formulation in the real code does, but it
# doesn't yet exploit this in computing for the
# multiple input values.
#
# It's interesting to note that the two formulations
# return slightly different values because of the
# sum-then-div in one and the div-then-sum in the
# other.
#
# Def terms
#     ( aa( \/*
#  o aa( ( = o [1,2]
#        ->( func o 1)
#        ; ( div
#  o [ - o [3,2]
#    , - o [1,2]]
#  )
#        )
#      o apndr
#      )
#  o distr
#  )
#     o distr
#     )
```

```
#
# Def p_of_x_by_N
#   ( \/+
#   o terms
#   o [ ( matrix
#       o xsub
#       o N
#       )
#     , id
#     ]
#   )

Def numerator
    ( aa( ( \/*
  o aa( ( = o [1,2]
-> _1.0
; - o [3,2]
)
      o apndr
      )
  o distr
  )
)
    o distr
    )

Def denominator
    ( aa( ( \/*
  o aa( = o [1,2]
      ->( div
o [ _1.0 , func o 1]
)
      ; - o [1,2]
      )
  )
)
    o 1
    )

Def matrix
    ( (aa distl)
    o distr
    o [id,id]
    )

Def xsub
    ( (aa (bu * 0.2))
    o iota
    )

Def p_of_x_by_N
  ( \/+
  o (aa div)
  o trans
  o [ numerator
    , denominator ]
  o [ ( matrix
      o xsub
      o 2
      )
    , 1
    ]
  )

Def func exp

Def test
    aa (
 p_of_x_by_N
      )
  o
    distr
  o
    [_<1.1, 1.2, 2.1, -1.1, 2.2>,id]
```

.8 Source Listing of Problem 8

```
# Problem 8
# Compute the first M columns of the divided difference table.
#
# We start with the structure
# <index
# ,<xi vector>
# ,<yi vector>
# >
# which we change to the structure
# <index
# ,<xi vector>
# ,<yi vector>
# ,<delta xi>
# >
# Each iteration reduces the index by one,
# 'slips' a new difference column before
# the delta xi, and computes a new, shorter,
# delta xi.
# When the iterations are complete, the
# (now zero) index and the delta xi column
# are removed.
#

Def test
    tl o
    tlr o
    iterate o
    append o
    [ id , [ diff o 2 ] ]

Def iterate
    ( while
( (bu < 0.0) o 1)
( append
  o [ (bur - 1) o 1 # reduce index
    , tlr o tl # body (less index and delta xi)
    , [ diff o 2r ]
    , [ sum o 1r ] # new, shortened, delta xi
    ]
)
    )

Def sum
    (aa +) o trans o [tl,tlr]

Def diff
    (aa -) o trans o [tl,tlr]
```

.9 Source Listing of Problem 9

```
# Problem 9
# Compute a new value Ai,j that is the average of
# its neighbors.

Def test
    (aa( aa( process ))) o
    build_matrix

# Here's where memory use grows nine-fold.  Nine new
# matricies are build--eight of which are trimmed by
# top, bottom, left, right, or diagonals.  The two
# transpositions shuffle this to a single matrix such
# that each element of the original matrix is "replaced"
# by a 3x3 matrix containing the values of the cell and
# its neighbors.  The append operation is used to
# strip out the null entries.  Surprisingly, there is
# no performance improvement to move it into the nested
# apply operation above.
#
Def build_matrix
    (aa( aa( \/append))) o
    (aa trans) o
    trans o
    [ (aa tlrfill) o tlrfill
    , tlrfill
    , (aa tlfill) o tlrfill
    , (aa tlrfill)
    , id
    , (aa tlfill)
    , (aa tlrfill) o tlfill
    , tlfill
    , (aa tlfill) o tlfill
    ]

Def process
      div o [\/+,length]

Def tlfill
      apndr o [tl, nullify o 1]

Def tlrfill
```

```
      apndl o [ nullify o 1, tlr]

Def nullify
      atom->_<>;(aa nullify)
```

.10 Source Listing of Problem 10

No program was written for this problem.

.11 Source Listing of Problem 11

```
# Problem 11
# Limit 0<data<1000, then log(1+d), then Fourier moments K=1,4
#
#   N
# Sum d * cos(pi*i*K/    ) /
# i=1  i     (     /(N+1))/ N
#
# Although it was possible to construct the test data with the
# index and array size bound to each value, it wouldn't have
# been a proper measure of the expressive power of the language.
# Instead the vector is first built and is then processed.

Def fourier_moments
  [ ( div
    o [ ( \/+ o (aa 1) )
      , 3 o 1
      ]
    )
  , ( div
    o [ ( \/+
o (aa ( *
      o [ 1
, ( cos
  o div
  o [ (bu * 3.1415926535) o 2
    , (bu + 1.0) o 3
    ]
  )
]
      )
  )
)
      , 3 o 1
      ]
    )
  , ( div
    o [ ( \/+
o (aa ( *
      o [ 1
, ( cos
  o div
  o [ (bu * 3.1415926535) o (bu * 2.0) o 2
    , (bu + 1.0) o 3
    ]
  )
]
      )
  )
)
      , 3 o 1
      ]
    )
  , ( div
    o [ ( \/+
o (aa ( *
      o [ 1
, ( cos
  o div
  o [ (bu * 3.1415926535) o (bu * 3.0) o 2
    , (bu + 1.0) o 3
    ]
  )
]
      )
  )
)
      , 3 o 1
      ]
    )
  ]

Def shuffle
```

```
  ( trans
  o [ 1 , iota o 2 , 2 ]
  o [ id , length ]
  )

Def range_limit
  (aa ( log
      o (bu + 1.0)
      o ( (bu < 0.0) ->( (bur < 1000.0) ->id ; _1000.0 ) ; _0.0 )
      )
  )

Def test
    fourier_moments o shuffle o range_limit
```

.12 Source Listing of Problem 12

```
# Problem 12
# Build a new matrix ABIG = | A C |
# where A is a nxm matrix,  | R a |
#       R is a nx1 vector,
#       C is a 1xm vector,
#   and a is a scalar.

Def test
    ( (aa append)
    o trans
    o [ apndr o [1,2]
      , apndr o [3,4]
      ]
    )
```

.13 Source Listing of Problem 13

```
# Problem 13
# Given vectors a,b,c,d, compute a new vector such
# that ai = ti + ci if sin(ti)<cos(ci), else
#           ti - di,  where ti = ai**sin(bi)

Def e
    (\/+
    o (aa
( *
o [id,id]
o ( (< o [sin o 1,2])
  ->(+ o [1,3])
  ; (- o [1,4])
  )
o [ pow o [abs o 1,sin o 2]
  , cos o 2
  , 3
  , 4
  ]
)
      )
    o trans
    )
```

.14 Source Listing of Problem 14

```
# Problem 14
# Test four integration methods
#  (trapezoidal, Simpson's, Runga-Kutta(?))
# on three functions
#  (exp, sqrt(abs(x-.2345)), 2.+101.*x*x)
# at ten levels of accuracy.
#  (10, 25, 50, 75, 100, 150, 200, 300, 500, 1000 parts)

# This problem is the first that shows the shortcoming
# of FP vis a vis other functional languages.  Since
# only the "standard" PFOs may operate on functions,
# one cannot define new PFOs to perform integration of
# functions over intervals.  As a consequence, one must
# program each function into each integration method.

# The first function, range to be integrated, and the "correct" answer
Def f1
    exp
Def r1
    _<0.0,1.0>
Def a1
    _1.71828182845
```

```
# The second function, range to be integrated, and the "correct" answer
Def f2
     sqrt
    o abs
    o bur - 0.2345
Def r2
    _<0.0,1.0>
Def a2
    _0.5222099422093

# The third function, range to be integrated, and the "correct" answer
Def f3
     +
    o [ id # OR SHOULD THIS BE (bu + 1.)?
      , (bu div 1.) o (bu + 1.) o (bu * 100.)
      ]
    o *
    o [id,id]
Def r3
    _<-1.0,2.0>
Def a3
    _2.33830

# The two methods--each invoking the three functions

Def simpson
    [ * o
      [ div o [1 o 1, _3.0]
      ,  \/ +
o [ f1 o 2 o 1
  , f1 o 3 o 1
  , \/+
    o aa (*
 o [ ((bu = 0) o (bur mod 2) o 2)->_2;_4
   , f1 o + o [2 o 1,* o [1 o 1,2]]
   ]
 )
    o distl
  ]
      ]
      o # build a 2-tuple: <h,L,U,(N-1),1/(N-1)> and an (N-1)-tuple
[ [ div o [- o [2,1] o r1,id]
```

```
    , 1 o r1
    , 2 o r1
    , id
    , bu div 1.0
    ]
  , iota o bur - 1
  ]
      , * o
        [ div o [1 o 1, _3.0]
        ,  \/ +
  o [ f2 o 2 o 1
    , f2 o 3 o 1
    , \/+
      o aa (*
   o [ ((bu = 0) o (bur mod 2) o 2)->_2;_4
     , f2 o + o [2 o 1,* o [1 o 1,2]]
     ]
   )
      o distl
    ]
        ]
        o # build a 2-tuple: <h,L,U,(N-1),1/(N-1)> and an (N-1)-tuple
  [ [ div o [- o [2,1] o r2,id]
    , 1 o r2
    , 2 o r2
    , id
    , bu div 1.0
    ]
  , iota o bur - 1
  ]
      , * o
        [ div o [1 o 1, _3.0]
        ,  \/ +
  o [ f3 o 2 o 1
    , f3 o 3 o 1
    , \/+
      o aa (*
   o [ ((bu = 0) o (bur mod 2) o 2)->_2;_4
     , f3 o + o [2 o 1,* o [1 o 1,2]]
     ]
   )
      o distl
  ]
      ]
      o # build a 2-tuple: <h,L,U,(N-1),1/(N-1)> and an (N-1)-tuple
[ [ div o [- o [2,1] o r3,id]
  , 1 o r3
  , 2 o r3
  , id
  , bu div 1.0
  ]
, iota o bur - 1
]
    ]
    o (( bu = 0 o bur div 2 ) -> id ; bur - 1 ) # odd number of points

Def trapezoid
    [ * o     # First function's integration
      [ + o
[
  \/+ o
  aa (f1 o + o [1,* o [- o [2,1],4]] o apndr ) o
  distl o [ apndr o [r1,1] , 2 ]
,
  bur div 2.0 o
  + o
  aa f1 o r1
]
      ,
div o [- o [2,1] o r1,1]  # compute h
      ]
    , * o     # Second function's integration
      [ + o
[
  \/+ o
  aa (f2 o + o [1,* o [- o [2,1],4]] o apndr ) o
  distl o [ apndr o [r2,1] , 2 ]
,
  bur div 2.0 o
  + o
  aa f2 o r2
]
      ,
div o [- o [2,1] o r2,1]  # compute h
```

```
      ]
    , * o     # Third function's integration
      [ + o
[
  \/+ o
  aa (f3 o + o [1,* o [- o [2,1],4]] o apndr ) o
  distl o [ apndr o [r3,1] , 2 ]
,
  bur div 2.0 o
  + o
  aa f3 o r3
]
      , div o [- o [2,1] o r3,1]  # compute h
      ]
    ]

    # build a 2-tuple: N and an (N-1)-tuple of points
    # at which the function is to be evaluated
Def build_data_set
    [ id
    , aa *
      o distr
      o [iota o bur - 1, bu div 1.0]
    ]

Def test
    [ [a1,a2,a3] # the "correct" answers
    , simpson
    , trapezoid
    ]
    o build_data_set
```

.15 Source Listing of Problem 15

No program was written for this problem.

.16 Source Listing of Problem 16

```
# Problem 16
# Solve Ax=B where A is NxN matrix and
# B is a Nx4 matrix with column 1 is <1,0...0>,
# column 2 is <1,1...1>, column 3 is 0.01 random,
# and column r is ....
#
# Uses a interative matrix inversion routine below:

# Compute an MxN inverse of an NxM matrix, N<M, such
# that the product (matrix X inverse) is a NxN matrix
# "close" to the Identity matrix.

# Group the matrix with an approximate inverse,
# an appropriately sized identity matrix and the
# iteration control parameters;
# interate until done and extract the inverse
#
Def invert [1,2,4]
 o iter
 o [ 1 , initial o 1 , Imax o trans o 1 , 2 ]

# Compute Newton's approximation to the inverse
#
Def iter while notyet once

## Approximation is 2*B - BxAxB
## (derived from B x (2*I-AxB)
## or from (2*I-BxA) x B )
# The first term is the original matrix,
# the second term is the possible inverse,
# the third term is an appropriately sized identity matrix,
# the forth term contains the interation termination parms.
#
Def once [ 1
 , (termdiff
   o [(aa aa (bu * 2)) o 2 ,cross o [2, cross] ]
   )
 , 3
 , [ (bur - 1) o 1 , 2 ] o 4
 ]

# Compare the identity matrix with the product
# of the matrix and a possible inverse.  Quit
# when the error is small enough or there have
# been "enough" iterations.
#
```

```
Def notyet # while (0<count)and(epsilon<errorterm)
    ( and
    o [ ( < o [ _0 , 1 o 4 ] )
      , ( <= o [ 2 o 4 , errmag o [ 3, cross] ] )
      ]
    )

Def cross (aa aa dot) o (aa distl)
o distr o [1,trans o 2]

Def dot (\/+) o (aa *) o trans

# Sum the termwise absolute error
Def errmag (\/+) o (aa abs)
o append o termdiff

Def termdiff (aa aa -)
o (aa trans) o trans

Def Imax (aa aa (eq -> _1 ; _0))
o buildmax o sizemax

Def buildmax (aa distl) o distr
o [id,id] o iota

Def sizemax (\/+) o (aa _1) o 1

# An adequate initial inverse is the transpose
# divided by the product of the row and column maximums
Def initial (aa aa *) o (aa distl)
o distl o [tau,trans]

Def tau (bu div 1.0)
o * o [rowmax,colmax]

Def colmax rowmax o trans

Def rowmax (\/max) o (aa \/+)

# Test frame to return the "answer".
Def test
    ( cross o [2 o 1,2]
    o [ invert o 1 , 2]
    )
```

.17 Source Listing of Problem 17

```
# Problem 17
# Adaptive quadrature

# the inputs are: epsilon, lower bound, interval,

Def Prog
    Aq
    o Prep

Def Prep [1 # 1 epsilon
         ,2 # 2 low
         ,div o [3,_2.0] # 3 h/2
         ,+ o [2,div o [3,_2.0]] # 4 mid
         ,div o [3,_4.0] # 5 h/4
         ,Trap o [2,3,div o [3,_2.0]]] # 6 approx

Def Aq
    ((< o [abs o - o [6,+ o [7,8]],1]) -> + o [7,8] ; A2)
    o A1

Def A2 + o [Aq o [div o [1,_2.0],2,5,+o[2,5],div o[5,_2.0],7]
           ,Aq o [div o [1,_2.0],4,5,+o[4,5],div o[5,_2.0],8]]

Def A1 [1,2,3,4,5,6,Trap o [2,3,5],Trap o [4,3,5]]

Def Trap
    +
    o [* o [Fun o 1,3]
      ,* o [Fun o + o [1,2],3]]

# This is the "probability function" that
# produces the bell-shaped curve
# Def Fun (bur * 0.3989422804) o exp o (bur div -2.0) o * o [id,id]

Def Fun sin
```

.18 Source Listing of utilities

```
Def tdiff
    ( [_5,id,(aa (exp))] #(bu pow 2.0)
    o (aa(bur div 10.0)) #(bur - 1.0)
    o iota
    )

# This function generates an N x M matrix with values +-1.0
Def tmax
    aa aa (* o [sin o 1, cos o 2]) o
    aa distl o distr o [iota o 1,iota o 2]

# This function generates an N x M matrix with values +1.0 - 0.0
Def tscore
      aa aa (bur div 2.0 o bu + 1.0 o * o [sin o 1, cos o 2])
    o aa distl
    o distr
    o [iota o 1,iota o 2]

# This function generates an N element vector with values +-1.0
Def tvec
    aa sin o iota
```