14
High-Performance Computing: Profiling and Tuning

Recall from Chap. 13 that your **problem** is to make a numerically intensive program run faster by better using your high-performance computer. By running the short implementations given in this chapter you may discover how to do that. In the process you will experiment with your computer's memory and experience some of the concerns, techniques, and rewards of high-performance computing (HPC).

In HPC, you generally modify or "tune" your program to take advantage of a computer's architecture (discussed in Chap. 13). Often the real problem is to determine which parts of your program get used the most and to decide whether they would run significantly faster if you modified them to take advantage of a computer's architecture. In this chapter we concentrate on how you determine the most numerically intensive parts of your program, and how specific hardware features affect them.

Be warned, there is a negative side to high-performance computing. Not only does it take hard work and time to tune a program, but as you optimize a program for a specific piece of hardware and its special software features, you make your program less portable and probably less readable. One school of thought says it is the compiler's, and not the scientist's, job to worry about computer architecture, and it is old-fashioned for you to tune your programs. Yet many computational scientists who run large and complex programs on a variety of machines frequently obtain a 300–500% speedup when they tune their programs for the CPU and memory architecture of a particular machine. You, of course, must decide whether it is worth the effort for the problem at hand; for a program run only once, it is probably not, for an essential tool used regularly, it probably is.

14.1
Rules for Optimization (Theory)

The type of optimization often associated with *High-Performance* or *Numerically Intensive* computing is one in which sections of a program are rewritten and reorganized in order to increase the program's speed. The overall value

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

ISBN: 978-3-527-40626-5

of doing this, especially as computers have become so fast and so available, is often a subject of controversy between computer scientists and computational scientists. Both camps would agree that using the optimization options of the compilers is a good idea. However, computational scientists tend to run large codes with large amounts of data in order to solve real-world problems, and often believe that you cannot rely on the compiler to do all the optimization, especially when you end up with time on your hands waiting for the computer to finish executing your program.

14.1.1 Programming for Virtual Memory (Method)

While paging makes little appear big, you pay a price because your program's run time increases with each page fault. If your program does not fit into RAM all at once, it will run significantly slower. If virtual memory is shared among multiple programs that run simultaneously, they all cannot have the entire RAM at once, and so there will be memory access *conflicts*, in which case the performance of all programs suffer.

The basic rules for programming for virtual memory are as follows.

1. Do not waste your time worry about reducing the amount of memory used (the *working set size*) unless your program is large. In that case, take a global view of your entire program and optimize those parts that contain the largest arrays.
2. Avoid page faults by organizing your programs to successively perform its calculations on subsets of data, each fitting completely into RAM.
3. Avoid simultaneous calculations in the same program to avoid competition for memory and consequent page faults. Complete each major calculation before starting another.
4. Group data elements close together in memory blocks if they are going to be used together in calculations.

14.1.2 Optimizing Programs; Java vs. Fortran/C

Many of the optimization techniques developed for Fortran and C are also relevant for Java applications. Yet while Java is a good language for scientific programming and is the most universal and portable of languages, at present Java code runs slower than Fortran or C code, and does not work well if you use MPI for parallel computing (see Chap. 21). In part, this is a consequence of the Fortran and C compilers having been around longer and thereby having been better refined to get the most out of a computer's hardware, and, in part,

this is also a consequence of Java being designed for portability and not speed. Since modern computers are so fast, whether a program takes 1 s or 3 s usually does not matter much, especially in comparison to the hours or days of *your* time that it might take to modify a program for different computers. However, you may want to convert the code to C (whose command structure is very similar to Java) if you are running a computation that takes hours or days to complete and will be doing it many times.

Especially when asked to, Fortran and C compilers will look at your entire code as a single entity and rewrite it for you so that it runs faster. (The rewriting is at a fairly basic level, so there is not much use in your studying the compiler's output as a way of improving your programming skills.) In particular, Fortran and C compilers are very careful with the accessing of arrays in memory. They also are careful with keeping the cache lines full so as not to keep the CPU waiting with nothing to do.

There is no fundamental reason why a program written in Java cannot be compiled to produce an equally efficient code, and indeed such compilers are being developed and becoming available. However, such code is optimized for a particular computer architecture and so are not portable. In contrast, the class file produced by Java is designed to be interpreted or recompiled by the *Java Virtual Machine* (just another program). When you change from a Unix to a Windows computer, to illustrate, the Java Virtual Machine program changes, but the byte code that runs via the Virtual Machine stays the same. This is the essence of Java's portability.

In order to improve the performance of Java, many computers and browsers now run a *Just-In-Time* (JIT) Java compiler. If a JIT is present, the Java Virtual Machine feeds your byte code `Prog.class` to the JIT so that it can be recompiled into native code explicitly tailored to the machine on which you are running. Although there is an extra step involved here, the total time it takes to run your program is usually 10–30 times faster with the JIT, as compared to line-by-line interpretation. Because the JIT is an integral part of the Java Virtual Machine on each operating system, this usually happens automatically.

In the exercises below you will investigate techniques to optimize both a Fortran and a Java program. In the process you will compare the speeds of Fortran *versus* Java for the same computation. If you run your Java code on a variety of machines (easy to do with Java), you should also be able to compare the speed of one computer to another. *Note* that you can do this exercise even if you do not know Fortran.

14.1.3 Good and Bad Virtual Memory Use (Experiment)

To see the effect of using virtual memory, run these simple pseudocode examples on your computer. Use a command such as `time` to measure the time being used for each example. These examples call functions `force12` and `force21`. You should write these functions and make them have significant memory requirements for both local and global variables.

BAD Program, Too Simultaneous

```
for j = 1, n; {
  for i = 1, n; {
    f12(i,j) = force12( pion(i), pion(j) )          // Fill f12
    f21(i,j) = force21( pion(i), pion(j) )          // Fill f21
    ftot = f12(i,j) + f21(i,j)                      // Fill ftot
  }
}
```

You see that each iteration of the `for` loop requires the data and code for all the functions as well as access to all elements of the matrices and arrays. The working set size of this calculation is the sum of the sizes of the arrays `f12(N,N)`, `f21(N,N)`, and `pion(N)` plus the sums of the sizes of the functions `force12` and `force21`.

A better way to perform the same calculation is to break the calculation into separate components:

GOOD Program, Separate Loops

```
for j = 1, n; {
  for i = 1, n; {
    f12(i,j) = force12( pion(i), pion(j) )          // Fill just f12
  }
} for j = 1, n; {
  for i = 1, n;  {                                    // Second nest
    f21(i,j) = force21( pion(i), pion(j) )          // Fill just f21
  }
} for j = 1, n; {
  for i = 1, n;  {                                    // Third nest
    ftot = f12(i,j) + f21(i,j)                      // Compute ftot
  }
}
```

Here the separate calculations are independent and the *working set size*, that is, the amount of memory used, is reduced. However, you do pay the additional overhead costs associated with creating extra `for` loops. Because the working set size of the first `for` loop is the sum of the sizes of the arrays `f12(N,N)` and `pion(N)`, and of the function `force12`, we have approximately half the previous size. The size of the last `for` loop is the sum of the sizes for the two

arrays. The working set size of the entire program is the larger of the working set sizes for the different `for` loops.

As an example of the need to group data elements close together in memory or Common blocks if they are going to be used together in calculations, consider the following code:

BAD Program, Discontinuous Memory

```
Common zed, ylt(9), part(9), zpart1(50000), zpart2(50000), med2(9)
for j = 1, n; {
  ylt(j) = zed * part(j) / med2(9)          // Discontinuous variables
```

Here the variables `zed`, `ylt`, and `part` are used in the same calculations and are adjacent in memory because the programmer grouped them together in `Common` (global variables). Later, when the programmer realized that the array `med2` was needed, it got tacked onto the end of Common. All the data comprising the variables `zed`, `ylt`, and `part` fit into one page, but the `med2` variable is on a different page because the large array `zpart2(50000)` separates it from the other variables. In fact, the system may be forced to make the entire 4-kB page available in order to fetch the 72 bytes of data in `med2`. While it is difficult for the Fortran or C programmer to assure the placement of variables within page boundaries, you will improve your chances by grouping data elements together:

GOOD Program, Continuous Memory

```
Common zed, ylt(9), part(9), med2(9), zpart1(50000), zpart2(50000)
for j = 1, n; {
  ylt(j) = zed * part(j) / med2(J)          // Continuous variables
```

14.1.4 Experimental Effects of Hardware on Performance

*We now return to our **problem** of making a numerically intensive program run faster. In this section you conduct an experiment in which you run a complete program in several languages, and on as many computers as you can get your hands on. In this way you explore how a computer's architecture and software affect a program's performance.*

Even if *you* do not know (or care) what is going on inside of a program, some optimizing compilers are smart and caring enough to figure it out for you, and then go about rewriting your program for improved performance. You control how completely the compiler does this when you add on *optimization options* to the compile command:

```
> f90 -O tune.f90
```

Here the `-O` turns on optimization (`O` is the capital letter "oh," not zero). The actual optimization that gets turned on often differs from compiler to compiler. Fortran and C compilers often have a bevy of such options and directives that lets you truly customize the resulting compiled code. Sometimes optimization options do make the code run faster, sometimes not, and sometimes the faster-running code gives the wrong answers (but does so quickly).

Because computational scientists often spend a good fraction of their time running compiled codes, the compiler options tend to get quite specialized. As a case in point, most compilers provide a number of levels of optimization for the compiler to attempt (their are no guarantees with these things). Although the speedup obtained depends upon the details of the program, higher levels may give greater speedup, as well as a concordant greater risk of being wrong. Some **Forte/Sun** Fortran compiler options, which are rather standard, include the following:

`-O`	Use default optimization level (`-O3`)
`-O1`	Provide minimum statement-level optimizations
`-O2`	Enable basic block-level optimizations
`-O3`	Add loop unrolling and global optimizations
`-O4`	Add automatic inlining of routines from same source file
`-O5`	Attempt aggressive optimizations (with profile feedback)

For the **Visual Fortran (Compaq, Intell)** compiler under windows, options are entered as `/optimize` and for optimization are as follows:

`/optimize:0`	Disable most optimizations
`/optimize:1`	Local optimizations in source program unit
`/optimize:2`	Global optimization, includes `/optimize:1`
`/optimize:3`	Additional global optimizations; speed at cost of code size: loop unrolling, instruction scheduling, branch code replication, padding arrays for cache
`/optimize:4`	Interprocedure analysis, inlining small procedures
`/optimize:5`	activates loop transformation optimizations

The **gnu compilers** `gcc, g77, g90` accept `-O` options, as well as specialized ones that include the following:

`-malign-double`	align doubles on 64-bit boundaries
`-ffloat-store`	when using IEEE 854 extended precision
`-fforce-mem, -fforce-addr`	improved loop optimization
`-fno-inline`	do not compile statement functions inline
`-ffast-math`	try non-IEEE handling of floats
`-funsafe-math-optimizations`	speeds up but incorrect results possible
`-fno-trapping-math`	assume no floating point traps generated
`-fstrength-reduce`	makes some loops faster
`-frerun-cse-after-loop`	
`-fexpensive-optimizations`	
`-fdelayed-branch`	
`-fschedule-insns`	
`-fschedule-insns2`	
`-fcaller-saves`	
`-funroll-loops`	unroll iterative do loops
`-funroll-all-loops`	unroll `DO WHILE` loops

14.1.5 Java versus Fortran/C

The various `tune` programs solve the matrix eigenvalue problem

$$\mathbf{H}\mathbf{c} = E\mathbf{c} \tag{14.1}$$

for the eigenvalues E and eigenvectors $\mathbf{c}$ of a Hamiltonian matrix $\mathbf{H}$. Here the individual Hamiltonian matrix elements are assigned the values

$$H_{i,j} = \begin{cases} i, & \text{for } i = j, \\ 0.3^{|i-j|}, & \text{for } i \neq j, \end{cases} = \begin{bmatrix} 1 & 0.3 & 0.09 & 0.027 & \dots \\ 0.3 & 2 & 0.3 & 0.9 & \dots \\ 0.09 & 0.3 & 3 & 0.3 & \dots \\ \ddots & & & & \end{bmatrix} \tag{14.2}$$

Because the Hamiltonian is almost diagonal, the eigenvalues should be close to the values of the diagonal elements and the eigenvectors should be close to N-dimensional unit vectors. For the present problem, the H matrix has dimension $N \times N \simeq 2000 \times 2000 = 4{,}000{,}000$, which means that matrix manipulations should take enough time for you to see the effects of optimization. If your computer has a large supply of central memory, you may need to make the matrix even larger to see what happens when a matrix does not all fit into RAM.

The solution to (14.1) is found via a variation of the *power* or *Davidson method*. We start off with an arbitrary first guess for the eigenvector $\mathbf{c}$, which

we represent as the unit, column vector:[1]

$$\mathbf{c}_0 \simeq \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{14.3}$$

Because the **H** matrix is nearly diagonal with the diagonal element that increase as we move along the diagonal, this guess should be close to the eigenvector with the smallest eigenvalue. We next calculate the energy corresponding to this eigenvector,

$$E = \frac{\mathbf{c}_0^\dagger \mathbf{H} \mathbf{c_0}}{\mathbf{c}_0^\dagger \mathbf{c}_0} \tag{14.4}$$

where $\mathbf{c}_0^\dagger$ is the row vector adjoint of $\mathbf{c}_0$. The heart of the algorithm is the guess that an improved eigenvector has the kth component

$$\mathbf{c}_1|_k \;\simeq\; \mathbf{c}_0|_k + \frac{[\mathbf{H} - E\mathbf{I}]\mathbf{c}_0|_k}{E - H_{k,k}} \tag{14.5}$$

where k ranges over the length of the eigenvector. If repeated, this method converges to the eigenvector with the smallest eigenvalue. It will be the smallest eigenvalue since it gets the largest weight (smallest denominator) in (14.5) each time. For the present case, six places of precision in the eigenvalue is usually obtained after 11 iterations.

- Vary the variable `err` in `tune` that controls precision and note how it affects the number of iterations required.
- Try some variations on the initial guess for the eigenvector (14.5) and see if you can get the algorithm to converge to some of the other eigenvalues.
- Keep a table of your execution times versus technique.
- Compile and execute `tune.f90`, and record the run time. On Unix systems, the compiled program will be placed in the file `a.out`. From a Unix shell, the compilation, timing, and execution can all be done with the commands:

```
> f90 tune.f90                    Fortran compilation
> cc -lm tune.c                   C compilation, gcc also likely
> time a.out                      Execution
```

Here the compiled Fortran program is given the (default) name `a.out` and the `time` command gives you the execution (`user`) time and `system` time in seconds to execute `a.out`.

[1] Note that the codes refer to the eigenvector c_0 as `coef`.

- As we indicated in Section 14.1.4, you can ask the compiler to produce a version of your program optimized for speed by including the appropriate compiler option:

```
> f90 -O tune.f90
```

Execute and time the optimized code, checking that it still gives the same answer, and note any speed up in your journal.

- Try out optimization options up to the highest levels and note the run time and accuracy obtained. Usually `-O3` is pretty good, especially for as simple a program as `tune` with only a main method. With only one program unit, we would not expect `-O4` or `-O5` to be an improvement over `-O3`. However, we do expect `-O3`, with its loop unrolling, to be an improvement over `-O2`.

- The program `tune4` does some *loop unrolling* (we will explore that soon). To see the best we can do with Fortran, record the time for the most optimized version of `tune4.f90`.

- The program `Tune.java` is the Java equivalent of the Fortran program `tune.f90`. It is given in Listing 14.1.

- To find an idea of what `Tune.java` does (and give you a feel for how hard life is for the poor computer), assume `ldim =2` and work through one iteration of `Tune` *by hand*. Assume that the iteration loop has converged, follow the code to completion, and write down the values assigned to the variables.

- Compile and execute `Tune.java`. You do not have to issue the `time` command since we built a timer into the Java program (however there is no harm in trying it). Check that you still get the same answer as you did with Fortran, and note how much longer it takes with Java.

- Try the -O option with the Java compiler and note if the speed changes (since this just inlines methods, it should not affect our one-method program).

- You might be surprised how much slower is Java than Fortran and that the Java optimizer does not seem to do much good. To see what the actual Java byte code does, invoke the Java profiler with the command

```
> javap -c Tune
```

This should produce a file `java.prof` for you to look at with an editor. Look at it and see if you agree with us that scientists have better things to do with their time than understand such files!

- We now want to perform a little experiment in which we see what happens to performance as we fill up the computer's memory. In order for this experiment to be reliable, it is best for you to *not* to be sharing the computer with any other users. On Unix systems, the `who -a` command will show you the other users (we leave it up to you to figure out how to negotiate with them).

Listing 14.1: `Tune.java` is meant to be a numerically intensive enough so as to show the results of various types of optimizations. The program solves the eigenvalue problem iteratively for a nearly diagonal Hamiltonian matrix using a variation of the power method.

```
// Tune.java: eigenvalue solution for performace tuning

public class Tune {

  public static void main(String[] argv) {

    final int Ldim = 2051;
    int i, j, iter = 0;
    double [][] ham = new double [Ldim] [Ldim];
    double [] coef = new double [Ldim];
    double [] sigma = new double [Ldim];
    double err, ener, ovlp, step = 0., time;

    time = System.currentTimeMillis();              // Initialize time
                                                 // Init matrix & vector
      for ( i = 1;  i <= Ldim-1;  i++ )  {
        for ( j=1;  j <= Ldim-1;  j++ ) {
          if (Math.abs(j-i) >10)  ham[j][i] = 0. ;
          else  ham[j][i] = Math.pow(0.3, Math.abs(j-i));
        }
        ham[i][i] = i ;
        coef[i] = 0.;
      }
      coef[1] = 1.;
      err = 1.;
      iter = 0 ;
                                                       // Start iteration
      while (iter  < 15 && err > 1.e-6)  {
        iter = iter + 1;
        ener = 0. ;
        ovlp = 0.;
                                                 // Compute E & normalize
        for ( i= 1;  i <= Ldim-1;  i++ )  {
          ovlp = ovlp + coef[i]*coef[i] ;
          sigma[i] = 0. ;
          for (j= 1; j <= Ldim-1;  j++)
                sigma[i] = sigma[i]+coef[j]*ham[j][i];
          ener = ener + coef[i]*sigma[i] ;
        }
        ener = ener/ovlp;
        for (  i = 1;  i <= Ldim-1;  i++ )  {
          coef[i] = coef[i]/Math.sqrt(ovlp) ;
          sigma[i] = sigma[i]/Math.sqrt(ovlp) ;
        }
        err = 0.;
                                                                 // Update
        for ( i = 2;  i <= Ldim-1;  i++ )  {
        step = (sigma[i] - ener*coef[i])/(ener-ham[i][i]) ;
         coef[i] = coef[i] + step  ;
         err = err +  step*step ;        }
        err = Math.sqrt(err) ;
        System.out.println
```

```
            ("iter, ener, err " + iter + ",  " + ener + ",  " + err);
        }
        time = (System.currentTimeMillis() - time)/1000;    // Elapsed t
        System.out.println("time = " + time + "s");
    }
}
```

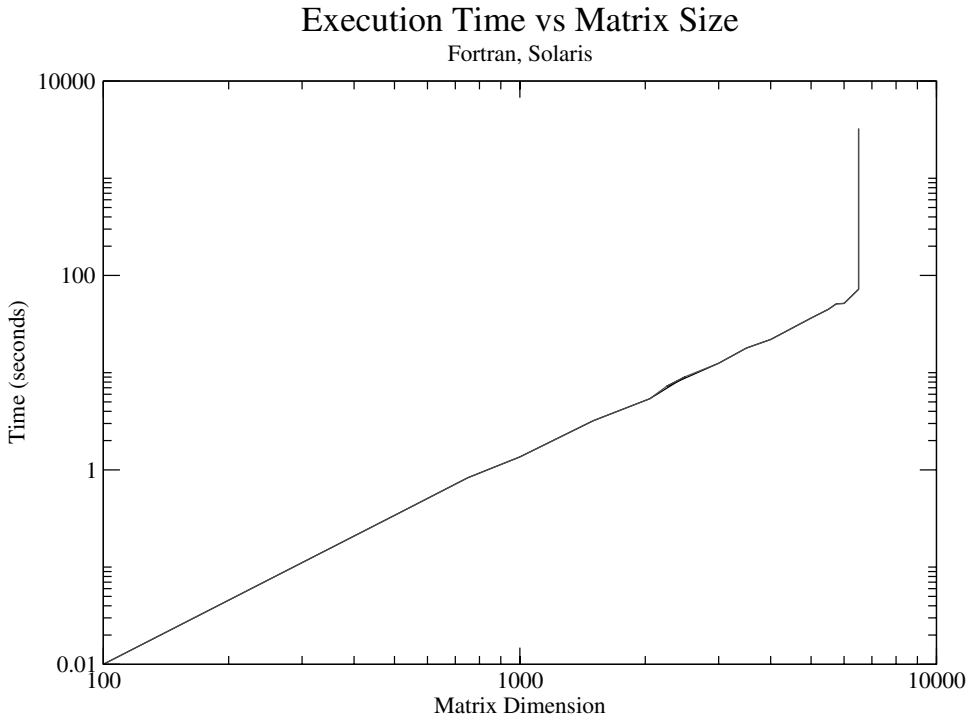

Fig. 14.1 Running time versus matrix size for eigenvalue search using `tune.f90`. Note Fortran's execution time is proportional to the matrix size squared.

- To obtain some idea of what aspect of our little program is making it so slow, compile and run `Tune.java` for the series of matrix sizes `ldim` = 10, 100, 250, 500, 750, 1025, 2500, and 3000. You may get an error message that Java is out of memory at 3000. This is because you have not turned on the use of virtual memory. In Java, the memory allocation pool for your program is called the *heap* and it is controlled by the -Xms and -Xmx options to the Java interpreter java:

`-Xms256m`	Set initial heap size to 256 Mbytes
`-Xmx512m`	Set maximum heap size to 512 Mbytes

- Make a graph of the run time versus matrix size. It should be similar to Figs. 14.1 and 14.2. However, if there are more than one user on your computer

while you run, you may get erratic results. You will note that as our matrix gets larger than $\sim 1000 \times 1000$ in size, the curve has a sharp increase in slope with execution time, in our case increasing like the *third* power of the dimension. Since the number of elements to compute increases like the *second* power of the dimension, something else is happening here. It is a good guess that the additional slowdown is due to page faults in accessing memory. In particular, accessing 2D arrays, with their elements scattered all through memory, can be very slow.

- Repeat the previous experiment with `tune.f90` that gauges the effect of increasing the `ham` matrix size, only now do it for `ldim` = 10, 100, 250, 500, 1025, 3000, 4000, 6000,. . ., . You should get a graph like ours. Although our implementation of Fortran has automatic virtual memory, its use will be exceedingly slow, especially for this problem (possibly 50-fold increase in time!). So if you submit your program and you get nothing on the screen (though you can hear the disk spin or see it flash busy), then you are probably in the virtual memory regime. If you can, let the program run for one or two iterations, kill it, and then scale your run time to the time it would have taken for a full computation.
- To test our hypothesis that the access of the elements in our 2D array `ham [i][j]` is slowing down the program, we have modified `Tune.java` into `Tune4.java` (Listing 14.2).

Listing 14.2: `Tune4.java` is similar to `Tune.java` in Listing 14.1, but does some loop unrolling by explicitly writing out two steps of a `for` loop (which is why the loop can proceed in steps of two). This results in better memory access and consequently faster execution.

```
// Tune4.java: matrix algebra program, basic optimization

public class Tune4  {

    public static void main(String[] argv)  {

      final int Ldim = 2051;
      int i, j, iter = 0;
      double [][] ham = new double [Ldim] [Ldim];
      double []   diag = new double [Ldim];
      double [] coef  = new double [Ldim];
      double [] sigma = new double [Ldim];
      double err, ener, ovlp, ovlp1, ovlp2;
      double step = 0., fact, time, t, t1, t2, u;

      time = System.currentTimeMillis();          // Store initial time
                                                  // Set up Hamiltonian
      for ( i = 1;  i <= Ldim-1;  i++ )   {
        for ( j = 1;  j <= Ldim-1;  j++ )
          if (Math.abs(j-i) >10) ham[j][i] = 0. ;
          else  ham[j][i] = Math.pow(0.3, Math.abs(j-i));
        }
                                              // Iterate toward solution
```

```
      for ( i=1;  i < Ldim-1;  i++ ) {
        ham[i][i] = i ;
        coef[i] = 0.;
        diag[i] = ham [i][i];
      }
      coef[1] = 1.;
      err = 1.;
      iter = 0 ;
      while (iter  < 15 && err > 1.e-6)  {
        iter = iter + 1;
                                  // compute current energy & normalize
        ener = 0. ;
        ovlp1 = 0.;
        ovlp2 = 0.;
        for ( i= 1;  i <= Ldim-2;  i = i + 2  ) {
          ovlp1 = ovlp1 + coef[i]*coef[i] ;
          ovlp2 = ovlp2 + coef[i+1]*coef[i+1] ;
          t1 = 0.;
          t2 = 0.;
          for ( j=1;  j <= Ldim-1;  j++ )  {
            t1 = t1 + coef[j]*ham[j][i];
            t2 = t2 + coef[j]*ham[j][i+1];
          }
          sigma[i] = t1;
          sigma[i + 1] = t2;
          ener = ener + coef[i]*t1 + coef[i+1]*t2 ;
        }
        ovlp = ovlp1 + ovlp2 ;
        ener = ener/ovlp;
        fact = 1./Math.sqrt(ovlp);
        coef[1] = fact*coef[1];
        err = 0.;
                                                  // Update & error norm
        for ( i = 2;  i <= Ldim-1;  i++ ) {
          t = fact*coef[i];
          u = fact*sigma[i]-ener*t;
          step = u/(ener-diag[i]) ;
          coef[i] = t + step  ;
          err = err +  step*step  ;
        }
        err = Math.sqrt(err) ;
        System.out.println
                  ("iter, ener, err "+iter+", " + ener + ", " + err);
      }
      time = (System.currentTimeMillis() - time)/1000;
      System.out.println("time = " + time + "s");      // Elapsed time
  }
}
```

- Look at `Tune4.java` and note where the nested `for` loop over `i` and `j` now takes step of $\Delta i = 2$ rather the unit steps in `Tune.java`. If things work as expected, the better memory access of `Tune4.java` should cut the runtime nearly in half. Compile and execute `Tune4.java`. Record your answer in your table.

• In order to cut the number of calls to the 2D array in half, we employed a technique known as *loop unrolling* in which we explicitly wrote out some of the lines of code which, otherwise, would be executed implicitly as the `for` loop went through all the values for its counters. This is not as clear a piece of code as before, but, evidently, it permits the compiler to produce a faster executable. To check that `Tune` and `Tune4` actually do the same thing, assume `ldim =4` and run through one iteration of `Tune4.java` *by hand*. Hand in your manual trial.

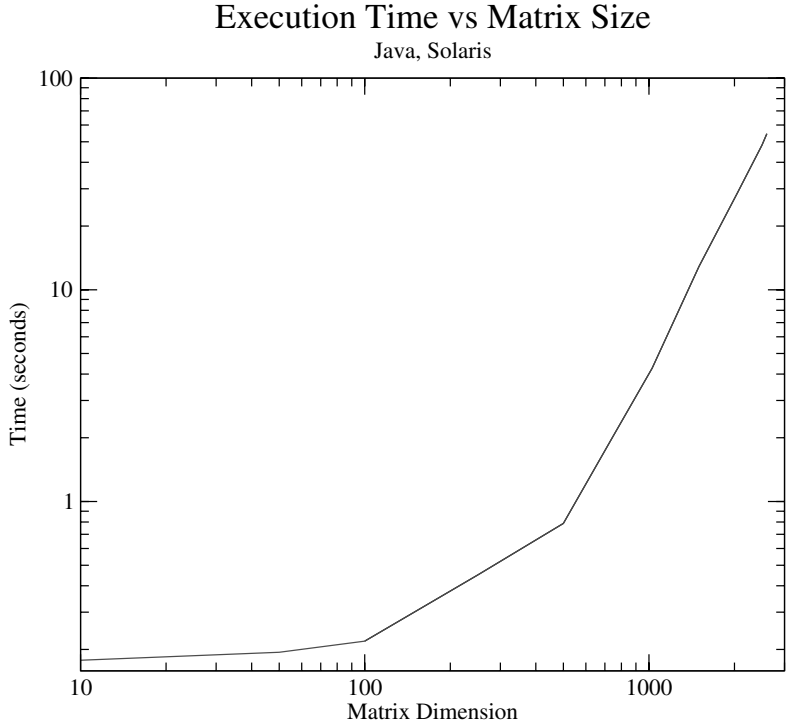

Fig. 14.2 Running time versus matrix size for eigenvalue search using `Tune.java`. Note how much faster is Fortran (Fig. 14.1), and that for large sizes, the Java execution time varies as the third power of the matrix size. The extra power of size (compared to Fortran) arises from the time spent on reading the matrix elements into and out of memory, something that the Fortran program has been optimized to do rather well.

14.2
Programming for Data Cache (Method)

Data caches are small, very fast memory used as temporary storage between the ultrafast CPU registers and the fast main memory. They have grown in importance as high-performance computers have become more prevalent. On systems that use a data cache, this may well be the single most important programming consideration; continually referencing data that are not in the cache (*cache misses*) may lead to an order-of-magnitude increase in CPU time.

As indicated in Figs. 13.2 and 14.3, the data cache holds a copy of some of the data in memory. The basics are the same for all caches but the sizes are manufacturer dependent. When the CPU tries to address a memory location, the *cache manager* checks to see if the data are in cache. If they are not, the manager reads the data from memory into cache and then the CPU deals with the data directly in cache. The cache manager's view of RAM is shown in Fig. 14.3.

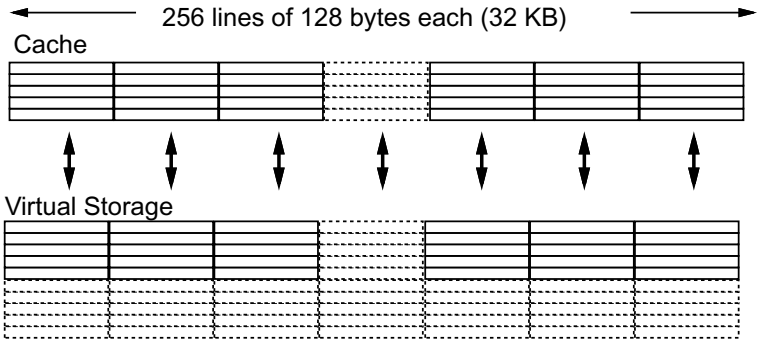

Fig. 14.3 The cache manager's view of RAM. Each 128-byte cache line is read into one of four lines in cache.

When considering how some matrix operation uses memory, it is important to consider the *stride* of that operation, that is, the number of array elements that get stepped through as an operation repeats. For instance, summing the diagonal elements of a matrix to form the trace

$$\mathrm{Tr}A = \sum_{i=1}^{N} a(i,i) \tag{14.6}$$

involves a large stride because the diagonal elements are stored far apart for large N. However, the sum

$$c(i) = x(i) + x(i+1) \tag{14.7}$$

has stride 1 because adjacent elements of `x` are involved. The basic rule in programming for cache is

- Keep the stride low, preferably at 1, which in practice means.
- Vary the leftmost index first on Fortran arrays.
- Vary the rightmost index first on Java and C arrays.

14.2.1 Exercise 1: Cache Misses

We have said a number of times that your program will be slowed down if the data it needs is in virtual memory and not in RAM. Likewise, your program will also be slowed down if the data required by the CPU is not cache. For high-performance computing, you should write programs that keep as much of the data being processed as possible in cache. To do this you should recall that Fortran matrices are stored in successive memory locations with the row index varying most rapidly (column-major order), while Java and C matrices are stored in successive memory locations with the column index varying most rapidly (row-major order). While it is difficult to isolate the effects of cache from other elements of the computer's architecture, you should now estimate its importance by comparing the time it takes to step through matrix elements row by row, to the time it takes to step through matrix elements column by column.

By actually running on machines available to you, check that these two simple codes with the same number of arithmetic operations will take significantly different times to run because one of them must make large jumps through memory with the memory locations addressed not yet read into cache:

Sequential Column and Row References

```
for j = 1, 9999; {
   x(j) = m(1,j)                          // Sequential column reference
```

```
for j = 1, 9999; {
    x(j) = m(j,1)                          // Sequential row reference
```

14.2.2 Exercise 2: Cache Flow

Test the importance of cache flow on your machine by comparing the time it takes to run these two simple programs. Run for increasing column size `idim` and compare the times for loop A versus those for loop B. A computer with very small caches may be most sensitive to stride.

GOOD f90, BAD Java/C Program; Minimum, Maximum Stride

```
Dimension Vec(idim, jdim)                                    // Loop A

for j = 1, jdim; {
  for i = 1, idim; {
    Ans = Ans + Vec(i, j) * Vec(i, j)          // Stride 1 fetch (f90)
  }
}
```

BAD f90, GOOD Java/C Program; Maximum, Minimum Stride

```
Dimension Vec(idim, jdim)                                    // Loop B

for i = 1, idim; {
  for j = 1, jdim; {
    Ans = Ans + Vec(i, j) * Vec(i, j)       // Stride jdim fetch (f90)
  }
}
```

Loop A steps through the matrix `Vec` in column order. Loop B steps in row order. By changing the size of the columns (the rightmost index for Fortran), we change the size of the step (*stride*) we take through memory in Fortran. Both loops take us through all elements of the matrix, but the stride is different. By increasing the stride in any language, we use fewer elements already present in cache, require additional swapping and loading of cache, and thereby slow down the whole process.

14.2.3 Exercise 3: Large Matrix Multiplication

As you increase the dimension of the arrays in your program, memory use increases geometrically, and at some point you should be concerned about efficient memory use. The penultimate example of memory usage is large matrix multiplication:

$$[C] = [A] \times [B] \tag{14.8}$$

This involves all the concerns with the different kinds of memory. The natural way to code (14.8) follows from the definition of matrix multiplication:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \times b_{kj} \tag{14.9}$$

The sum is over a row of A times a column of B.

Try out these two codes on your computer. In Fortran, the first code has B with stride 1, but C with stride N. This is cured in the second code by performing the initialization in another loop. In Java and C, the problems are

reversed. On one of our machines, we found a factor of 100 difference in CPU times even though the number of operations is the same!

BAD f90, GOOD Java/C Program; Maximum, Minimum Stride

```
for i = 1, N; {                                          // Row
  for j = 1, N; {                                       // Column
    c(i, j) = 0.                                       // Initialize
      for k = 1, N; {
        c(i, j) = c(i, j) + a(i, k) * b(k, j)          // Accumulate sum
      }
  }
}
```

GOOD f90, BAD Java/C Program; Minimum, Maximum Stride

```
for j = 1, N; {                                          // Initialization
    for  i = 1, N; {
      c(i,j) = 0.0
    }
    for k = 1, N; {
      for  i = 1, N; {
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      }
    }
}
```