Experience with a Large Scientific Application in a Functional Language

Rex L. Page
Amoco Production Company
501 WestLake Park Blvd
Houston, TX 77079
rlpage@amoco.com

Brian D. Moe
Department of Computer Science
University of Wisconsin – Milwaukee
P.O.Box 784
Milwaukee, WI 53201
bri@cs.uwm.edu

Abstract

Simulating fluid flow in oil reservoirs requires a great deal of numeric computation. Even a modest model may consume many hours on the fastest computing equipment. The software expressing reservoir models is large and complex, typically tens of thousands to hundreds of thousands of lines of Fortran code. For these reasons, reservoir models provide an interesting environment in which to test ideas in programming languages and high performance computation.

This paper reports on the first part of an experiment in these domains, namely the development in a purely functional language of a reservoir simulation system comparable in function to models used in the oil industry. The product of this effort is a 200-page LaTeXdocument in which a Miranda program is embedded. The program was derived from the scientific equations underlying reservoir models and from the engineering documentation associated with existing software written in Fortran. The Fortran code itself was not consulted except to build test datasets. The Miranda/LaTeXdocument, as it stands, is suitable for reservoir engineers to use as a readable specification of a reservoir simulation system.

The document can serve as a basis for studies of the properties of large functional programs expressing scientific computations. It can also serve as a basis for the originally planned second part of the experiment: mapping the computation onto a high performance, parallel computer. Software researchers may find the experiences reported in this paper a useful data point in studies of the effects of programming languages on scientific application development.

1 Project history

In the mid-eighties, when parallel computing systems with dozens to hundreds of nodes came onto the commercial market, Amoco began an experiment in the use of such systems. The experiment attempted to compare conventional software development methods with equation-based programming.

The researchers chose three oil exploration and production applications of moderate size (three to five thousand lines of Fortran each): seismic migration, hydrocarbon maturation, and an iterative method for solving sparse systems of linear equations. One team used a conventional approach, looking at the algorithms as expressed in Fortran code and converting this code to operate on an early hypercube system with 64 processing elements. Meanwhile, another researcher looked at the equations underlying the computations and derived algorithms in the fashion, more or less, of a compiler for an equation-based language such as Haskell [6].

The equations underlying two of the applications, hydrocarbon maturation and linear systems, suggested algorithms essentially the same as had been developed by starting from Fortran codes. The equations underlying the seismic migration, however, led to a different view of parallel computation strategies for that application. A program developed from this new view performed about seventy percent better on the hypercube system than the conventionally developed code [13].

These results suggested a two-step approach to software development for parallel computing systems: (1) express the computation as a collection of equations; (2) translate the equations into an equivalent procedural form suitable for the target parallel computing system. The equational presentation of the computation can serve as both documentation and (if a processor exists for the notation, as it would if the notation were Haskell or Miranda[1] [11], for example) as a platform for experimentation with features that might enhance the application. The procedural code, on the other hand, would be the code used in production runs.

An important aspect of this approach (Figure 1) is that it does not require the existence of an efficient processor for the equational notation on the target parallel system. If an efficient processor for the equational presentation exists on the target parallel system, then the second step of the approach, in which people translate the equations into procedural code, is of course unnecessary. However, no processor exists at present to carry out high performance computations on distributed-memory, parallel computers from software specifications formulated as collections of equations. In fact, no such processor was even on the horizon in the mid-eighties, when the Amoco experiment began. So, it continues to be important that the approach can be followed in spite of performance deficiencies in functional language processors.

Another important aspect of the approach is that scientists whose expertise and interests focus on the application area (geoscience, for example) can take the lead in designing

[1]Miranda is a trademark of Research Software Ltd.

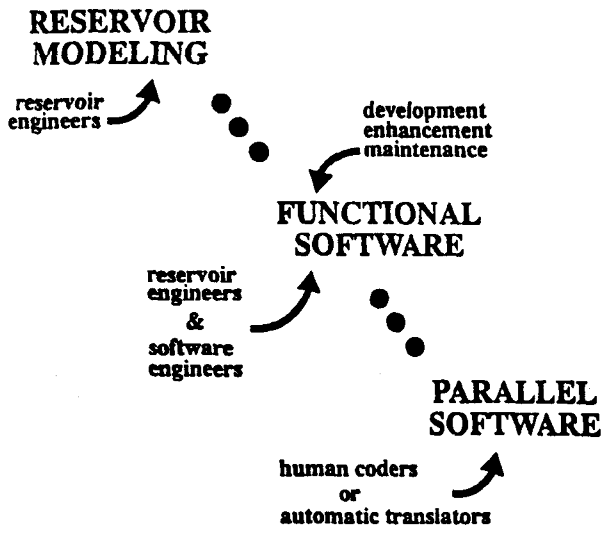

RESERVOIR MODELING
reservoir engineers
development enhancement maintenance
FUNCTIONAL SOFTWARE
reservoir engineers & software engineers
PARALLEL SOFTWARE
human coders or automatic translators

Figure 1: Software Development Strategy

the model and, later, enhancing it. They can even participate fully in the maintenance of the software, and they can do all of this in terms of the equations rather than arcane, procedural details. That is, application scientists can retain control of the basic function of the software while leaving to computing experts the details of encoding the equations for efficient processing in a procedural form.

Because of positive results in the initial experiments (with applications of moderate size), Amoco decided to pursue a more rigorous follow-on experiment involving a large application. Work on the large application selected for this follow-on experiment, oil reservoir modeling, is the topic of this article.

2 The application

Oil reservoir modeling entails the analysis of fluid flow in porous media. Several components of both gaseous and liquid fluids are simulated. A system of partial differential equations in time and space describe the model. Physical laws such as thermodynamic equilibrium and conservation of mass constrain the equations, and physical properties of the reservoir structure, such as rock permeability, pore size, well locations, and the injection and extraction rates of fluids in these wells, affect the coefficients. Factors of ten to a hundred in pressure, temperature, and time scales of interest may be present within a particular simulation. All of these factors contribute to the difficulty of carrying out the computations required in simulations.

Models typically ignore such factors as chemical reactions, rock dissolution, deposition, and other geologic processes. Lack of account of these factors probably has little effect on the accuracy of simulation.

Difference equations are used to approximate the system of differential equations. Time grids for the difference equations tend to be fine-grained (minutes to days) relative to the span of a simulation (months or years), but space grids are typically coarse (hundreds of feet in diameter) relative to the structures that conduct the fluids in the reservoir. These factors affect the accuracy of simulation, but are unavoidable because of limitations in computational capacity.

Amoco's most commonly used reservoir modeling software consists of about 150,000 lines of Fortran code containing many hundreds of procedures and a few hundred COMMON blocks. This is backed up by a thousand pages of engineering documentation that describe equations, algorithms, and several hundred data formats.

Reservoir modeling is a complex activity that consumes many hours of computation time on mainframes, high performance workstations, and vector-based supercomputers. It is also an activity that is central to the business of efficiently recovering hydrocarbon resources, especially in fields where primary recovery methods are no longer effective and secondary or tertiary methods need to be planned.

Reservoir modeling was selected as the target application for this experiment both because of its complexity and because of its importance in the oil industry. In addition, reservoir modeling is a compute-bound application that has not mapped well onto vector supercomputers. It can be cost-effective on such systems, but stays well short of their computational capacities (ten to twenty percent of peak rates). Yet, it appears that much of the computation is amenable to more general sorts of parallel computation than vector supercomputers are able to supply. This potential for cost-effective use of massively parallel computing systems was a third factor in the choice of reservoir modeling as the target application of the experiment.

3 Programming language

The primary criterion for choosing a notation to specify the application in the form of equations was that the notation should require the entire specification to stay within an equational framework. A notation that permitted escapes into the procedural world would undermine the project because it would lead to interminable discussions among the participants at every juncture where someone wanted to use the escape. The result would be a specification that, while it might consist mostly of equations, would contain a substantial procedural element. The procedural element would be likely to impose some sequentiality on parts of the code, which would increase the difficulty of translating the specification to a parallel environment.

Other criteria were that the chosen language should be convenient to use on Unix workstations and should support concise descriptions of computations. Issues such as evaluation strategy (lazy or eager), polymorphism, and support for higher order functions were less important, except for their effects on conciseness.

These criteria led the researchers to choose Miranda [11] as their programming language when the project began in 1989. Standard ML [7] was seriously considered. Its biggest advantage over Miranda was the existence of a compiler able to generate reasonably efficient object code [1]. However, the research team did not choose Standard ML, primarily because they feared the temptation to use elements outside the purely functional core of the language would be difficult to resist and that once the code contained a few procedural sections it would be difficult to estimate the ratio of purely functional code to procedural code in the completed program. That is, the desire to stay within the purely functional paradigm took precedence over the need for efficient object code.

For the most part, Miranda appears to have been a sound choice. However if the choice were to be made today, the range of viable possibilities would be broader (e.g., Id [8], SISAL [9], Clean [12], Haskell[2] [6]).

4 The computation

Fluid flow in an oil reservoir is modeled by a system of partial differential equations in time and space that relate flow rates in the reservoir and movements in mass over time. Initial values in time and boundary values in space are given. Flow rates at wells, where fluids are injected or extracted, receive special attention. Equations are indexed by fluid components (water, hydrocarbons, etc.) and are non-linear.

Finite difference approximations of the differential equations provide the first step toward a solvable model. Since the difference equations are non-linear, a Newton iteration is used to solve them. The Newton iteration involves deriving the coefficients of a system of linear equation, solving that system, and then repeating this process until the Newton iteration converges, usually in three to five steps.

The linear system is large (tens of thousands to hundreds of thousands of equations) and sparse. Iterative methods such as conjugate gradient techniques with preconditioning [5] are used to solve the linear system.

Figure 2 is a schematic diagram of this basic model and computational approach.

Following this model and computation strategy, a typical reservoir computation proceeds as follows. The initial state of the reservoir is described by a quarter to a half megabyte of data. From this state, taking into account the properties of the reservoir, the coefficients of the nonlinear difference equations are derived. Then the Newton iteration, combined with multiple uses of the linear solver, produces a new reservoir state, completing the circuit for one time-step. The simulation continues for a few thousand time steps, and the reservoir state is recorded periodically. Thus, the computation delivers a sequence of reservoir states sampled at various points in the process.

Figure 3 illustrates the flow of data through the parts of the computation. The time required to interpret reservoir states and derive coefficients can range from about the same amount of time as is required to solve the equations up to as much as three times as long.

The process of deriving coefficients involves applying a variety of formulas at points corresponding to different spatial regions in the reservoir. The particular formula applied depends on the composition of fluids at the point of application, but all of these computations can proceed independently. This part of the computation does not map well onto vector-type supercomputers, but should perform well on less

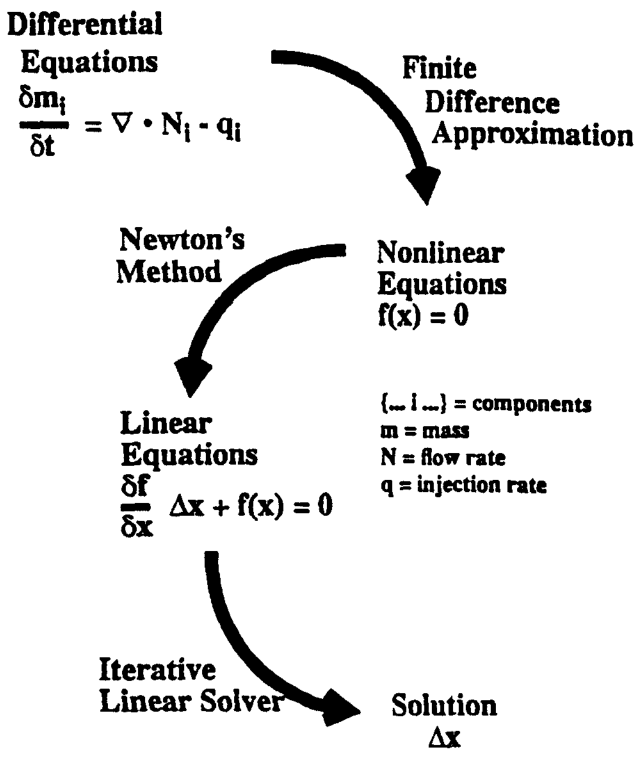

Differential Equations
$\frac{\delta m_i}{\delta t} = \nabla \cdot N_i - q_i$
Finite Difference Approximation
Newton's Method
Nonlinear Equations
$f(x) = 0$
$\{\ldots i \ldots\}$ = components
m = mass
N = flow rate
q = injection rate
Linear Equations
$\frac{\delta f}{\delta x}\Delta x + f(x) = 0$
Iterative Linear Solver
Solution
Δx

Figure 2: Model and Solution Method

[2]At one point, a conversion to Haskell was considered, primarily to address performance problems associated with using the Miranda interpreter in a computationally intense application. The biggest obstacle to conversion was that Haskell distinguishes between integral and floating point variables at the syntactic level, while Miranda makes such distinctions during numeric interpretation. Overloading could have solved the problem if Haskell compilers had been mature enough to handle it efficiently. Or, the problem could have been avoided if the research team had foreseen the difficulty and distinguished between integral and floating point variables by introducing type definitions in the Miranda code. Neither condition held, unfortunately

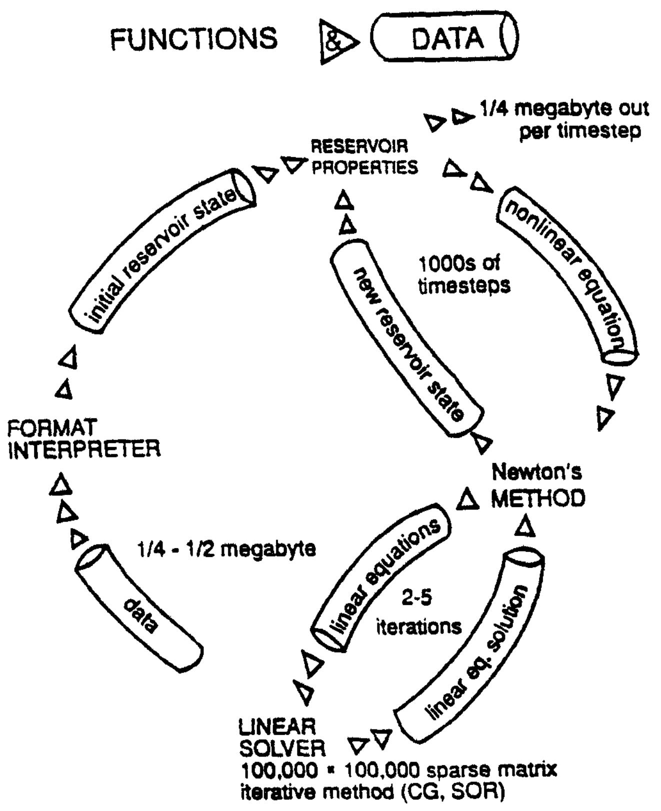


Figure 3: The Computation

constrained parallel systems programmed in single program, multiple data mode (a generalization of single instruction, multiple data mode).

The process of solving the linear system can, depending on the chosen method, run reasonably well on vector oriented computing systems. However, it tends to run poorly on parallel systems with distributed memory because of heavy communication among the computational components. Efficient implementation of these computations on distributed memory systems is an active area of research [10].

Based on current understanding, it appears that an implementation of the reservoir model on a distributed memory system would spend most of its time in the linear solver portion of the software. The generation of the coefficients would be relatively quick because it could use the parallel hardware efficiently.

5 Miranda implementation

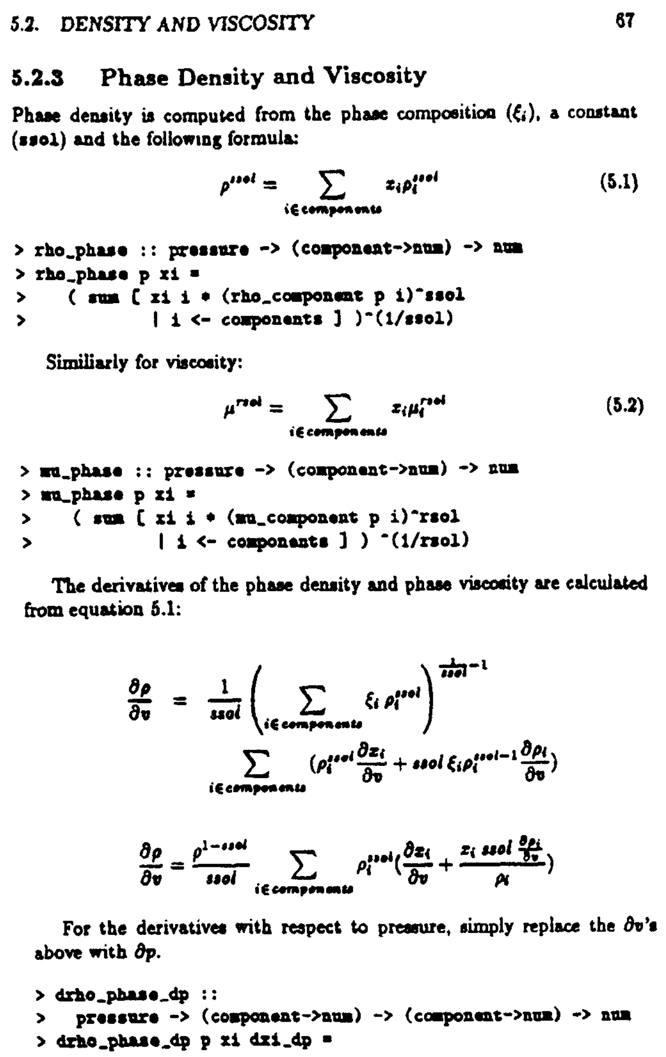
The Miranda software implementing the reservoir model consists of about two hundred pages of code, written as a LaTeXdocument in Miranda's literate mode, where lines to be interpreted by the processor are flagged with a special initial character and all other lines are commentary. Near the beginning of the project, the software developers viewed their creation as a program for a computer—and that is what it looked like. Several months into the project, the developers began to see the software as a document intended for human consumption, and it gradually evolved in that direction.

A surprising phenomenon followed from this new perception of the software: the amount of commentary approximately doubled and the number of flagged lines of code approximately halved[3]. In the new mode, a typical page of the document (Figure 4) presents equations in standard mathematical form (subscripts, superscripts, summation symbols, etc.), much like the usual engineering documentation accompanying a complex piece of software in the scientific domain, plus a Miranda representation of the equations. The mathematical form is intended for application scientists, while the Miranda form addresses the computer. In this way, application scientists can understand the software in terms of the equations in a familiar notation, and computing experts accept the responsibility of making sure the translations to Miranda are accurate.

The Miranda implementation encompasses about a quarter of the function of the Fortran implementation of the model used by practicing reservoir engineers. That is, it permits simulation of about a quarter of the variety of reservoirs that a reservoir simulator for general purpose use in the oil industry would permit. The particular types of models selected for implementation include models of frequent interest in practice, and they tend to concentrate on the more computationally intensive models[4]. Thus, the Miranda software

[3]Some modules in the Miranda program (developed before the idea of software as human communication began to dominate the thinking of the team) have remained in the old style. In such modules, the ratio of code to commentary is about three to one. In modules reflecting the new philosophy, the ratio is reversed: commentary occupies about three quarters of the space. On the average, the Miranda code in its current form is about half commentary and half code.

[4]For example, the Miranda implementation covers "fully implicit" forms of the equations, and this calls for substantially more computation than equations that provide explicit formulas for some terms and leave other terms to be derived from implicit forms.


5.2. *DENSITY AND VISCOSITY* 67

5.2.3 Phase Density and Viscosity

Phase density is computed from the phase composition (ξ_i), a constant (ssol) and the following formula:

$$\rho^{ssol} = \sum_{i \in components} z_i \rho_i^{ssol} \qquad (5.1)$$

```
> rho_phase :: pressure -> (component->num) -> num
> rho_phase p xi =
>     ( sum [ xi i * (rho_component p i)^ssol
>                 | i <- components ] )^(1/ssol)
```

Similiarly for viscosity:

$$\mu^{rsol} = \sum_{i \in components} z_i \mu_i^{rsol} \qquad (5.2)$$

```
> mu_phase :: pressure -> (component->num) -> num
> mu_phase p xi =
>     ( sum [ xi i * (mu_component p i)^rsol
>                 | i <- components ] ) ^(1/rsol)
```

The derivatives of the phase density and phase viscosity are calculated from equation 5.1:

$$\frac{\partial \rho}{\partial v} = \frac{1}{ssol} \left(\sum_{i \in components} \xi_i \rho_i^{ssol} \right)^{\frac{1}{ssol}-1} \sum_{i \in components} (\rho_i^{ssol} \frac{\partial z_i}{\partial v} + ssol\, \xi_i \rho_i^{ssol-1} \frac{\partial \rho_i}{\partial v})$$

$$\frac{\partial \rho}{\partial v} = \frac{\rho^{1-ssol}}{ssol} \sum_{i \in components} \rho_i^{ssol} (\frac{\partial z_i}{\partial v} + \frac{z_i\, ssol\, \frac{\partial \rho_i}{\partial v}}{\rho_i})$$

For the derivatives with respect to pressure, simply replace the ∂v's above with ∂p.

```
> drho_phase_dp ::
>   pressure -> (component->num) -> (component->num) -> num
> drho_phase_dp p xi dxi_dp =
```

Figure 4: Typical Page of the Software Document

admits realistic simulations of oil reservoirs, and it provides a basis for exercising high performance computing systems.

The research team invested about six full-time-equivalent years developing this software, the effort being about equally divided between building a scientific understanding of the underlying reservoir model and programming. A research team composed of application scientists and computing experts (in about equal measure), would probably be able to complete the task a little faster because the application scientists would have a better understanding of the scientific and mathematical issues to be pursued and the appropriate solution strategies than did the research team for this project.

One would like to compare the effort invested in the Miranda version of the reservoir modeling system to the effort required to produce the Fortran version. Unfortunately, the necessary data to estimate the Fortran effort was never captured and cannot now be reliably estimated by the participants. Though less desirable than recorded measurements of actual development effort, one can use standard software production rates to get some idea of relative effort required to produce a Fortran version.

The reservoir simulator in Miranda encompasses a function equivalent to about 25,000 lines of Fortran in a system of production quality. Brooks used data gathered by Corbató to estimate that a programmer typically produces about 1,200 lines of correct code per year [3]. Based on this estimate, the development of a Fortran program equivalent to the reservoir simulator in Miranda would require an effort of about twenty full-time-equivalent work years. This is about three times the total effort invested in the Miranda version, including both programming effort and time spent on the application science side of the project. It is about five times the programming effort invested in the Miranda version[5].

Brooks [3] also commented that the data of Corbató and others make plausible a projection that the rate of production of correct lines of code produced in a large software project will be about 1,200 lines per year per programmer, regardless of what language is used. This figure is a reasonable estimate of the rate of code production in the Miranda version of the reservoir simulator. The difference is that the Miranda version is about five times shorter than a Fortran equivalent, making software production in this application about five times more efficient than the Corbató data suggests for a Fortran version.[6] Section 6 comments on how this efficiency bears on the efficacy of the software development approach (Figure 1) taken in this project.

The general structure of the Miranda program implementing the reservoir model is displayed in Figure 5. Generalizing the software to handle the full variety of simulations of a production quality system would increase the sizes of the modules needed for data interpretation and reservoir

[5]The reservoir model in Miranda contains about five thousand lines of code (disregarding commentary). This reflects a programming effort of between three and four full-time-equivalent work years (not counting the effort invested in understanding the scientific and engineering ideas in reservoir modeling).

[6]As one of the referees pointed out, it may be a bit unfair to compare today's levels of productivity with those of the seventies, the era of the Corbató data and of the initial development of the Fortran version of the reservoir simulator. Developers of the Miranda version had the advantages of Unix (e.g., interactive editors and automatic compilation and linking) in addition to the advantages of a more modern language. One must consider these factors in any interpretation of relative productivity.

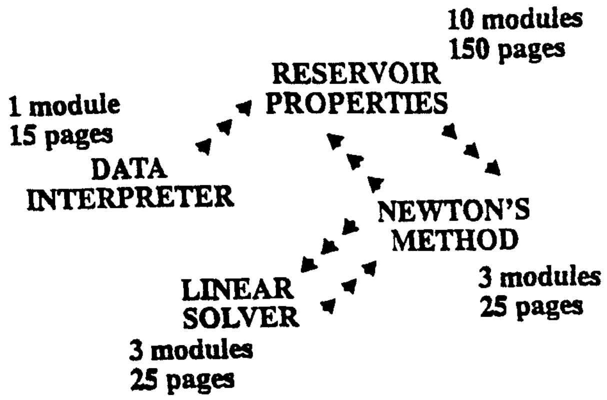


Figure 5: Structure of Miranda Implementation of Reservoir Model

properties. Additional solution methods might be needed in the linear solver portion of the software, so its relative size compared to the overall structure could also increase. The software architecture and flow of data would remain the same.

The Miranda version of the reservoir modeling software was developed directly from the engineering documentation of Amoco's production model. The Fortran code was not consulted, except to validate data used in testing and, in a few cases, to confirm or revise formulas that appeared to be incorrect in the documentation[7]. Even so, a few Fortran-like constructs made their way into the Miranda program, at least temporarily, because the documentation was written with a Fortran implementation in mind.

For example, in the phase calculations, specialized equations were given for the values associated with each phase.

$$x_i = \frac{\xi_i K_i^{ga}}{K_i^{ga}K_i^{go} + l^* K_i^{ga}(1 - K_i^{go}) + \Omega K_i^{go}(1 - K_i^{ga})}$$

$$y_i = \frac{\xi_i K_i^{go} K_i^{ga}}{K_i^{ga}K_i^{go} + l^* K_i^{ga}(1 - K_i^{go}) + \Omega K_i^{go}(1 - K_i^{ga})}$$

$$w_i = \frac{\xi_i K_i^{go}}{K_i^{ga}K_i^{go} + l^* K_i^{ga}(1 - K_i^{go}) + \Omega K_i^{go}(1 - K_i^{ga})}$$

These formulas took into account certain relationships among the K values to reduce the amount of computation required, but at some cost in code complexity.

```
> x kgo kga xi el omega i
>     = (xi!i)*(kga i)/(kga i*kgo i          +
>                       el*kga i*(1-kgo i) +
>                       omega*kgo i*(1-kga i))

> y kgo kga xi el omega i
>     = (xi!i)*(kgo i)*(kga i)/
>                      (kga i*kgo i          +
>                       el*kga i*(1-kgo i) +
>                       omega*kgo i*(1-kga i))

> w kgo kga xi el omega i
>     = (xi!i)*(kgo i)/(kga i*kgo i          +
>                       el*kga i*(1-kgo i) +
>                       omega*kgo i*(1-kga i))
```

By replacing the names x_i, y_i, w_i, l^*, and Ω with the more general names $\zeta_{g,i}$, $\zeta_{o,i}$, $\zeta_{w,i}$, s_o and s_w, and by exploiting (in reverse) the properties of the K values originally used to derive the above formulas, a more general formula emerges which cuts the size of the code substantially and increases its quality as a mathematical communication:

$$\zeta_{m,i} = \frac{\xi_i}{\sum_{r \in phases} s_r K_i^{rm}}$$

```
>  zeta k xi s m i =
>     xi i / sum [ s r * k r m i | r <- phases ]
```

In fact, it also removes a limitation in the original code. The model is designed to deal with three phases, as reflected in the three equations in the documentation. The more general version of the Miranda code, on the other hand, has no such limitation. It is conceivable that this generalization might have advantages in practice at some future date.

The preference for effective communication with people over minor efficiencies in computation was a guiding philosophy in the project. Even parallel computation, a prime consideration in the planned sequel to the development of the Miranda implementation, was not a prime consideration at this stage. The development team came to think of the software more as a monograph on computational reservoir modeling than as a program. The effects of this viewpoint may serve well in future uses of the software.

6 Performance

The Miranda version of the reservoir simulator does its work slowly. Because of poor performance, it has not been possible to test the simulator on large datasets. On small ones it appears to run several thousand times slower than its Fortran equivalent[8]. Performance comparisons on small kernels of floating-point intensive computations such as recursive fast Fourier transforms (Figures 6 and 7) and applications mapping trigonometric functions onto vectors of data indicate that Miranda processes numeric data about a hundred times slower than Fortran.

The reservoir simulator is about another hundred times worse than the performance of such kernels would suggest. It is possible that the development philosophy emphasizing human communication over most other factors has led to a

[7]The process uncovered a few errors in both the engineering documentation and in the Fortran code

[8]For example, the Miranda version running on a high performance Unix workstation computes one time step of a simulation in about a day (a twenty-four hour day) on a dataset representing a two-dimensional section of a reservoir discretized with ten grid blocks (such a small simulation is of no practical import) An equivalent simulator in Fortran could be expected to complete the same computation in seconds.

```
dft:: [complex] -> [complex]
dft xs
 = scale (1/n) (dfth fs xs us)
   where
   us = replicate(map conjugate (rootsOfUnity n))
   fs = factors n
   n = #xs

dfth:: [num]->[complex]->[complex]->[complex]
dfth (f:fs) xs us
 = slowFT xs us,                      if fs=[]
 = map sumComplex (transpose partials), otherwise
   where
   partials = [(pDft k) $times (part 0 k us)
                                    | k<-[0..f-1] ]
   partDft k = repl f (dfth fs (part k f xs)
                               (part 0 f us))
   part offset stride=everyNth stride.drop offset

slowFT:: [complex] -> [complex] -> [complex]
slowFT xs us=[sumComplex(xs $times pus)
               | pus<-[part 0 k us|k<-[0..#xs-1]]]
```

Figure 6: Fast Fourier Transform in Miranda

hundred-fold degradation in potential performance, but it is difficult to believe that most of that loss cannot be recovered without abandoning the philosophy. This conjecture awaits further investigation.

If the hundred-fold loss is endemic to the application, then purely functional programming systems are still a long way from providing the type of support needed to develop applications in the mode suggested in this paper. However, if (as seems more likely) the hundred-fold loss is mostly recoverable from analysis and revision of the code, then Miranda is only a factor of a hundred away from Fortran performance. Miranda is mostly interpretive[9]. It seems reasonable to expect that a Miranda compiler could provide performance improvements of at least an order of magnitude over the existing interpretive system[10]. This would put it within a factor of five to ten of Fortran performance.

Most application scientists would not use, in practice, a code that ran five times slower than they could expect a Fortran code to perform (maybe a factor of two, with a lot of cajoling, but never a factor of five). This suggests that dropping the second part of the strategy (translating by hand to efficient procedural code), will require improvements in compilers for functional languages that do not appear to be eminent.

Nevertheless, relative performance within an order of magnitude would make potentially feasible the mode of software development suggested in this project (Figure 1), combining the expertise of application scientists and computing scientists in the development of a constructive specification based on equations (e.g., a Haskell or Miranda program) and translating this by hand into an efficient procedural code for

[9]However, many of the functions in the Miranda library are coded in C, especially higher order functions like the folds and map. Once the interpreter gets into a section of code using these functions, it runs at compiled speeds.

[10]Augustsson's Haskell compiler, hbc, may already deliver performance in this range, based on tests of FFT and mapped trigonometric kernel functions [2].

```
subroutine dft(x,n, f) ! Warning: n>2048 is fatal
  complex x(0:n-1), f(0:n-1)
  complex u(0:2047)
  integer p(11), np
  call factor(n, p,np)
  call roots(n, u)
  call conj(n, u)
  call dfth(p,np,x,u,n, f)
  call fscale(n, f)
end

subroutine dfth(p,np,x,u,n, f)
  implicit automatic(a-z)
  integer p(np)
  complex x(0:n-1), u(0:n-1), f(0:n-1)
  complex xPrt(0:2047), uPrt(0:1023),fPrt(0:1023)
  if (np .eq. 1) then
    call sfth(x,u,n, f)
  else
    call evNth(p(1),u,n, uPrt)
    nr = n/p(1)
    do i=0,n-1
      f(i) = 0
    end do
    do k=0,p(1)-1    ! do partial dft's
      call evNth(p(1),x(k),n, xPrt)
      call dfth(p(2),np-1,xPrt,uPrt,nr, fPrt)
      do i=0,nr-1          ! mix-in partials
        do j=0,p(1)-1
          f(i+j*nr) =                                &
          f(i+j*nr) + fPrt(i)*u(mod((i+j*nr)*k,n))
        end do
      end do               ! end mix-in
    end do           ! end partial dft's
  end if
end

subroutine sfth(x,u,n, f)
  complex x(0:n-1), u(0:n-1), f(0:n-1)
  complex fdot; external fdot
  do i=0,n-1
    f(i) = fdot(x,u,n,i)
  end do
end

complex function fdot(x,u,n,k)
  complex x(0:n-1), u(0:n-1)
  integer n, k
  fdot = 0
  do i=0,n-1
    fdot = fdot + x(i)*u(mod(i*k,n))
  end do
end
```

Figure 7: Fast Fourier Transform in Fortran

a high performance computing system (probably a massively parallel one[11]). In the current project, it appears that the development of an equational specification goes about five times faster than the development of a Fortran program of production quality. If the translation from equational to procedural form consumes less than four times the effort invested in the equation based program, then this overall strategy is competitive with conventional software development methods.

The new strategy offers a few advantages beyond the potential for raw cost savings in initial development. One advantage is that application scientists can devote more of their time to work in their field of expertise and less of their time fighting the vagaries of computing systems. Another is that experiments with new ideas intended to enhance the application can be carried out rapidly in terms of equations rather than through modifications of procedural code (which one might expect to require several times more effort). A third advantage is that the equation based program is a living document communicating to both application scientists and computers the precise details of the application. Finally, the equation-based program places fewer constraints on computational procedure than a conventional program would. Therefore, it leaves open more opportunities for optimization on a variety of computing systems, including, especially, various massively parallel architectures.

One concludes that, while better tools than the ones used in this project will be needed to make software development in the proposed mode practical, such tools may well be available now. What will remain is the problem of convincing application scientists that they can be more effective if they communicate their ideas in the form of equations than in the form of Fortran programs. They will need to be convinced that they can work more effectively when they let computing scientists manipulate these equations into functional programs as a first step and efficient procedural programs as a second step[12]. If this organizational problem can be solved, the method can get tested in ways that go beyond the results of the present experiment.

7 Future work

The reservoir model in Miranda in its original form contains equations that Amoco regards as proprietary. However, a revised version replaces the proprietary portions of the model by non-proprietary equations that, nevertheless, provide for adequate modeling. This version is suitable for publication in the public domain [4]. The public domain version of the model can provide the basis for experiments in translating a large scientific application from equations to procedural code. The research team hopes experiments of this kind will be carried out in a variety of massively parallel computing environments.

Another use envisioned for the public domain version of the software is in the area of benchmarking. The performance of functional language processors in the domain of scientific applications can be significantly challenged with a program of this magnitude. As it stands, of course, the program can serve only as a benchmark for Miranda processors, but translations to related notations may be a reasonable task whose benefits could justify the required investment.

Such a benchmark would be valuable not only for testing processing speed and space utilization, but also for evaluating tools emerging in the functional programming domain. In fact the process of translating the software from Miranda to other notations offers opportunities to use such tools in a realistic project. Other possible uses of the code include gauging the importance of lazy evaluation in functional programs and measuring its effects on performance[13] and for measuring the value of implementation strategies for various features of functional languages. The research team hopes that this code can serve as a serious test for functional language processors and related tools well into the future.

[11] As one of the referees suggested, it could happen, as it did in the earlier Amoco experiment mentioned in Section 1 [13], that the functional presentation would clarify opportunities for important gains in parallel computation

[12] Another problem is convincing first rate computing scientists that scientific application programs provide an interesting and rewarding arena in which to apply their talents.

8 Acknowledgements

The authors would like to express their deep gratitude for the efforts over the past few years of the research team that carried out much of the work that this article reports. Their work, performed at Amoco Production Company's Research Center in Tulsa, made this project a success.

Roger Wainwright developed a significant portion of the Miranda code in the early years. He also helped the team understand some of the fine points in the mathematics of the engineering documentation that described the model from which the software was derived, especially when the mathematical formulas contained small errors that tend not to mislead reservoir engineers but can cause major problems for people who are not experts in the area.

Guy Argo analyzed alternative language possibilities at a time when conversion to a compiled language for performance reasons was being considered. He also prepared a Haskell version (now in the public domain) of the linear solver to use as a test environment.

Daniel Vasicek served as the primary interpreter of the engineering documentation and as the link to the Fortran version of the model for purposes of dataset creation, testing, and comparison of function. He also documented the overall structure of the Miranda software, which helped, early on, in project scheduling.

Julio Díaz applied his extensive expertise in sparse linear systems to develop a method effective for the kinds of equations arising in the simulation. It would have been difficult to succeed without his acting as "resident reservoir engineer" for the group, Julio being the only member of the team with significant prior experience in reservoir modeling.

Thanks guys. It was fun.

References

[1] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Lecture Notes in Computer Science 274: Functional Programming Languages and Computer Architecture*, pages 301-324, Portland, Oregon, September 1987. Springer-Verlag.

[13] One of the referees pointed out this possibility

[2] Lennart Augustsson, December 1992. Personal communication, email Message-Id: 9212162140.AA11688 @ animal.cs.chalmers.se.

[3] Frederick P. Brooks. *The Mythical Man-Month*, page 93. Addison-Wesley, 1975.

[4] Julio-César Díaz, Brian D. Moe, Rex L. Page, Daniel J. Vasicek, and Roger L. Wainwright. A reservoir model described as a computationally constructive system of equations. Technical report, University of Tulsa, Department of Mathematics and Computer Science, 1993. In preparation.

[5] Julio-César Díaz and Kamini Shenoi. Domain decomposition extensions of reservoir preconditioning to the well equations in reservoir simulation. In *Domain Decomposition Conference*, Como, Italy, June 1992.

[6] P. Hudak, S. Peyton-Jones, P. Wadler, et al. Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992. Section R.

[7] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[8] R. S. Nikhil. Id (version 90.0) reference manual. Technical report, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1990.

[9] R. R. Oldehoeft, D. C. Cann, J. T. Feo, and A. P. W. Bohm. The SISAL 2.0 reference manual. Technical Report UCRL-MA-109098, Lawrence Livermore National Laboratory, December 1991.

[10] J. M. Rutledge, D. R. Jones, W. H. Chen, and E. Y. Chung. The use of a massively parallel SIMD computer for reservoir simulation. In *Eleventh SPE Symposium on Reservoir Simulation*, pages 117–124, Anaheim, California, February 1991.

[11] David A. Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, December 1986.

[12] Mark van Eekelen, Eric Nöcker, Rinus Plasmeijer, and Sjaak Smetsers. Concurrent clean (version 0.6/0.7). Technical report, University of Nijmegen, November 1990.

[13] Roger L. Wainwright. Deriving parallel computations from functional specifications: a seismic example on a hypercube. *International Journal of Parallel Programming*, 16(3):243–260, March 1987.