Chapter 22. Introduction to Parallel Programming

22.0 Why Think Parallel?

In recent years we Numerical Recipes authors have increasingly become convinced that a certain revolution, cryptically denoted by the words "parallel programming," is about to burst forth from its gestation and adolescence in the community of supercomputer users, and become the mainstream methodology for all computing.

Let's review the past: Take a screwdriver and open up the computer (workstation or PC) that sits on your desk. (Don't blame us if this voids your warranty; and be sure to unplug it first!) Count the integrated circuits — just the bigger ones, with more than a million gates (transistors). As we write, in 1995, even lowly memory chips have one or four million gates, and this number will increase rapidly in coming years. You'll probably count at least dozens, and often hundreds, of such chips in your computer.

Next ask, how many of these chips are CPUs? That is, how many implement von Neumann processors capable of executing arbitrary, stored program code? For most computers, in 1995, the answer is: about one. A significant number of computers do have secondary processors that offload input-output and/or video functions. So, two or three is often a more accurate answer, but only one is usually under the user's direct control.

Why do our desktop computers have dozens or hundreds of memory chips, but most often only one (user-accessible) CPU? Do CPU chips intrinsically cost more to manufacture? No. Are CPU chips more expensive than memory chips? Yes, primarily because fixed development and design costs must be distributed over a smaller number of units sold. We have been in a kind of economic equilibrium: CPU's are relatively expensive because there is only one per computer; and there is only one per computer, because they are relatively expensive.

Stabilizing this equilibrium has been the fact that there has been no standard, or widely taught, methodology for parallel programming. Except for the special case of scientific computing on supercomputers (where large problems often have a regular or geometric character), it is not too much of an exaggeration to say that nobody *really knows how* to program multiprocessor machines. Symmetric multiprocessor

operating systems, for example, have been very slow in developing; and efficient, parallel methodologies for query-serving on large databases are even now a subject of continuing research.

However, things are now changing. We consider it an easy prognostication that, by the first years of the new century, the typical desktop computer will have 4 to 8 user-accessible CPUs; ten years after that, the typical number will be between 16 and 512. It is not coincidence that these numbers are characteristic of supercomputers (including some quite different architectures) in 1995. The rough rule of ten years' lag from supercomputer to desktop has held firm for quite some time now.

Scientists and engineers have the advantage that techniques for parallel computation in their disciplines *have* already been developed. With multiprocessor workstations right around the corner, we think that now is the right time for scientists and engineers who use computers to start *thinking parallel*. We don't mean that you should put an axe through the screen of your fast serial (single-CPU) workstation. We do mean, however, that you should start programming somewhat differently on that workstation, indeed, start thinking a bit differently about the way that you approach numerical problems in general.

In this volume of *Numerical Recipes in Fortran*, our pedagogical goal is to show you that there are conceptual and practical benefits in parallel thinking, even if you are using a serial machine today. These benefits include conciseness and clarity of code, reusability of code in wider contexts, and (not insignificantly) increased portability of code to today's parallel supercomputers. Of course, on parallel machines, either supercomputers today or desktop machines tomorrow, the benefits of thinking parallel are much more tangible: They translate into significant improvements in efficiency and computational capability.

Thinking Parallel with Fortran 90

Until very recently, a strong inhibition to thinking parallel was the lack of any standard, architecture-independent, computer language in which to think. That has changed with the finalization of the Fortran 90 language standard, and with the availability of good, optimizing Fortran 90 compilers on a variety of platforms.

There is a significant body of opinion (with which we, however, disagree) that there is no such thing as architecture-independent parallel programming. Proponents of this view, who are generally committed wizards at programming on one or another particular architecture, point to the fact that algorithms that are optimized to one architecture can run hundreds of times more slowly on other architectures. And, they are correct!

Our opposing point of view is one of pragmatism. We think that it is not hard to learn, in a general way, what kinds of architectures are in general use, and what kinds of parallel constructions work well (or poorly) on each kind. With this knowledge (much of which we hope to develop in this book) the user can, we think, write good, general-purpose parallel code that works on a variety of architectures — including, importantly, on purely serial machines. Equally important, the user will be aware of when certain parts of a code can be significantly improved on some, but not other, architectures.

Fortran 90 is a good test-bench for this point of view. It is not the perfect language for parallel programming. But it is *a* language, and it is the only

cross-platform *standard* language now available. The committee that developed the language between 1978 and 1991 (known technically as X3J3) had strong representation from both a traditional "vectorization" viewpoint (e.g., from the Cray XMP and YMP series of computers), and also from the "data parallel" or "SIMD" viewpoints of parallel machines like the CM-2 and CM-5 from Thinking Machines, Inc. Language compromises were made, and a few (in our view) almost essential features were left out (see §22.5). But, by and large, the necessary tools are there: If you learn to think parallel in Fortran 90, you will easily be able to transfer the skill to future parallel standards, whether they are Fortran-based, C-based, or other.

CITED REFERENCES AND FURTHER READING:

Metcalf, M., and Reid, J. 1996, *Fortran 90/95 Explained* (Oxford: Oxford University Press).

22.1 Fortran 90 Data Parallelism: Arrays and Intrinsics

The underlying model for parallel computation in Fortran 90 is *data parallelism*, implemented by the use of arrays of data, and by the provision of operations and intrinsic functions that act on those arrays in parallel, in a manner optimized by the compiler for each particular hardware architecture. We will not try to draw a fine definitional distinction between "data parallelism" and so-called SIMD (single instruction multiple data) programming. For our purposes the two terms mean about the same thing: The programmer writes a single operation, "+" say, and the compiler causes it to be carried out on multiple pieces of data in as parallel a manner as the underlying hardware allows.

Any kind of parallel computing that is not SIMD is generally called MIMD (multiple instruction multiple data). A parallel programming language with MIMD features might allow, for example, several different subroutines — acting on different parts of the data — to be called into execution simultaneously. Fortran 90 has few, if any, MIMD constructions. A Fortran 90 compiler might, on some machines, execute MIMD code in implementing some Fortran 90 intrinsic functions (`pack` or `unpack`, e.g.), but this will be hidden from the Fortran 90 user. Some extensions of Fortran 90, like HPF, do implement MIMD features explicitly; but we will not consider these in this book. Fortran 95's `forall` and `PURE` extensions (see §21.6) will allow some significantly greater access to MIMD features (see §22.5).

Array Parallel Operations

We have already met the most basic, and most important, parallel facility of Fortran 90, namely, the ability to use whole arrays in expressions and assignments, with the indicated operations being effected in parallel across the array. Suppose, for example, we have the two-dimensional matrices `a`, `b`, and `c`,

```
REAL, DIMENSION(30,30) :: a,b,c
```

Then, instead of the serial construction,

```
do j=1,30
   do k=1,30
      c(j,k)=a(j,k)+b(j,k)
   end do
end do
```

which is of course perfectly valid Fortran 90 code, we can simply write

```
c=a+b
```

The compiler deduces from the declaration statement that `a`, `b`, and `c` are matrices, and what their bounding dimensions are.

Let us dwell for a moment on the conceptual differences between the serial code and parallel code for the above matrix addition. Although one is perhaps used to seeing the nested do-loops as simply an idiom for "do-the-enclosed-on-all-components," it in fact, according to the rules of Fortran, specifies a very particular time-ordering for the desired operations. The matrix elements are added by rows, in order (`j=1,30`), and within each row, by columns, in order (`k=1,30`).

In fact, the serial code above *overspecifies* the desired task, since it is guaranteed by the laws of mathematics that the order in which the element operations are done is of no possible relevance. Over the 50 year lifetime of serial von Neuman computers, we programmers have been brainwashed to break up all problems into single executable streams *in the time dimension only*. Indeed, the major design problem for supercomputer compilers for the last 20 years has been to *undo* such serial constructions and recover the underlying "parallel thoughts," for execution in vector or parallel processors. Now, rather than taking this expensive detour into and out of serial-land, we are asked simply to say what we mean in the first place, `c=a+b`.

The essence of parallel programming is *not* to force "into the time dimension" (i.e., to serialize) operations that naturally extend across a span of data, that is, "in the space dimension." If it were not for 50-year-old collective habits, and the languages designed to support them, parallel programming would probably strike us as more natural than its serial counterpart.

Broadcasts and Dimensional Expansion: SSP vs. MMP

We have previously mentioned the Fortran 90 rule that a scalar variable is conformable with any shape array. Thus, we can implement a calculation such as

$$y_i = x_i + s, \qquad i = 1, \ldots, n \tag{22.1.1}$$

with code like

```
y=x+s
```

where we of course assume previous declarations like

```
REAL(SP) :: s
REAL(SP), DIMENSION(n) :: x,y
```

with `n` a compile-time constant or dummy argument. (Hereafter, we will omit the declarations in examples that are this simple.)

This seemingly simple construction actually hides an important underlying parallel capability, namely, that of *broadcast*. The sums in `y=x+s` are done in parallel

on different CPUs, each CPU accessing different components of x and y. Yet, they all must access the same scalar value s. If the hardware has local memory for each CPU, the value of s must be replicated and transferred to each CPU's local memory. On the other hand, if the hardware implements a single, global memory space, it is vital to do something that mitigates the traffic jam potentially caused by all the CPUs trying to access the same memory location at the same time. (We will use the term "broadcast" to refer equally to both cases.) Although hidden from the user, Fortran 90's ability to do broadcasts is an essential feature of it as a parallel language.

Broadcasts can be more complicated than the above simple example. Consider, for example, the calculation

$$w_i = \sum_{j=1}^{n} |x_i + x_j|, \qquad i = 1, \ldots, n \tag{22.1.2}$$

Here, we are doing n^2 operations: For each of n values of i there is a sum over n values of j.

Serial code for this calculation might be

```
do i=1,n
   w(i)=0.
   do j=1,n
      w(i)=w(i)+abs(x(i)+x(j))
   end do
end do
```

The obvious immediate parallelization in Fortran 90 uses the sum intrinsic function to eliminate the inner do-loop. This would be a suitable amount of parallelization for a small-scale parallel machine, with a few processors:

```
do i=1,n
   w(i)=sum(abs(x(i)+x))
end do
```

Notice that the conformability rule implies that a new value of x(i), a scalar, is being broadcast to all the processors involved in the abs and sum, with each iteration of the loop over i.

What about the outer do-loop? Do we need, or want, to eliminate it, too? That depends on the architecture of your computer, and on the tradeoff between time and memory in your problem (a common feature of all computing, no less so parallel computing). Here is an implementation that is free of all do-loops, in principle capable of being executed in a small number (independent of n) of parallel operations:

```
REAL(SP), DIMENSION(n,n) :: a
...
a = spread(x,dim=2,ncopies=n)+spread(x,dim=1,ncopies=n)
w = sum(abs(a),dim=1)
```

This is an example of what we call *dimensional expansion*, as implemented by the spread intrinsic. Although the above may strike you initially as quite a cryptic construction, it is easy to learn to read it. In the first assignment line, a matrix is constructed with all possible values of x(i)+x(j). In the second assignment line, this matrix is collapsed back to a vector by applying the sum operation to the absolute value of its elements, across one of its dimensions.

More explicitly, the first line creates a matrix a by adding two matrices each constructed via spread. In spread, the dim argument specifies which argument is *duplicated*, so that the first term *varies* across its first (row) dimension, and vice versa for the second term:

$$a_{ij} = x_i + x_j$$
$$= \begin{pmatrix} x_1 & x_1 & x_1 & \dots \\ x_2 & x_2 & x_2 & \dots \\ x_3 & x_3 & x_3 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} + \begin{pmatrix} x_1 & x_2 & x_3 & \dots \\ x_1 & x_2 & x_3 & \dots \\ x_1 & x_2 & x_3 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{22.1.3}$$

Since equation (22.1.2) above is symmetric in i and j, it doesn't really matter what value of dim we put in the sum construction, but the value dim=1 corresponds to summing across the rows, that is, down each column of equation (22.1.3).

Be sure that you understand that the spread construction changed an $O(n)$ memory requirement into an $O(n^2)$ one! If your values of n are large, this is an impossible burden, and the previous implementation with a single do-loop remains the only practical one. On the other hand, if you are working on a massively parallel machine, whose number of processors is comparable to n^2 (or at least much larger than n), then the spread construction, and the underlying broadcast capability that it invokes, leads to a big win: All n^2 operations can be done in parallel. This distinction between small-scale parallel machines — which we will hereafter refer to as *SSP machines* — and massively multiprocessor machines — which we will refer to as *MMP machines* — is an important one. A main goal of parallelism is to saturate the available number of processors, and algorithms for doing so are often different in the SSP and MMP opposite limits. Dimensional expansion is one method for saturating processors in the MMP case.

Masks and "Index Loss"

An instructive extension of the above example is the following case of a product that omits one term (the diagonal one):

$$w_i = \prod_{\substack{j=1 \\ j \neq i}}^{n} (x_j - x_i), \qquad i = 1, \dots, n \tag{22.1.4}$$

Formulas like equation (22.1.4) frequently occur in the context of interpolation, where all the x_i's are known to be distinct, so let us for the moment assume that this is the case.

Serial code for equation (22.1.4) could be

```
do i=1,n
   w(i)=1.0_sp
   do j=1,n
      if (j /= i) w(i)=w(i)*(x(j)-x(i))
   end do
end do
```

Parallel code for SSP machines, or for large enough n on MMP machines, could be

```
do i=1,n
   w(i)=product( x-x(i), mask=(x/=x(i)) )
end do
```

Here, the `mask` argument in the `product` intrinsic function causes the diagonal term to be omitted from the product, as we desire. There are some features of this code, however, that bear commenting on.

First, notice that, according to the rules of conformability, the expression `x/=x(i)` broadcasts the scalar `x(i)` and generates a logical array of length n, suitable for use as a `mask` in the `product` intrinsic. It is quite common in Fortran 90 to generate masks "on the fly" in this way, particularly if the mask is to be used only once.

Second, notice that the `j` index has disappeared completely. It is now implicit in the two occurrences of `x` (equivalent to `x(1:n)`) on the right-hand side. With the disappearance of the `j` index, we also lose the ability to do the test on `i` and `j`, but must use, in essence, `x(i)` and `x(j)` instead! That is a very general feature in Fortran 90: when an operation is done in parallel across an array, there is *no associated index* available within the operation. This "index loss," as we will see in later discussion, can sometimes be quite an annoyance.

A language construction present in CM [Connection Machine] Fortran, the so-called `forall`, which would have allowed access to an associated index in many cases, was eliminated from Fortran 90 by the X3J3 committee, in a controversial decision. Such a construction will come into the language in Fortran 95.

What about code for an MMP machine, where we are willing to use dimensional expansion to achieve greater parallelism? Here, we can write,

```
a = spread(x,dim=2,ncopies=n)-spread(x,dim=1,ncopies=n)
w = product(a,dim=1,mask=(a/=0.))
```

This time it does matter that the value of `dim` in the `product` intrinsic is 1 rather than 2. If you write out the analog of equation (22.1.3) for the present example, you'll see that the above fragment is the right way around. The problem of index loss is still with us: we have to construct a mask from the array `a`, not from its indices, *both* of which are now lost to us!

In most cases, there are workarounds (more, or less, awkward as they may be) for the problem of index loss. In the worst cases, which are quite rare, you have to create objects to hold, and thus bring back into play, the lost indices. For example,

```
INTEGER(I4B), DIMENSION(n) :: jj
...
jj = (/ (i,i=1,n) /)
do i=1,n
   w(i)=product( x-x(i), mask=(jj/=i) )
end do
```

Now the array `jj` is filled with the "lost" j index, so that it is available for use in the mask. A similar technique, involving spreads of `jj`, can be used in the above MMP code fragment, which used dimensional expansion. (Fortran 95's `forall` construction will make index loss much less of a problem. See §21.6.)

Incidentally, the above Fortran 90 construction, `(/ (i,i=1,n) /)`, is called an *array constructor with implied do list*. For reasons to be explained in §22.2, we almost never use this construction, in most cases substituting a Numerical Recipes utility function for generating arithmetical progressions, which we call `arth`.

Interprocessor Communication Costs

It is both a blessing and a curse that Fortran 90 completely hides from the user the underlying machinery of interprocessor communication, that is, the way that data values computed by (or stored locally near) one CPU make their way to a different CPU that might need them next. The blessing is that, by and large, the Fortran 90 programmer need not be concerned with how this machinery works. If you write

```
a(1:10,1:10) = b(1:10,1:10) + c(10:1:-1,10:1:-1)
```

the required upside-down-and-backwards values of the array `c` are just *there*, no matter that a great deal of routing and switching may have taken place. An ancillary blessing is that this book, unlike so many other (more highly technical) books on parallel programming (see references below) need not be filled with complex and subtle discussions of CPU connectivity, topology, routing algorithms, and so on.

The curse is, just as you might expect, that the Fortran 90 programmer can't control the interprocessor communication, even when it is desirable to do so. A few regular communication patterns are "known" to the compiler through Fortran 90 intrinsic functions, for example `b=transpose(a)`. These, presumably, are done in an optimal way. However, many other regular patterns of communication, which might also allow highly optimized implementations, don't have corresponding intrinsic functions. (An obvious example is the "butterfly" pattern of communication that occurs in fast Fourier transforms.) These, if coded in Fortran 90 by using general vector subscripts (e.g., `barr=arr(iarr)` or `barr(jarr)=arr`, where `iarr` and `jarr` are integer arrays), lose all possibility of being optimized. The compiler can't distinguish a communication step with regular structure from one with general structure, so it must assume the worst case, potentially resulting in very slow execution.

About the only thing a Fortran 90 programmer can do is to start with a general awareness of the kind of apparently parallel constructions that *might* be quite slow on his/her parallel machine, and then to refine that awareness by actual experience and experiment. Here is our list of constructions most likely to cause interprocessor communication bottlenecks:

- vector subscripts, like `barr=arr(iarr)` or `barr(jarr)=arr` (that is, general gather/scatter operations)
- the `pack` and `unpack` intrinsic functions
- mixing positive strides and negative strides in a single expression (as in the above `b(1:10,1:10)+c(10:1:-1,10:1:-1)`)
- the `reshape` intrinsic when used with the `order` argument
- possibly, the `cshift` and `eoshift` extrinsics, especially for nonsmall values of the shift.

On the other hand, the fact is that these constructions *are* parallel, and *are* there for you to use. If the alternative to using them is strictly serial code, you should almost always give them a try.

Linear Algebra

You should be alert for opportunities to use combinations of the `matmul`, `spread`, and `dot_product` intrinsics to perform complicated linear algebra calculations. One useful intrinsic that is not provided in Fortran 90 is the *outer product*

of two vectors,

$$c_{ij} = a_i b_j \tag{22.1.5}$$

We already know how to implement this (cf. equation 22.1.3):

```
c = spread(a,dim=2,ncopies=size(b))*spread(b,dim=1,ncopies=size(a))
```

In fact, this operation occurs frequently enough to justify making it a utility function, `outerprod`, which we will do in Chapter 23. There we also define other "outer" operations between vectors, where the multiplication in the outer product is replaced by another binary operation, such as addition or division.

Here is an example of using these various functions: Many linear algebra routines require that a submatrix be updated according to a formula like

$$a_{jk} = a_{jk} + b_i a_{ji} \sum_{p=i}^{m} a_{pi} a_{pk}, \qquad j = i, \ldots, m, \quad k = l, \ldots, n \tag{22.1.6}$$

where i, m, l, and n are fixed values. Using an array slice like `a(:,i)` to turn a_{pi} into a vector indexed by p, we can code the sum with a `matmul`, yielding a vector indexed by k:

```
temp(l:n)=b(i)*matmul(a(i:m,i),a(i:m,l:n))
```

Here we have also included the multiplication by b_i, a scalar for fixed i. The vector `temp`, along with the vector a_{ji} = `a(:,i)`, is then turned into a matrix by the `outerprod` utility and used to increment a_{jk}:

```
a(i:m,l:n)=a(i:m,l:n)+outerprod(a(i:m,i),temp(l:n))
```

Sometimes the update formula is similar to (22.1.6), but with a slight permutation of the indices. Such cases can be coded as above if you are careful about the order of the quantities in the `matmul` and the `outerprod`.

CITED REFERENCES AND FURTHER READING:

Akl, S.G. 1989, *The Design and Analysis of Parallel Algorithms* (Englewood Cliffs, NJ: Prentice Hall).

Bertsekas, D.P., and Tsitsiklis, J.N. 1989, *Parallel and Distributed Computation: Numerical Methods* (Englewood Cliffs, NJ: Prentice Hall).

Carey, G.F. 1989, *Parallel Supercomputing: Methods, Algorithms, and Applications* (New York: Wiley).

Fountain, T.J. 1994, *Parallel Computing: Principles and Practice* (New York: Cambridge University Press).

Golub, G., and Ortega, J.M. 1993, *Scientific Computing: An Introduction with Parallel Computing* (San Diego, CA: Academic Press).

Fox, G.C., et al. 1988, *Solving Problems on Concurrent Processors*, Volume I (Englewood Cliffs, NJ: Prentice Hall).

Hockney, R.W., and Jesshope, C.R. 1988, *Parallel Computers 2* (Bristol and Philadelphia: Adam Hilger).

Kumar, V., et al. 1994, *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms* (Redwood City, CA: Benjamin/Cummings).

Lewis, T.G., and El-Rewini, H. 1992, *Introduction to Parallel Computing* (Englewood Cliffs, NJ: Prentice Hall).

Modi, J.J. 1988, *Parallel Algorithms and Matrix Computation* (New York: Oxford University Press).

Smith, J.R. 1993, *The Design and Analysis of Parallel Algorithms* (New York: Oxford University Press).

Van de Velde, E. 1994, *Concurrent Scientific Computing* (New York: Springer-Verlag).

22.2 Linear Recurrence and Related Calculations

We have already seen that Fortran 90's *array constructor with implied do list* can be used to generate simple series of integers, like `(/ (i,i=1,n) /)`. Slightly more generally, one might want to generate an arithmetic progression, by the formula

$$v_j = b + (j-1)a, \qquad j = 1, \ldots, n \tag{22.2.1}$$

This is readily coded as

```
v(1:n) = (/ (b+(j-1)*a, j=1,n) /)
```

Although it is concise, and valid, *we don't like this coding.* The reason is that it violates the fundamental rule of "thinking parallel": it turns a parallel operation across a data vector into a serial do-loop over the components of that vector. Yes, we know that the compiler might be smart enough to generate parallel code for implied do lists; but it also might *not* be smart enough, here or in more complicated examples.

Equation (22.2.1) is also the simplest example of a *linear recurrence relation*. It can be rewritten as

$$v_1 = b, \qquad v_j = v_{j-1} + a, \qquad j = 2, \ldots, n \tag{22.2.2}$$

In this form (assuming that, in more complicated cases, one doesn't know an explicit solution like equation 22.2.1) one can't write an explicit array constructor. Code like

```
v(1) = b
v(2:n) = (/ (v(j-1)+a,j=2,n) /)     ! wrong
```

is legal Fortran 90 syntax, but illegal semantics; it does *not* do the desired recurrence! (The rules of Fortran 90 require that all the components of `v` on the right-hand side be evaluated before any of the components on the left-hand side are set.) Yet, as we shall see, techniques for accomplishing the evaluation in parallel are available.

With this as our starting point, we now survey some particular tricks of the (parallel) trade.

Subvector Scaling: Arithmetic and Geometric Progressions

For explicit arithmetic progressions like equation (22.2.1), the simplest parallel technique is *subvector scaling* [1]. The idea is to work your way through the desired vector in larger and larger parallel chunks:

$$\begin{aligned} v_1 &= b \\ v_2 &= b + a \\ v_{3\ldots4} &= v_{1\ldots2} + 2a \\ v_{5\ldots8} &= v_{1\ldots4} + 4a \\ v_{9\ldots16} &= v_{1\ldots8} + 8a \end{aligned} \tag{22.2.3}$$

And so on, until you reach the length of your vector. (The last step will not necessarily go all the way to the next power of 2, therefore.) The powers of 2, times a, can of course be obtained by successive doublings, rather than the explicit multiplications shown above.

You can see that subvector scaling requires about $\log_2 n$ parallel steps to process a vector of length n. Equally important for serial machines, or SSP machines, the scalar operation count for subvector scaling is no worse than entirely serial code: each new component v_i is produced by a single addition.

If addition is replaced by multiplication, the identical algorithm will produce geometric progressions, instead of arithmetic progressions. In Chapter 23, we will use subvector scaling to implement our utility functions `arth` and `geop` for these two progressions. (You can then call one of these functions instead of recoding equation 22.2.3 every time you need it.)

Vector Reduction: Evaluation of Polynomials

Logically related to subvector scaling is the case where a calculation can be parallelized across a vector that *shrinks* by a factor of 2 in each iteration, until a desired *scalar* result is reached. A good example of this is the parallel evaluation of a polynomial [2]

$$P(x) = \sum_{j=0}^{N} c_j x^j \tag{22.2.4}$$

For clarity we take the special case of $N = 5$. Start with the vector of coefficients (imagining appended zeros, as shown):

$$c_0, \quad c_1, \quad c_2, \quad c_3, \quad c_4, \quad c_5, \quad 0, \quad \ldots$$

Now, add the elements by pairs, multiplying the second of each pair by x:

$$c_0 + c_1 x, \quad c_2 + c_3 x, \quad c_4 + c_5 x, \quad 0, \quad \ldots$$

Now, the same operation, but with the multiplier x^2:

$$(c_0 + c_1 x) + (c_2 + c_3 x)x^2, \quad (c_4 + c_5 x) + (0)x^2, \quad 0, \quad \ldots$$

And a final time, with multiplier x^4:

$$[(c_0 + c_1x) + (c_2 + c_3x)x^2] + [(c_4 + c_5x) + (0)x^2]x^4, \quad 0, \quad \ldots$$

We are left with a vector of (active) length 1, whose value is the desired polynomial evaluation. (You can see that the zeros are just a bookkeeping device for taking account of the case where the active subvector has odd length.) The key point is that the combining by pairs is a parallel operation at each stage.

As in subvector scaling, there are about $\log_2 n$ parallel stages. Also as in subvector scaling, our total operations count is only negligibly different from purely scalar code: We do one add and one multiply for each original coefficient c_j. The only extra operations are $\log_2 n$ successive squarings of x; but this comes with the extra benefit of better roundoff properties than the standard scalar coding. In Chapter 23 we use vector reduction to implement our utility function `poly` for polynomial evaluation.

Recursive Doubling: Linear Recurrence Relations

Please don't confuse our use of the word "recurrence" (as in "recurrence relation," "linear recurrence," or equation 22.2.2) with the words "recursion" and "recursive," which both refer to the idea of a subroutine calling itself to obtain an efficient or concise algorithm. There are ample grounds for confusion, because recursive algorithms are in fact a good way of obtaining parallel solutions to linear recurrence relations, as we shall now see!

Consider the general first order linear recurrence relation

$$u_j = a_j + b_{j-1}u_{j-1}, \qquad j = 2, 3, \ldots, n \tag{22.2.5}$$

with initial value $u_1 = a_1$. On a serial machine, we evaluate such a recurrence with a simple do-loop. To parallelize the recurrence, we can employ the powerful general strategy of *recursive doubling*. Write down equation (22.2.5) for $2j$ and for $2j - 1$:

$$u_{2j} = a_{2j} + b_{2j-1}u_{2j-1} \tag{22.2.6}$$

$$u_{2j-1} = a_{2j-1} + b_{2j-2}u_{2j-2} \tag{22.2.7}$$

Substitute equation (22.2.7) in equation (22.2.6) to eliminate u_{2j-1} and get

$$u_{2j} = (a_{2j} + a_{2j-1}b_{2j-1}) + (b_{2j-2}b_{2j-1})u_{2j-2} \tag{22.2.8}$$

This is a new recurrence of the same form as (22.2.5) but over only the even u_j, and hence involving only $n/2$ terms. Clearly we can continue this process recursively, halving the number of terms in the recurrence at each stage, until we are left with a recurrence of length 1 or 2 that we can do explicitly. Each time we finish a subpart of the recursion, we fill in the odd terms in the recurrence, using equation (22.2.7). In practice, it's even easier than it sounds. Turn to Chapter B5 to see a straightforward implementation of this algorithm as the recipe `recur1`.

On a machine with more processors than n, all the arithmetic at each stage of the recursion can be done simultaneously. Since there are of order $\log n$ stages in the

recursion, the execution time is $O(\log n)$. The total number of operations carried out is of order $n + n/2 + n/4 + \cdots = O(n)$, the same as for the obvious serial do-loop.

In the utility routines of Chapter 23, we will use recursive doubling to implement the routines `poly_term`, `cumsum`, and `cumprod`. We *could* use recursive doubling to implement parallel versions of `arth` and `geop` (arithmetic and geometric progressions), and `zroots_unity` (complex nth roots of unity), but these can be done slightly more efficiently by subvector scaling, as discussed above.

Cyclic Reduction: Linear Recurrence Relations

There is a variant of recursive doubling, called *cyclic reduction*, that can be implemented with a straightforward iteration loop, instead of a recursive procedure call. [3] Here we start by writing down the recurrence (22.2.5) for *all* adjacent terms u_j and u_{j-1} (not just the even ones, as before). Eliminating u_{j-1}, just as in equation (22.2.8), gives

$$u_j = (a_j + a_{j-1}b_{j-1}) + (b_{j-2}b_{j-1})u_{j-2} \tag{22.2.9}$$

which is a first order recurrence with new coefficients a'_j and b'_j. Repeating this process gives successive formulas for u_j in terms of u_{j-2}, u_{j-4}, $u_{j-8}\ldots$. The procedure terminates when we reach u_{j-n} (for n a power of 2), which is zero for all j. Thus the last step gives u_j equal to the last set of a'_j's.

Here is a code fragment that implements cyclic reduction by direct iteration. The quantities a'_j are stored in the variable `recur1`.

```
recur1=a
bb=b
j=1
do
   if (j >= n) exit
   recur1(j+1:n)=recur1(j+1:n)+bb(j:n-1)*recur1(1:n-j)
   bb(2*j:n-1)=bb(2*j:n-1)*bb(j:n-j-1)
   j=2*j
enddo
```

In cyclic reduction the length of the vector u_j that is updated at each stage does *not* decrease by a factor of 2 at each stage, but rather only decreases from $\sim n$ to $\sim n/2$ during all $\log_2 n$ stages. Thus the total number of operations carried out is $O(n \log n)$, as opposed to $O(n)$ for recursive doubling. For a serial machine or SSP machine, therefore, cyclic reduction is rarely superior to recursive doubling when the latter can be used. For an MMP machine, however, the issue is less clear cut, because the pattern of communication in cyclic reduction is quite different (and, for some parallel architectures, possibly more favorable) than that of recursive doubling.

Second Order Recurrence Relations

Consider the second order recurrence relation

$$y_j = a_j + b_{j-2}y_{j-1} + c_{j-2}y_{j-2}, \qquad j = 3, 4, \ldots, n \tag{22.2.10}$$

with initial values

$$y_1 = a_1, \qquad y_2 = a_2 \tag{22.2.11}$$

Our labeling of subscripts is designed to make it easy to enter the coefficients in a computer program: You need to supply $a_1, \ldots, a_n$, $b_1, \ldots, b_{n-2}$, and $c_1, \ldots, c_{n-2}$. Rewrite the recurrence relation in the form ([3])

$$\begin{pmatrix} y_j \\ y_{j+1} \end{pmatrix} = \begin{pmatrix} 0 \\ a_{j+1} \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ c_{j-1} & b_{j-1} \end{pmatrix} \begin{pmatrix} y_{j-1} \\ y_j \end{pmatrix}, \qquad j = 2, \ldots, n-1 \tag{22.2.12}$$

that is,

$$\mathbf{u}_j = \mathbf{a}_j + \mathbf{b}_{j-1} \cdot \mathbf{u}_{j-1}, \qquad j = 2, \ldots, n-1 \tag{22.2.13}$$

where

$$\mathbf{u}_j = \begin{pmatrix} y_j \\ y_{j+1} \end{pmatrix}, \quad \mathbf{a}_j = \begin{pmatrix} 0 \\ a_{j+1} \end{pmatrix}, \quad \mathbf{b}_{j-1} = \begin{pmatrix} 0 & 1 \\ c_{j-1} & b_{j-1} \end{pmatrix}, \quad j = 2, \ldots, n-1 \tag{22.2.14}$$

and

$$\mathbf{u}_1 = \mathbf{a}_1 = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \tag{22.2.15}$$

This is a first order recurrence relation for the vectors $\mathbf{u}_j$, and can be solved by the algorithm described above (and implemented in the recipe `recur1`). The only difference is that the multiplications are matrix multiplications with the 2×2 matrices $\mathbf{b}_j$. After the first recursive call, the zeros in **a** and **b** are lost, so we have to write the routine for general two-dimensional vectors and matrices.

Note that this algorithm does not avoid the potential instability problems associated with second order recurrences that are discussed in §5.5 of Volume 1. Also note that the algorithm generalizes in the obvious way to higher-order recurrences: An nth order recurrence can be written as a first order recurrence involving n-dimensional vectors and matrices.

Parallel Solution of Tridiagonal Systems

Closely related to recurrence relations, recursive doubling, and cyclic reduction is the parallel solution of tridiagonal systems. Since Fortran 90 vectors "know their own size," it is most logical to number the components of both the sub- and super-diagonals of the tridiagonal matrix from 1 to $N-1$. Thus equation (2.4.1), here written in the special case of $N = 7$, becomes (blank elements denoting zero),

$$\begin{bmatrix} b_1 & c_1 & & & & & \\ a_1 & b_2 & c_2 & & & & \\ & a_2 & b_3 & c_3 & & & \\ & & a_3 & b_4 & c_4 & & \\ & & & a_4 & b_5 & c_5 & \\ & & & & a_5 & b_6 & c_6 \\ & & & & & a_6 & b_7 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \end{bmatrix} \tag{22.2.16}$$

The basic idea for solving equation (22.2.16) on a parallel computer is to partition the problem into even and odd elements, recurse to solve the former, and

then solve the latter in parallel. Specifically, we first rewrite (22.2.16), by permuting its rows and columns, as

$$\begin{bmatrix} b_1 & & & & c_1 & & \\ & b_3 & & & a_2 & c_3 & \\ & & b_5 & & & a_4 & c_5 \\ & & & b_7 & & & a_6 \\ a_1 & c_2 & & & b_2 & & \\ & a_3 & c_4 & & & b_4 & \\ & & a_5 & c_6 & & & b_6 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_3 \\ u_5 \\ u_7 \\ u_2 \\ u_4 \\ u_6 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_3 \\ r_5 \\ r_7 \\ r_2 \\ r_4 \\ r_6 \end{bmatrix} \tag{22.2.17}$$

Now observe that, by row operations that subtract multiples of the first four rows from each of the last three rows, we can eliminate all nonzero elements in the lower-left quadrant. The price we pay is bringing some new elements into the lower-right quadrant, whose nonzero elements we now call x's, y's, and z's. We call the modified right-hand sides q. The transformed problem is now

$$\begin{bmatrix} b_1 & & & & c_1 & & \\ & b_3 & & & a_2 & c_3 & \\ & & b_5 & & & a_4 & c_5 \\ & & & b_7 & & & a_6 \\ & & & & y_1 & z_1 & \\ & & & & x_1 & y_2 & z_2 \\ & & & & & x_2 & y_3 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_3 \\ u_5 \\ u_7 \\ u_2 \\ u_4 \\ u_6 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_3 \\ r_5 \\ r_7 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \tag{22.2.18}$$

Notice that the last three rows form a new, smaller, tridiagonal problem, which we can solve simply by recursing! Once its solution is known, the first four rows can be solved by a simple, parallelizable, substitution. This algorithm is implemented in `tridag` in Chapter B2.

The above method is essentially cyclic reduction, but in the case of the tridiagonal problem, it does not "unwind" into a simple iteration; on the contrary, a recursive subroutine is required. For discussion of this and related methods for parallelizing tridiagonal systems, and references to the literature, see Hockney and Jesshope [3].

Recursive doubling can also be used to solve tridiagonal systems, the method requiring the parallel solution (as above) of both a first order recurrence and a second order recurrence [3,4]. For tridiagonal systems, however, cyclic reduction is usually more efficient than recursive doubling.

CITED REFERENCES AND FURTHER READING:

Van Loan, C.F. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S.I.A.M.) §1.4.2. [1]

Estrin, G. 1960, quoted in Knuth, D.E. 1981, *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.6.4. [2]

Hockney, R.W., and Jesshope, C.R. 1988, *Parallel Computers 2: Architecture, Programming, and Algorithms* (Bristol and Philadelphia: Adam Hilger), §5.2.4 (cyclic reduction); §5.4.2 (second order recurrences); §5.4 (tridiagonal systems). [3]

Stone, H.S. 1973, *Journal of the ACM*, vol. 20, pp. 27–38; 1975, *ACM Transactions on Mathematical Software*, vol. 1, pp. 289–307. [4]

22.3 Parallel Synthetic Division and Related Algorithms

There are several techniques for parallelization that relate to synthetic division but that can also find application in wider contexts, as we shall see.

Cumulants of a Polynomial

Suppose we have a polynomial

$$P(x) = \sum_{j=0}^{N} c_j x^{N-j} \tag{22.3.1}$$

(Note that, here, the c_j's are indexed from highest degree to lowest, the reverse of the usual convention.) Then we can define the *cumulants* of the polynomial to be partial sums that occur in the polynomial's usual, serial evaluation,

$$\begin{aligned} P_0 &= c_0 \\ P_1 &= c_0 x + c_1 \\ &\cdots \\ P_N &= c_0 x^N + \cdots + c_N = P(x) \end{aligned} \tag{22.3.2}$$

Evidently, the cumulants satisfy a simple, linear first order recurrence relation,

$$P_0 = c_0, \qquad P_j = c_j + x P_{j-1}, \qquad j = 2, \ldots, N \tag{22.3.3}$$

This is slightly simpler than the general first order recurrence, because the value of x does not depend on j. We already know, from §22.2's discussion of recursive doubling, how to parallelize equation (22.3.3) via a recursive subroutine. In Chapter 23, the utility routine `poly_term` will implement just such a procedure. An example of a routine that calls `poly_term` to evaluate a recurrence equivalent to equation (22.3.3) is `eulsum` in Chapter B5.

Notice that while we could use equation (22.3.3), parallelized by recursive doubling, simply to evaluate the polynomial $P(x) = P_N$, this is likely somewhat slower than the alternative technique of vector reduction, also discussed in §22.2, and implemented in the utility function `poly`. Equation (22.3.3) should be saved for cases where the rest of the P_j's (not just P_N) can be put to good use.

Synthetic Division by a Monomial

We now show that evaluation of the cumulants of a polynomial is equivalent to synthetic division of the polynomial by a monomial, also called *deflation* (see §9.5 in Volume 1). To review briefly, and by example, here is a standard tableau from high school algebra for the (long) division of a polynomial $2x^3 - 7x^2 + x + 3$ by the monomial factor $x - 3$.

$$
\begin{array}{r|l}
 & 2x^2 - x - 2 \\
\hline
x-3 & 2x^3 - 7x^2 + x + 3 \\
 & \underline{2x^3 - 6x^2} \\
 & \quad -x^2 + x \\
 & \quad \underline{-x^2 + 3x} \\
 & \qquad -2x + 3 \\
 & \qquad \underline{-2x + 6} \\
 & \qquad\quad -3 \text{ (remainder)}
\end{array}
\tag{22.3.4}
$$

Now, here is the same calculation written as a *synthetic division*, really the same procedure as tableau (22.3.4), but with unnecessary notational baggage omitted (and also a changed sign for the monomial's constant, so that subtractions become additions):

$$
\begin{array}{r|llll}
 & & 6 & -3 & -6 \\
\hline
3 & 2 & -7 & +1 & +3 \\
 & 2 & -1 & -2 & -3
\end{array}
\tag{22.3.5}
$$

If we substitute symbols for the above quantities with the correspondence

$$
\begin{array}{r|llll}
x & c_0 & c_1 & c_2 & c_3 \\
\hline
 & P_0 & P_1 & P_2 & P_3
\end{array}
\tag{22.3.6}
$$

then it is immediately clear that the P_j's in equation (22.3.6) are simply the P_j's of equation (22.3.3); the calculation is thus parallelizable by recursive doubling. In this context, the utility routine `poly_term` is used by the routine `zroots` in Chapter B9.

Repeated Synthetic Division

It is well known from high-school algebra that repeated synthetic division of a polynomial yields, as the remainders that occur, first the value of the polynomial, next the value of its first derivative, and then (up to multiplication by the factorial of an integer) the values of higher derivatives.

If you want to parallelize the calculation of the value of a polynomial and one or two of its derivatives, it is not unreasonable to evaluate equation (22.3.3), parallelized by recursive doubling, two or three times. Our routine `ddpoly` in Chapter B5 is meant for such use, and it does just this, as does the routine `laguer` in Chapter B9.

There are other cases, however, for which you want to perform repeated synthetic division and "go all the way," until only a constant remains. For example, this is the preferred way of "shifting a polynomial," that is, evaluating the coefficients of a polynomial in a variable y that differs from the original variable x by an additive constant. (The recipe `pcshft` has this as its assigned task.) By way of example, consider the polynomial $3x^3 + x^2 + 4x + 7$, and let us perform repeated synthetic division by a general monomial $x - a$. The conventional calculation then proceeds according to the following tableau, reading it in conventional lexical order (left-to-right and top-to-bottom):

$$
\begin{array}{ccccccc}
3 & & 1 & & 4 & & 7 \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
3 & \xrightarrow{a} & 3a+1 & \xrightarrow{a} & 3a^2+a+4 & \xrightarrow{a} & 3a^3+a^2+4a+7 \\
\downarrow & & \downarrow & & \downarrow & & \\
3 & \xrightarrow{a} & 6a+1 & \xrightarrow{a} & 9a^2+2a+4 & & \\
\downarrow & & \downarrow & & & & \\
3 & \xrightarrow{a} & 9a+1 & & & & \\
\downarrow & & & & & & \\
3 & & & & & &
\end{array}
\tag{22.3.7}
$$

Here, each row (after the first) shows a synthetic division or, equivalently, evaluation of the cumulants of the polynomial whose coefficients are the preceding row. The results at the right edge of the rows are the values of the polynomial and (up to integer factorials) its three nonzero derivatives, or (equivalently, without factorials) coefficients of the shifted polynomial.

We could parallelize the calculation of each row of tableau (22.3.7) by recursive doubling. That is a lot of recursion, which incurs a nonnegligible overhead. A much better way of doing the calculation is to deform tableau (22.3.7) into the following equivalent tableau,

$$
\begin{array}{ccccccccc}
3 & \longrightarrow & 3 & & & & & & \\
 & & \downarrow{\scriptstyle a} & \searrow & & & & & \\
1 & \longrightarrow & 3a+1 & & 3 & & & & \\
 & & \downarrow{\scriptstyle a} & \searrow & \downarrow{\scriptstyle a} & \searrow & & & \\
4 & \longrightarrow & 3a^2+a+4 & & 6a+1 & & 3 & & \\
 & & \downarrow{\scriptstyle a} & \searrow & \downarrow{\scriptstyle a} & \searrow & \downarrow{\scriptstyle a} & \searrow & \\
7 & \longrightarrow & 3a^3+a^2+4a+7 & & 9a^2+2a+4 & & 9a+1 & & 3
\end{array}
\tag{22.3.8}
$$

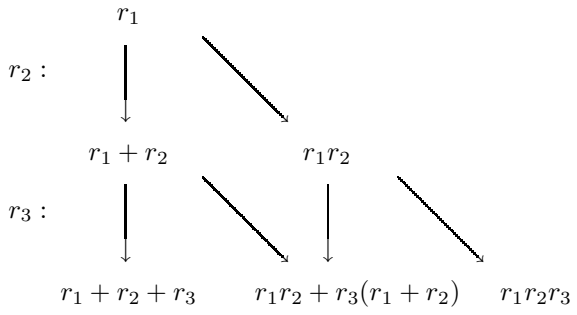
Now each row explicitly depends on only the previous row (and the given first column), so the rows can be calculated in turn by an explicit parallel expression, with no recursive calls needed. An example of coding (22.3.8) in Fortran 90 can be found in the routine `pcshft` in Chapter B5. (It is also possible to eliminate most of the multiplications in (22.3.8), at the expense of a much smaller number of divisions. We have not done this because of the necessity for then treating all possible divisions by zero as special cases. See [1] for details and references.)

Actually, the deformation of (22.3.7) into (22.3.8) is the same trick as was used in Volume 1, p. 167, for evaluating a polynomial and its derivative simultaneously, also generalized in the Fortran 77 implementation of the routine `ddpoly` (Chapter 5). In the Fortran 90 implementation of `ddpoly` (Chapter B5) we *don't* use this trick, but instead use `poly_term`, because, there, we want to parallelize over the length of the polynomial, not over the number of desired derivatives.

Don't confuse the cases of *iterated* synthetic division, discussed here, with the simpler case of doing many simultaneous synthetic divisions. In the latter case, you can simply implement equation (22.3.3) serially, exactly as written, but with each operation being data-parallel across your problem set. (This case occurs in our routine `polcoe` in Chapter B3.)

Polynomial Coefficients from Roots

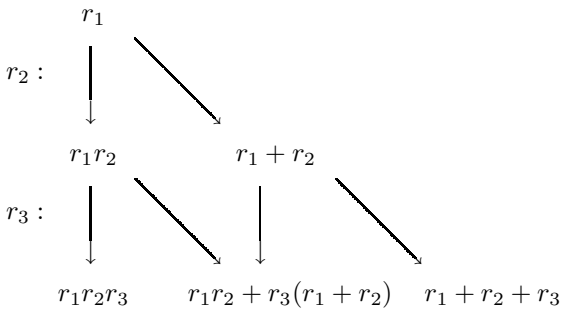
A parallel calculation algorithmically very similar to (22.3.7) or (22.3.8) occurs when we want to find the coefficients of a polynomial $P(x)$ from its roots $r_1, \ldots, r_N$. For this, the tableau is



(22.3.9)

As before, the rows are computed consecutively, from top to bottom. Each row is computed via a single parallel expression. Note that values moving on vertical arrows are simply added in, while values moving on diagonal arrows are multiplied by a new root before adding. Examples of coding (22.3.9) in Fortran 90 can be found in the routines `vander` (Chapter B2) and `polcoe` (Chapter B3).

An equivalent deformation of (22.3.9) is



(22.3.10)

Here the diagonal arrows are simple additions, while the vertical arrows represent multiplication by a root value. Note that the coefficient answers in (22.3.10) come out in the opposite order from (22.3.9). An example of coding (22.3.10) in Fortran 90 can be found in the routine `fixrts` in Chapter B13.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.6.4, p. 470. [1]

22.4 Fast Fourier Transforms

Fast Fourier transforms are beloved by computer scientists, especially those who are interested in parallel algorithms, because the FFT's hierarchical structure generates a complicated, but analyzable, set of requirements for interprocessor communication on MMPs. Thus, almost all books on parallel algorithms (e.g., [1–3]) have a chapter on FFTs.

Unfortunately, the resulting algorithms are highly specific to particular parallel architectures, and therefore of little use to us in writing general purpose code in an architecture-independent parallel language like Fortran 90.

Luckily there is a good alternative that covers almost all cases of both serial and parallel machines. If, for a one-dimensional FFT of size N, one is satisfied with parallelism of order $\sqrt{N}$, then there is a good, general way of achieving a parallel FFT with *quite minimal* interprocessor communication; and the communication required is simply the matrix transpose operation, which Fortran 90 implements as an intrinsic. That is the approach that we discuss in this section, and implement in Chapter B12.

For a machine with M processors, this approach will saturate the processors (the desirable condition where none are idle) when the size of a one-dimensional Fourier transform, N, is large enough: $N > M^2$. Smaller N's will not achieve maximum parallelism. But such N's are in fact so small for one-dimensional problems that they are unlikely to be the rate-determining step in scientific calculations. If they are, it is usually because you are doing many such transforms independently, and you should recover "outer parallelism" by doing them all at once.

For two or more dimensions, the adopted approach will saturate M processors when *each* dimension of the problem is larger than M.

Column- and Row-Parallel FFTs

The basic building block that we assume (and implement in Chapter B12) is a routine for simultaneously taking the FFT of each *row* of a two-dimensional matrix. The method is exactly that of Volume 1's `four1` routine, but with array sections like `data(:,j)` replacing scalars like `data(j)`. Chapter B12's implementation of this is a routine called `fourrow`. If all the data for one column (that is, all the values `data(i,:)`, for some `i`) are local to a single processor, then the parallelism involves no interprocessor communication at all: The independent FFTs simply proceed, data parallel and in lockstep. This is architecture-independent parallelism with a vengeance.

We will also need to take the FFT of each *column* of a two-dimensional matrix. One way to do this is to take the transpose (a Fortran 90 intrinsic that hides a lot of interprocessor communication), then take the FFT of the rows using `fourrow`, then take the transpose again. An alternative method is to recode the `four1` routine with array sections in the other dimension (`data(j,:)`) replacing `four1`'s scalars (`data(j)`). This scheme, in Chapter B12, is a routine called `fourcol`. In this case, good parallelism will be achieved only if the values `data(:,i)`, for some `i`, are local to a single processor. Of course, Fortran 90 does not give the user direct control over how data are distributed over the machine; but extensions such as HPF are designed to give just such control.

On a serial machine, you might think that `fourrow` and `fourcol` should have identical timings (acting on a square matrix, say). The two routines do exactly the same operations, after all. Not so! On modern serial computers, `fourrow` and `fourcol` can have timings that differ by a factor of 2 or more, even when their detailed arithmetic is made identical (by giving to one a data array that is the transpose of the data array given to the other). This effect is due to the multilevel cache architecture of most computer memories, and the fact that serial Fortran always stores matrices by columns (first index changing most rapidly). On our workstations, `fourrow` is significantly faster than `fourcol`, and this is likely the generic behavior. However, we do not exclude the possibility that some machines, and some sizes of matrices, are the other way around.

One-Dimensional FFTs

Turn now to the problem of how to do a single, one-dimensional, FFT. We are given a complex array f of length N, an integer power of 2. The basic idea is to address the input array as if it were a two-dimensional array of size $m \times M$, where m and M are each integer powers of 2. Then the components of f can be addressed as

$$f(Jm + j), \qquad 0 \le j < m,\ 0 \le J < M \tag{22.4.1}$$

where the j index changes more rapidly, the J index more slowly, and parentheses denote Fortran-style subscripts.

Now, suppose we had some magical (parallel) method to compute the discrete Fourier transform

$$F(kM + K) \equiv \sum_{j,J} e^{2\pi i(kM+K)(Jm+j)/(Mm)} f(Jm + j), \qquad 0 \le k < m,\ 0 \le K < M \tag{22.4.2}$$

Then, you can see that the indices k and K would address the desired result (FFT of the original array), with K varying more rapidly.

Starting with equation (22.4.2) it is easy to verify the following identity,

$$F(kM + K) = \sum_j \left[e^{2\pi ijk/m} \left(e^{2\pi ijK/(Mm)} \left[\sum_J e^{2\pi iJK/M} f(Jm + j) \right] \right) \right] \tag{22.4.3}$$

But this, reading it from the innermost operation outward, is just the magical method that we need:

- Reshape the original array to $m \times M$ in Fortran normal order (storage by columns).
- FFT on the second (column) index for all values of the first (row) index, using the routine `fourrow`.
- Multiply each component by a phase factor $\exp[2\pi ijK/(Mm)]$.
- Transpose.

- Again FFT on the second (column) index for all values of the first (row) index, using the routine `fourrow`.
- Reshape the two-dimensional array back into one-dimensional output.

The above scheme uses `fourrow` exclusively, on the assumption that it is faster than its sibling `fourcol`. When that is the case (as we typically find), it is likely that the above method, implemented as `four1` in Chapter B12, is faster, even on scalar machines, than Volume 1's scalar version of `four1` (Chapter 12). The reason, as already mentioned, is that `fourrow`'s parallelism is taking better advantage of cache memory locality.

If `fourrow` is *not* faster than `fourcol` on your machine, then you should instead try the following alternative scheme, using `fourcol` only:

- Reshape the original array to $m \times M$ in Fortran normal order (storage by columns).
- Transpose.
- FFT on the first (row) index for all values of the second (column) index, using the routine `fourcol`.
- Multiply each component by a phase factor $\exp[2\pi ijK/(Mm)]$.
- Transpose.
- Again FFT on the first (row) index for all values of the second (column) index, using the routine `fourcol`.
- Transpose.
- Reshape the two-dimensional array back into one-dimensional output.

In Chapter B12, this scheme is implemented as `four1_alt`. You might wonder why `four1_alt` has three transpose operations, while `four1` had only one. Shouldn't there be a symmetry here? No. Fortran makes the arbitrary, but consistent, choice of storing two-dimensional arrays by columns, and this choice favors `four1` in terms of transposes. Luckily, at least on our serial workstations, `fourrow` (used by `four1`) is faster than `fourcol` (used by `four1_alt`), so it is a double win.

For further discussion and references on the ideas behind `four1` and `four1_alt` see [4], where these algorithms are called the four-step and six-step frameworks, respectively.

CITED REFERENCES AND FURTHER READING:

Fox, G.C., et al. 1988, *Solving Problems on Concurrent Processors*, Volume I (Englewood Cliffs, NJ: Prentice Hall), Chapter 11. [1]

Akl, S.G. 1989, *The Design and Analysis of Parallel Algorithms* (Englewood Cliffs, NJ: Prentice Hall), Chapter 9. [2]

Hockney, R.W., and Jesshope, C.R. 1988, *Parallel Computers 2* (Bristol and Philadelphia: Adam Hilger), §5.5. [3]

Van Loan, C. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S.I.A.M.), §3.3. [4]

22.5 Missing Language Features

A few facilities that are fairly important to parallel programming are missing from the Fortran 90 language standard. On scalar machines this lack is not a

problem, since one can readily program the missing features by using do-loops. On parallel machines, both SSP machines and MMP machines, one must hope that hardware manufacturers provide library routines, callable from Fortran 90, that provide access to the necessary facilities, or use extensions of Fortran 90, such as High Performance Fortran (HPF).

Scatter-with-Combine Functions

Fortran 90 allows the use of *vector subscripts* for so-called *gather* and *scatter* operations. For example, with the setup

```
REAL(SP), DIMENSION(6) :: arr,barr,carr
INTEGER(I4B), DIMENSION(6) :: iarr,jarr
...
iarr = (/ 1,3,5,2,4,6 /)
jarr = (/ 3,2,3,2,1,1 /)
```

Fortran 90 allows both the *one-to-one* gather and the *one-to-many* gather,

```
barr=arr(iarr)
carr=arr(jarr)
```

It also allows the one-to-one scatter,

```
barr(iarr)=carr
```

where the elements of `carr` are "scattered" into `barr` under the direction of the vector subscript `iarr`.

Fortran 90 does *not* allow the *many-to-one* scatter

```
barr(jarr)=carr        ! illegal for this jarr
```

because the repeated values in `jarr` try to assign different components of `carr` to the same location in `barr`. The result would not be deterministic.

Sometimes, however, one would in fact like a many-to-one construction, where the colliding elements get combined by a (commutative and associative) operation, like `+` or `*`, or `max()`. These so-called *scatter-with-combine* functions are readily implemented on serial machines by a do-loop, for example,

```
barr=0.
do j=1,size(carr)
   barr(jarr(j))=barr(jarr(j))+carr(j)
end do
```

Fortran 90 unfortunately provides no means for effecting scatter-with-combine functions in parallel. Luckily, almost all parallel machines do provide such a facility as a library program, as does HPF, where the above facility is called `SUM_SCATTER`. In Chapter 23 we will define utility routines `scatter_add` and `scatter_max` for scatter-with-combine functionalities, but the implementation given in Fortran 90 will be strictly serial, with a do-loop.

Skew Sections

Fortran 90 provides no good, parallel way to access the diagonal elements of a matrix, either to read them or to set them. Do-loops will obviously serve this need on serial machines. In principle, a construction like the following bizarre fragment could also be utilized,

```
REAL(SP), DIMENSION(n,n) :: mat
REAL(SP), DIMENSION(n*n) :: arr
REAL(SP), DIMENSION(n) :: diag
...
arr = reshape(mat,shape(arr))
diag = arr(1:n*n:n+1)
```

which extracts every $(n+1)$st element from a one-dimensional array derived by reshaping the input matrix. However, it is unlikely that any foreseeable parallel compiler will implement the above fragment without a prohibitive amount of unnecessary data movement; and code like the above is also exceedingly slow on all serial machines that we have tried.

In Chapter 23 we will define utility routines `get_diag`, `put_diag`, `diagadd`, `diagmult`, and `unit_matrix` to manipulate diagonal elements, but the implementation given in Fortran 90 will again be strictly serial, with do-loops.

Fortran 95 (see §21.6) will essentially fix Fortran 90's skew sections deficiency. For example, using its `forall` construction, the diagonal elements of an array can be accessed by a statement like

```
forall (j=1:20) diag(j) = arr(j,j)
```

SIMD vs. MIMD

Recall that we use "SIMD" (single-instruction, multiple data) and "data parallel" as interchangeable terms, and that "MIMD" (multiple-instruction, multiple data) is a more general programming model. (See §22.1.)

You should not be too quick to jump to the conclusion that Fortran 90's data parallel or SIMD model is "bad," and that MIMD features, absent in Fortran 90, are therefore "good." On the contrary, Fortran 90's basic data-parallel paradigm has a lot going for it. As we discussed in §22.1, most scientific problems naturally have a "data dimension" across which the time ordering of the calculation is irrelevant. Parallelism across this dimension, which is by nature most often SIMD, frees the mind to think clearly about the computational steps in an algorithm that actually need to be sequential. SIMD code has advantages of clarity and predictability that should not be taken lightly. The general MIMD model of "lots of different things all going on at the same time and communicating data with each other" is a programming and debugging nightmare.

Having said this, we must at the same time admit that a few MIMD features — most notably the ability to go through different logical branches for calculating different data elements in a data-parallel computation — are badly needed in certain programming situations. Fortran 90 is quite weak in this area.

Note that the `where...elsewhere...end where` construction is *not* a MIMD construction. Fortran 90 requires that the `where` clause be executed completely before the `elsewhere` is started. (This allows the results of any calculations in the former

clause to be available for use in the latter.) So, this construction cannot be used to allow two logical branches to be calculated in parallel.

Special functions, where one would like to calculate function values for an array of input quantities, are a particularly compelling example of the need for some MIMD access. Indeed, you will find that Chapter B6 contains a number of intricate, and in a few cases truly bizarre, workarounds, using allowed combinations of `merge`, `where`, and `CONTAINS` (the latter, for separating different logical branches into formally different subprograms).

Fortran 95's `ELEMENTAL` and `PURE` constructions, and to some extent also `forall` (whose body will be able to include `PURE` function calls), will go a long way towards providing exactly the kind of MIMD constructions that are most needed. Once Fortran 95 becomes available and widespread, you can expect to see a new version of this volume, with a much-improved Chapter B6.

Conversely, the number of routines outside of Chapter B6 that can be significantly improved by the use of MIMD features is relatively small; this illustrates the underlying viability of the basic data-parallel SIMD model, even in a future language version with useful MIMD features.