Achieving High Performance in Numerical Computations on RISC Workstations and Parallel Systems

CONF-970849--

RECEIVED
FRR 02 1998
OSTI

Los Alamos National Laboratory

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. The Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
Achieving high performance in numerical computations on RISC workstations and parallel systems

Stefan Goedecker,
Max-Planck Institute for Solid State Research,
Stuttgart, Germany
goedeck@prr.mpi-stuttgart.mpg.de

Adolfy Hoisie,
Computer, Information and Communications Division,
Los Alamos National Laboratory, USA
hoisie@lanl.gov

August 20, 1997

1 Prolog

The nominal peak speeds of both serial and parallel computers is raising rapidly. At the same time however it is becoming increasingly difficult to get out a significant fraction of this high peak speed from modern computer architectures. In this tutorial we will give the scientists and engineers involved in numerically demanding calculations and simulations the necessary basic knowledge to write reasonably efficient programs. The basic principles are rather simple and the possible rewards large. Writing a program by taking into account optimization techniques related to the computer architecture can significantly speedup your program, often by factors of 10-100. As such, optimizing a program can for instance be a much better solution than buying a faster computer! If a few basic optimization principles are applied during program development, the additional time needed for obtaining an efficient program is practically negligible. In-depth optimization is usually only needed for a few subroutines or kernels and the effort involved is therefore also acceptable.
## Contents

1 Prolog ........................... 1

2 Some basic efficiency guidelines 3
2.1 Select the best algorithm ........................................ 3
2.2 Use off-the-shelf libraries ....................................... 3
2.3 Design your program for optimal data locality ................. 4
2.4 Preliminary conclusions .......................................... 10

3 Timing and Profiling a Program 10
3.1 Subroutine based profiling ...................................... 10
3.2 Tick based profiling ............................................. 12
3.3 Timing small sections of your program ......................... 12
3.4 Assembler output, hardware performance monitors ............. 13

4 Floating point operations 13
4.1 Parallelism on the chip .......................................... 14
4.2 Improving the ratio of floating point operations to memory accesses ............... 17
4.3 Special functions .............................................. 18
4.4 Eliminating Overheads .......................................... 19

5 A closer look at cache related performance issues 20
5.1 Directly and indirectly mapped caches ......................... 20
5.2 Cache trashing ..................................................... 21
5.3 Performance of RISC processors for semi-local data access .......... 22
5.4 TLB misses ....................................................... 23
5.5 Blocking strategies ............................................. 25

6 Parallel programs 27
6.1 Introduction to parallelism ....................................... 27
6.2 Ideal and observed speedup ....................................... 28
6.3 Profiling parallel programs ....................................... 30
6.4 Message passing libraries ....................................... 33
6.5 Data locality ..................................................... 35
6.6 Load balancing .................................................... 35
6.7 Coarse-grain parallelism against fine-grain parallelism .......... 36

7 Epilog, acknowledgments and disclaimer 37

8 Appendices 38
8.1 Timing routine for BLAS library .................................. 38
8.2 Program for memory testing ...................................... 39
8.3 Program that should run at peak speed ......................... 43
8.4 Program to test suitability of parallel computers for fine grained applications ...... 47
8.5 MPI timing routine .............................................. 48
2 Some basic efficiency guidelines

2.1 Select the best algorithm

If you want to solve a certain problem there are usually several algorithms at your disposal. Choosing the algorithm that is best suited for both your problem and the computer you want to use is clearly the first and most important step. Frequently the complexity or scaling behavior of two algorithms is different. For instance the number of operations to do matrix-matrix multiplication is $N^3$ for a $N \times N$ matrices with the ordinary algorithm, but only $N^{2.8}$ with the Strassen algorithm. If you have large data sets, the advantage of using a low complexity algorithm (the Strassen algorithm in the matrix multiplication example) is overwhelming and certainly dominating other suitability aspects. If, under such circumstances, the implementation of a low complexity algorithm is not well adapted to your computer architecture, you should rather try to find a better implementation of this algorithm instead of using the high complexity algorithm. Even though the high complexity algorithm may be running faster, the payoff for improving the implementation of the low complexity algorithm could be significant. If, however, for your specific application, two algorithms differ in their operation count only by a small factor, the suitability of this algorithm to your computer architecture may be the dominating aspect. In this tutorial you will learn the basic facts about modern RISC computer architectures allowing you to better assess the suitability of a specific algorithm for these computer architectures.

2.2 Use off-the-shelf libraries

Many high quality numerical libraries are available today. For linear algebra computations the public domain version of the BLAS (Basic Linear Algebra Subroutines) library can be obtained in source form by e-mailing to netlib@ornl.gov (first just send the message "help") or on http://www.netlib.org/index.html. At the same time, most computer manufacturers have a version of BLAS that is specifically optimized for their hardware. A full set of optimized BLAS can be found in ESSL for the IBM machines, in the SCILIB library for the CRAYs, in the DXML library for the DECs and in the SGIMATH library for the SGIs. Since the calling sequences is the same in all vendor-optimized implementations, a program using BLAS routines makes your code portable to any computer. There are 3 levels of BLAS. Level 1 contains vector-vector kernels, such as a scalar product between two vectors. Level 2 deals with matrix-vector operations, such as matrix $\times$ vector multiplication. Level 3 contains matrix-matrix routines, such as matrix $\times$ matrix multiplication. For reasons that will be discussed later, the higher level BLAS routines run faster than the lower level ones. As shown in Table 1, Level 3 comes frequently close to the peak speed of the machine. Other levels of BLAS give good performance too. You should therefore try to replace all the parts of your program doing some of the operations you can find in the BLAS library by calls to this library. By reordering loops in your code it is also frequently possible to replace several lower level BLAS calls by a single call to a higher level routine. This was for instance done in the program 'blaslevel.f', listed in the Appendix, that was used to obtain the timing results for three BLAS routines representative of each level.
Table 1: Comparison of the speed of different levels of BLAS routines on an IBM 590 workstation with a peak speed of 265 Mflops. The same matrix-matrix multiplication is done by calls to different BLAS levels. In this example Level 1 BLAS has non unit stride and for comparison the performance for unit stride is given as well.

<table>
<thead>
<tr>
<th>BLAS routine</th>
<th>Speed (Mflops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGEMM (Level3: matrix matrix mult.)</td>
<td>255</td>
</tr>
<tr>
<td>DGEMV (Level2: matrix vector mult.)</td>
<td>155</td>
</tr>
<tr>
<td>DDOT (Level1: scalar product)</td>
<td>4</td>
</tr>
<tr>
<td>DDOT (scalar product, unit stride)</td>
<td>110</td>
</tr>
</tbody>
</table>

Another public domain library is LAPACK. In this library you can find all standard linear algebra routines, such as the solution of linear systems of equations, singular value decompositions and eigenvalue problems. It supersedes the older LINPACK and EISPACK libraries. This library is built on top of the BLAS library leading to very good performance particularly if an optimized version of BLAS is utilized. Some of the LAPACK routines can also be found in machine optimized form in some vendor libraries. Further information on LAPACK can be found in its users' guide (E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen, SIAM Philadelphia 1992) or on http://www.netlib.org/index.html.

In addition to these public domain libraries, several computer manufacturers have scientific libraries containing additional routines, such as special functions, Fast Fourier Transforms, integration and curve fitting routines. The ESSL library from IBM and the SCILIB library from Cray contain very efficient algorithmic implementations for their specific architecture.

There are other libraries available, both commercially (such as the NAG library) or in the public domain (in the NETLIB repository for example), but the efficiency of these routines varies widely.

EXERCISE: Find the location of the BLAS library on your computer (typically either in /lib, /usr/lib, or /usr/local/lib) and measure the speed of 3 routines belonging to different BLAS levels.

2.3 Design your program for optimal data locality

Modern RISC processors have potentially a very high floating point performance. In most cases though it is not possible to feed the data fast enough into the processor to take advantage of the full floating point performance. The most basic optimization strategy is the minimization of the amount of memory accesses and the organization of the memory traffic in an optimal way. This requires some knowledge about the structure of the memory on RISC architectures which will be presented now.

The memory subsystem of RISC computers is organized hierarchically. It consists of a sequence of memory layers that are getting bigger and bigger as you go up in the hierarchy, but whose access times are getting longer and longer at the same time. The fastest memory level are the registers, placed on the chip housing the floating point unit.
On practically all RISC processors of interest there are 32 logical registers that can be accessed without any delay (0 cycles). The next level in the memory hierarchy is the cache. On most machines there are actually 2 cache levels, level 1 (L1) cache and level 2 cache (L2). L1 cache can usually be accessed at the cost of 1 cycle. Since in many cases a load or store to a register can be overlapped with floating point operations, this 1 cycle can frequently effectively be hidden giving the appearance of a 0 cycle access time. Data transfer from L1 cache to the registers takes place in units of words (i.e. single or double precision numbers). The next hierarchy level is the memory attached to one processor. Data transfer from this local memory to the cache is done in units of cache lines. Depending on the machine, a cache line can hold between 4 and 32 words. If the floating point unit accesses a data word that is not contained in the cache a so-called cache miss occurs. The cache line containing this data has then to be loaded into cache before it can be loaded into a register. A considerable latency, typically 10-100 cycles, is related to the initialization of a cache line load. In the case of a parallel computer, the farthest level in the hierarchy is the global memory comprising all of the local memories of the individual processors.

It is usually difficult to find all the relevant memory performance numbers for all the different computer models. We are therefore providing a benchmarking program ("memory_test" listed in the Appendix) that allows you to find those numbers. It accesses data sets of different sizes, with different strides (stride is the distance measured in words between two memory locations consecutively accessed by a code). The performance figures obtained from this program for load, store and copy (i.e. an equal mix of loads and stores) operations are shown for an IBM 590 workstation in Tables 2 to 4. The first column gives the number of cycles that is necessary to access the data with unit stride. One clearly sees, in all cases, a sudden increase for data sets larger than $2^{15} = 32768$ double precision (8 byte) words. The reason is that the IBM 590 machine has an L1 cache that can hold 32768 double precision words. For small data sets, the stride 1 performance is also poor due to the significant overhead for these short loops. If one looks at the performance for a data set larger than the cache as a function of the stride (presented in the rows in the lower part of the tables), one sees that the performance drops continuously and then saturates for strides larger than or equal to 32. The asymptotic value is quite high, 27 to 28 cycles for a memory copy. This reveals that one cache line holds 32 double-words for this 590 model and that the latency for the transfer of this cache line from memory to cache is between 27 and 28 cycles. Loading all 32 words takes only slightly more cycles than just loading every 32nd element.

Using the information from the copy memory test we can deduce the schematic view of the memory hierarchy of the IBM 590 workstation shown in Figure 1.

In order to obtain optimal data locality you have to try to organize your program in such a way that memory references taking place within short time intervals access memory locations that are spatially close to each other as well. To achieve this desirable effect, it is sometimes enough to order loops in a specific way. In some instances however, it may be necessary to design your data structures in such a way that quantities that are used together will be adjacent in memory as well.
Table 2: Average number of cycles per memory access for the copy case (half and half mix of load and store) for data sets of different size accessed with different strides on an IBM 590 workstation. Both data size and stride are in units of powers of 2.

<table>
<thead>
<tr>
<th>D</th>
<th>3</th>
<th>1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>.8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>.6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>.4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>.4</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>8</td>
<td>.3</td>
</tr>
<tr>
<td>9</td>
<td>.3</td>
<td>.6</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>.3</td>
</tr>
<tr>
<td>11</td>
<td>.3</td>
<td>.5</td>
</tr>
<tr>
<td>12</td>
<td>.3</td>
<td>.5</td>
</tr>
<tr>
<td>13</td>
<td>.3</td>
<td>.5</td>
</tr>
<tr>
<td>S</td>
<td>14</td>
<td>.3</td>
</tr>
<tr>
<td>15</td>
<td>1.0</td>
<td>1.9</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>1.0</td>
</tr>
<tr>
<td>17</td>
<td>1.0</td>
<td>1.9</td>
</tr>
<tr>
<td>Z</td>
<td>18</td>
<td>1.0</td>
</tr>
<tr>
<td>19</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>E</td>
<td>20</td>
<td>1.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

| S | T | R | I | D | E |

Table 3: Same as Table 2, but for loads only

<table>
<thead>
<tr>
<th>D</th>
<th>3</th>
<th>2.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>.9</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td>.6</td>
</tr>
<tr>
<td>7</td>
<td>.5</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>8</td>
<td>.4</td>
</tr>
<tr>
<td>9</td>
<td>.4</td>
<td>.8</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>.4</td>
</tr>
<tr>
<td>11</td>
<td>.4</td>
<td>.7</td>
</tr>
<tr>
<td>12</td>
<td>.4</td>
<td>.6</td>
</tr>
<tr>
<td>13</td>
<td>.4</td>
<td>.6</td>
</tr>
<tr>
<td>S</td>
<td>14</td>
<td>.4</td>
</tr>
<tr>
<td>15</td>
<td>.4</td>
<td>.6</td>
</tr>
<tr>
<td>I</td>
<td>16</td>
<td>.7</td>
</tr>
<tr>
<td>17</td>
<td>.7</td>
<td>1.5</td>
</tr>
<tr>
<td>Z</td>
<td>18</td>
<td>.7</td>
</tr>
<tr>
<td>19</td>
<td>.8</td>
<td>1.6</td>
</tr>
<tr>
<td>E</td>
<td>20</td>
<td>.8</td>
</tr>
<tr>
<td>21</td>
<td>.8</td>
<td>1.6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 4: Same as Table 2, but for stores only

<table>
<thead>
<tr>
<th>3</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td>T</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>13</td>
</tr>
<tr>
<td>S</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>15</td>
</tr>
<tr>
<td>I</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>17</td>
</tr>
<tr>
<td>Z</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>19</td>
</tr>
<tr>
<td>E</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>21</td>
</tr>
</tbody>
</table>

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|

S T R I D E

Figure 1: Schematic view of the memory hierarchy of an IBM 590 workstation

MAIN MEMORY: arbitrary size

Data packet size: 33 doublewords
Access time: (27 / 32) cycles

CACHE: 32768 doublewords

Data Packet size: 1 doubleword
Access time: 3 cycles

32 REGISTERS

Data packet size: 1 doubleword
Access time: 0 cycles

CPU
Let us look at the following two loops that differ only in their loop ordering:

```
10 dimension a(n,n),b(n,n)
LOOP A
   do 10,i=1,n
   do 10,j=1,n
10   a(i,j)=b(i,j)
20 dimension a(n,n),b(n,n)
LOOP B
   do 20,j=1,n
   do 20,i=1,n
20   a(i,j)=b(i,j)
```

According to the Fortran convention, the physical ordering of matrix elements in memory is the following:

```
a(1,1),a(2,1),a(3,1),... ,a(n,1) , a(1,2),a(2,2), a(3,2), ... ,a(n,2), .......
```

We refer to this access pattern as "column major order".

In the C language the storage conventions is just the opposite. The last index is "running fastest", we access arrays in a "row major order".

Let us now consider two limit cases of small matrix sizes (n=32) and large matrix sizes (n=1000). To be specific, let us furthermore assume that we are on an IBM 590 workstation with the memory characteristics discussed above and that the first matrix elements of both a and b are aligned on a cache line boundary (i.e. the first elements of each matrix are also the first elements of a cache-line). Let us also assume that all the data are out of cache at the beginning of the calculation.

We distinguish the following cases:

- **Loop A, small matrices**
  In this case the matrix elements will be accessed in the following order:

```
x(1,1),x(1,2),x(1,3),... ,x(1,32) , x(2,1),x(2,2),x(2,3),... ,x(2,32), .......
```

where x denotes either a or b. However, this is not the physical order in memory. x(1,i) and x(1,i + 1) are actually 32 double-word positions apart and thus a cache-miss will occur for all of the first 32 loads for both a and b. Since the cache can house the $2 \times 32^2 = 2048$ words that were brought into cache during the first 32 iterations of the double loop, all subsequent memory references will be in cache and no more cache misses will occur in all of the remaining loop iterations. To load all the $2 \times 32^2$ numbers we have therefore $2 \times 32$ cache misses, i.e. one cache miss for every 32 words.

- **Loop A, big matrices**
  The memory access pattern will be the same as above, i.e. consecutive memory references are not adjacent. In this case they are 1000 double-words apart. After the first 512 iterations of the double loop (each of which caused a cache miss) we will have loaded $2 \times 512 \times 32 = 32768$ double-words and the cache will therefore be filled. All of the following cache misses will bring in cache lines that will overwrite the ones brought in before. The element x(2,1), which was loaded into cache with x(1,1), will not be available anymore since it was overwritten by another cache-line. This then implies that you have $2 \times 1000^2$ cache misses, i.e. exactly one cache miss for every accessed word.
Loop B

In this case we access the matrix elements exactly in the order in which they are stored in memory. We will then have a cache miss every 32 double-words, but all the data brought into cache by the cache-misses will immediately be used in the following loop iterations. We have therefore only one cache-miss for every 32 memory accesses.

We see that, for large matrices, a tremendous performance loss occurs if we use the loop ordering described in Loop A. The performance degrades roughly by a factor of 30 on an IBM 590. The loop ordering B gives the best data locality and is therefore always to be preferred, even though in the case of small matrices the performance is the same for both orderings. In the more realistic case, when the matrices are not aligned on a cache line boundary, the above conclusion remains valid. In the case of Loop A for small matrices, we would just have on average not one but probably 2 cache misses for loading 32 words.

This example was just chosen for didactic reasons. In reality you should use the BLAS DCOPY routine instead of writing your own:

```fortran
CALL DCOPY(N,N,B,1,A,1)
```

To get optimal data locality one has frequently to design the data structures in the program in such a way that quantities that are used in the same context are physically close in memory (i.e., spatial locality should be enforced). Let us look at the following two versions of the main loop in a molecular dynamics code. The resulting forces are summed up for all particles separated by a distance smaller than the interaction range of the potential 'cutoff'.

**DATA STRUCTURE A**

```fortran
DIMENSION RX(N),RY(N),RZ(N),FX(N),FY(N),FZ(N)

DO 100,I=1,N
  DO 100,J=1,N
    DIST2=(RX(I)-RX(J))**2 +(RY(I)-RY(J))**2 +(RZ(I)-RZ(J))**2
    IF (DIST2.LE.CUTOFF) THEN
      DF= ......
      DFZ= ......
      FX(J)=FX(J)+DF
      FY(J)=FY(J)+DF
      FZ(J)=FZ(J)+DFZ
    ENDIF
  C CALCULATE INTERACTION
  DO 100,I=1,N
  DO 100,J=1,N
  DIST2=(RX(I)-RX(J))**2 +(RY(I)-RY(J))**2 +(RZ(I)-RZ(J))**2
  IF (DIST2.LE.CUTOFF) THEN
    DF= ......
    DFZ= ......
    FX(J)=FX(J)+DF
    FY(J)=FY(J)+DF
    FZ(J)=FZ(J)+DFZ
  ENDIF
  100 CONTINUE
```

**DATA STRUCTURE B**

```fortran
DIMENSION R(3,N),F(3,N)
```
The memory access patterns of the arrays $fx$, $fy$ and $fz$ will be more or less random. In particular, we can not expect that if particle $j$ is close to particle $i$, then also particle $j+1$ will be close to particle $i$. This means that memory references for the $(j+1)$th force component brought into cache by accessing the $j$ component will most likely be overwritten in cache before they will be used in a later accumulation step (we assume here that the cache is not big enough to hold all the data). We have therefore essentially one cache miss for every force component reference. Since there are 3 components there are thus 3 cache misses. For data structure B we can however load all of the three components with one cache miss since they are adjacent in memory. The data structure B is therefore more efficient.

### 2.4 Preliminary conclusions

By following these three basic guidelines: best algorithm, good libraries and optimal data locality, you should be able to write reasonably efficient programs. From now on, we will focus on some more advanced topics that will allow you to further improve the performance of your code. This kind of advanced optimization work is worthwhile only for programs that have a long execution time. You may decide to use the sophisticated optimization techniques described later only for a few important kernels in your program, the so-called hot-spots.

### 3 Timing and Profiling a Program

The first thing you have to know when starting to optimize a program is which parts of it take most of the CPU time, i.e. identification of the hot-spots. Profiling your program will answer this question. Several profiling systems are available and will be discussed in the next sections.

#### 3.1 Subroutine based profiling

For the more traditional subroutine based profiling you have first to compile your program with a special option (usually `-p`). The compiler will then insert timing calls at the begin-
ning and end of each subroutine. At execution time, your program's timing information is written to a file (typically mon.out). The prof command will then allow you to read in the information contained in this file and to analyze your code in the form of a "profile". Let us profile the following short program:

```fortran
implicit real*8 (a-h,o-z)
parameter(nexp=13,n=2**nexp)
dimension x(n)

do 15,i=1,n
  x(i)=1.234d0

  do ic=1,1000
    call sub1(n,x,sum1)
    call sub2(n,x,sum2)
  enddo
  write(6,*) sum1,sum2

end

subroutine sub1(n,x,sum)
  implicit real*8 (a-h,o-z)
dimension x(n)
  sum=0.d0
  do 10,i=1,n
    sum=sum+2.d0*x(i)+(x(i)-1.d0)**2+3.d0*i*1.d-20-4.d0*(i-200)**2
  return
end

subroutine sub2(n,x,sum)
  implicit real*8 (a-h,o-z)
dimension x(n)
  sum=0.d0
  do 10,i=1,n
    sum=sum+x(i)**.3333333333d0
  return
end
```

The output obtained from the prof command looks like this:

<table>
<thead>
<tr>
<th>Name</th>
<th>%Time</th>
<th>Seconds</th>
<th>Cumsecs</th>
<th>#Calls</th>
<th>msec/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>_pow</td>
<td>63.8</td>
<td>21.11</td>
<td>21.11</td>
<td>8192000</td>
<td>0.0026</td>
</tr>
<tr>
<td>_mcount</td>
<td>28.4</td>
<td>9.39</td>
<td>30.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub1</td>
<td>3.8</td>
<td>1.27</td>
<td>31.77</td>
<td>1000</td>
<td>1.270</td>
</tr>
<tr>
<td>sub2</td>
<td>2.3</td>
<td>0.77</td>
<td>32.54</td>
<td>1000</td>
<td>0.770</td>
</tr>
</tbody>
</table>

First, one realizes that the time spent in the compiler library functions is not attributed to the subroutines from which they are called, but to the functions themselves (the power
function is called from subroutine sub2). As a result, if you call the same library function from several subroutines it is not possible to find out directly which subroutines invokes them more frequently and takes thus more time. Second, we note an additional category "mcount". mcount is the time spent in the timing routines themselves, i.e. the overhead of profiling. This can be a significant fraction of the total execution time and it can bias the timing analysis.

3.2 Tick based profiling

All these drawbacks can be avoided by using advanced profiling utilities based on tick counting. Whenever the operating system is interrupting the execution of your program (usually every 1/100 of a second), such a profiling tool will check in which subroutine the interrupt occurred and augment the number of ticks associated with this subroutine. If you compiled your program with some additional options, it can even trace back the interrupt to the line number of your program and add a tick to this line number. The exact sequence of commands you have to use on IBM systems is the following:

```
xlf -g prof.f
tprof a.out
```

For the subroutine in the program above the line level profiling output looks as follows:

```
Ticks Profile for sub6 in prof.f

<table>
<thead>
<tr>
<th>Line</th>
<th>Ticks</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>-</td>
<td>sum=0.0</td>
</tr>
<tr>
<td>73</td>
<td>-</td>
<td>do 10,i=1,n</td>
</tr>
<tr>
<td>74</td>
<td>913</td>
<td>sum=sum+x(i)**.33333333333d0</td>
</tr>
<tr>
<td>75</td>
<td>-</td>
<td>return</td>
</tr>
<tr>
<td>76</td>
<td>-</td>
<td>end</td>
</tr>
</tbody>
</table>

913 Total Ticks for sub6 in prof.f
```

3.3 Timing small sections of your program

Very frequently you only want to know the absolute CPU time for a small, but important, section of your program. If you know the number of floating point operations done in this section, the timing information allows you to calculate the absolute speed. To get these absolute timings you have to insert timing calls yourself, as shown for instance in the program for memory testing above. On the IBM, SGI and NEC computers the timing routine, callable from Fortran, that returns the elapsed CPU time is 'mclock', on the Cray systems it is 'second', on DEC, HP and SUN systems it is 'etime'. The calling sequences and time units of all these routines are different and can be found in the man pages. As in the case of profiling, the time spent by calling the timing routines can become significant if they are called very frequently.

A possible problem with this profiling method is that the resolution of the timing routines is usually fairly coarse (1/100 of a second for 'mclock', 1/1000 of a second for 'second'). In order to get good statistics, the run time of the part of the program that is
analyzed has therefore to be much longer than this resolution. That frequently necessitates to artificially repeat the execution of the relevant part by bracketing it with an additional timing loop. Since in these repeated executions data can be available in cache at the beginning of a new timing iteration, the performance numbers can be artificially high. In order to avoid this pitfall, a call to a cache flushing routine has to be inserted at the beginning of each timing iteration.

In timing runs one also has to make sure that no floating point exceptions (such as overflows) are present, since that will increase the CPU time. Testing with zeroes can also be problematic on some machines. Some compilers are able to figure out that you never use the numerical results of your timing loops and not execute those parts. You then might even get close to infinite speed! A good practice is therefore to print out some of your numerical results of a timing run in order to make sure that the compiler is not outsmarting you.

3.4 Assembler output, hardware performance monitors

For in depth optimization work it is useful to look at the assembler language output generated by the compiler. On IBM machines you get a rather readable pseudo-assembler output by using the compiling options `-qlist -qsource`, that actually also prints out the estimated number of cycles for each instruction. On many other machines the assembler language output can be obtained by using the compiler option `-S`. In general, you will find that the compiler is not doing a perfect job in optimizing the code, even when the optimization flag you used invokes the highest optimization level available (typically `"-O3"`). If your are interested in squeezing out the last ounce of performance, but do not want to start programming in assembler, you can try to rearrange your Fortran program so as to give the compiler some hints. Compilers are however usually quite stubborn creatures and this way of proceeding can therefore be rather frustrating and time consuming. It is in the culture of compilers developers that outsmarting you is beneficial!

Several vendors also provide so-called hardware performance monitors. They give you very detailed information about a program, such as the number of floating point operations, the floating point speed, the number of memory accesses and the number of cache misses. This information can be highly useful for optimization work.

4 Floating point operations

As we have pointed out several times, memory access problems are frequently the single most detrimental factor leading to large performance degradation. In the following section we will concentrate on the optimization of floating point operations. We will assume in this section that all the necessary data are available in cache and that therefore there are no memory access problems. The topic of memory access optimization will again be taken up in the next chapter.
4.1 Parallelism on the chip

Modern RISC processors are called super-scalar architectures. This means that they can do several operations in one cycle. Typically a processor can do one add, one multiply, one load/store and one branching instruction in one cycle. This instruction-level parallelism can be implemented in hardware or software. In hardware, a processor can have several floating point units, whereas in software parallelism is achieved by special instructions, such as the fused multiply-add, that merge two floating point operations into one assembler instruction. Another important fact is that modern RISC processors use pipelines to effectuate the floating point operations. The pipeline can start a new operation in each cycle, but the result can become available only a few cycles later. From the point of view of the programmer, a multistage pipeline has the same effect as several separate floating point units, if the output of floating point operations are immediately used as input for other floating point operations. The important characteristics of several commonly utilized processors are summarized in Table 5.

Table 5: Instruction level parallelism for several processors. The repeat rate and latency for each instruction is given in brackets

<table>
<thead>
<tr>
<th>Processor</th>
<th>Repeat Rate</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM RS/6000 POWER2 (up to 130 MHz)</td>
<td>4 floating point operations per cycle (peak speed 520 Mflops)</td>
<td>4 memory access operations per cycle</td>
</tr>
<tr>
<td>Cray T3E (DEC Alpha EV5) (up to 350 MHz)</td>
<td>2 floating point operations per cycle (peak speed 700 Mflops)</td>
<td>1 memory access operations per cycle</td>
</tr>
<tr>
<td>SGI R10000 (200 MHz)</td>
<td>2 floating point operations per cycle (peak speed 400 Mflops)</td>
<td>1 memory access operations per cycle</td>
</tr>
</tbody>
</table>

To illustrate the effect of parallelism within a single CPU let us look at a simple vector norm calculation.
Let us optimize this vector norm subroutine for an IBM 590 workstation. Looking at Table 5, we see that this processor has 2 floating point units. However, the program has only one independent thread and thus cannot keep two units busy. Furthermore, we see that the output \( tt \) of one fused multiply-add is the input for the next fused multiply-add. We thus have a dependency. Since the latency of the fused multiply-add is 2 cycles, we cannot start a new fused multiply-add every cycle, as it were possible if we had independent operations. With this in mind we restructure the program as follows.

```fortran
subroutine lngth4(n,a,tt)
  implicit real*8 (a-h,o-z)
  dimension a(n)
  t1=0.d0
  t2=0.d0
  t3=0.d0
  t4=0.d0
  do 100, j=1,n-3,4
    t1=t1+a(j+0)*a(j+0)
    t2=t2+a(j+1)*a(j+1)
    t3=t3+a(j+2)*a(j+2)
    t4=t4+a(j+3)*a(j+3)
  100 continue
  tt=t1+t2+t3+t4
  return
end
```

EXERCISE: Generalize the subroutine lngth4 in such a way that it gives the correct result for data sets of arbitrary length.

This kind of transformation is called loop unrolling. Its effect is that it usually improves the availability of parallelism within a loop. On an IBM 590 workstation, the unrolled version runs at peak speed, whereas the original version runs at just a quarter of peak speed.
Loop unrolling is done automatically by some compilers (e.g. SGI, Cray). On most other compilers, flags or compiler directives for doing it are available. Since in an unrolled loop the operands are treated in a different order, the numerical results are not bitwise identical to the results in the original version. Excessive loop unrolling can be detrimental to performance. In this case one runs out of registers and the content of the registers has to be spilled into cache for intermediate storage, leading to a large number of useless load/stores. For larger and more complicated inner loops the number of registers is frequently the limiting factor. A look at the assembler listing can reveal, and allow you to correct, problems caused by spilling. In this context it is worthwhile pointing out that usually the number of physical registers is larger than the number of logical registers (32 on all RISC processors of commercial interest). If you have a processor with a deep pipeline, of say 4 cycles, you might be concerned that all the registers associated with the operation are occupied for the full duration of 4 cycles. By doing a register re-mapping to the excess physical registers, these registers are freed and the number of logical registers required is the same as if the the pipeline had a one cycle depth.

In the subroutine lngth4 shown above, another dependency would be present if the instructions were performed exactly in the order they are given in the program. Each line first loads an array element of a, which is then input for the fused multiply-add. However, the result of the load is not immediately available, but only after one cycle. So the real sequence of instructions looks rather as follows:

c first floating point unit, first cycle:
c do one MA (both t1 and a0 are available in registers) and load a1
   t1=t1+a0*a0
   a1=a(j+1)

c first floating point unit, second cycle:
c do one MA (both t2 and a1 are available in registers) and load a0 for next iteration of loop
   t2=t2+a1*a1
   a0=a(j+0+4)

c second floating point unit, first cycle:
c do one MA (both t3 and a2 are available in registers) and load a3
   t3=t3+a2*a2
   a3=a(j+2)

c second floating point unit, second cycle:
c do one MA (both t4 and a3 are available in registers) and load a2 for next iteration of loop
   t4=t4+a3*a3
   a2=a(j+1+4)

Arranging instructions in groups that can be executed together in one cycle, without delays caused by dependencies, is called software pipelining. An attempt to do software pipelining is done by most compilers. In easy cases, like our norm calculation, their attempt is usually successful. In more complicated cases, they can fail to generate optimally pipelined code. Trying to do software pipelining by hand can in some circumstances help.
Accurate software pipelining is actually becoming less important. Several RISC processors (SGI R1000, HP PA-8000, Intel's Pentium Pro) have out of order instruction execution. There is a queue of several instructions (16 instructions for instance on the R10000) waiting to be executed, but only those for which all the input data are available are executed. For such chips it is not necessary that the programmer finds a way to order the instructions in an optimal way.

4.2 Improving the ratio of floating point operations to memory accesses

In addition to the advantages already discussed, loop unrolling can also be used to improve the ratio of floating point operations to load/stores. If the CPU time of the loop is dominated by loads/stores, this can result in significant speedups. Let us look at the following two matrix-vector multiplication routines.

```fortran
subroutine mult(n1,nd1,n2,nd2,y,a,x)
implicit real*8 (a-h,o-z)
dimension a(nd1,nd2) ,y(nd2) ,x(nd1)

do i=1,n1
t=0.d0
do j=1,n2
   t=t+a(j,i)*x(j)
endo
y(i)=t
enddo
return
end

subroutine multo (n1,nd1,n2,nd2, y,a,x)
c works correctly only if n1,n2 are multiples of 4
implicit real*8 (a-h,o-z)
dimension a(nd1,nd2) ,y(nd2) ,x(nd1)

do i=1,n1-3,4
   t1=0.d0
t2=0.d0
t3=0.d0
t4=0.d0
do j=1,n2-3,4
   t1=t1+a(j+0,i+0)*x(j+0)+a(j+1,i+0)*x(j+1)+
   a(j+2,i+0)*x(j+2)+a(j+3,i+0)*x(j+3)
   t2=t2+a(j+0,i+1)*x(j+0)+a(j+1,i+1)*x(j+1)+
   a(j+2,i+1)*x(j+2)+a(j+3,i+1)*x(j+3)
   t3=t3+a(j+0,i+2)*x(j+0)+a(j+1,i+2)*x(j+1)+
   a(j+2,i+2)*x(j+2)+a(j+3,i+2)*x(j+3)
   t4=t4+a(j+0,i+3)*x(j+0)+a(j+1,i+3)*x(j+1)+
   a(j+2,i+3)*x(j+2)+a(j+3,i+3)*x(j+3)
endo
```

17
First of all, let us note that the loop ordering is the optimal one from the point of view of data locality. All of the memory accesses have unit stride. If the loop over $i$ was the innermost one, the stride would be $ndl$ for accesses to $a$. The effect to be demonstrated here is related to the better floating point to load/store ratio in the optimized version ”multo”. In the case of the unoptimized version, two array elements have to be loaded into registers ($a(j,i), x(j)$), resulting in 2 loads for 1 multiplication and 1 addition per loop iteration. In the unrolled case 20 elements (16 elements of $a$, 4 of $x$) have to be loaded into registers, resulting in 20 loads for 16 multiplications and 16 additions. This effect is particularly important on machines that can do more floating point operations than loads per cycle. At the same time, we have also eliminated dependencies and better exposed the parallelism. Timing on an IBM 590 workstation shows that, for the unoptimized version, 1 cycle is needed per loop iteration, whereas in the optimized version only .27 cycles are needed, corresponding to a speed of 240 Mflops. This impressive performance number can not be explained solely by the eliminated dependencies and improved parallelism. If we had 2 loads per loop iteration, the best we could expect from Table 3 would be .8 cycles. Our earlier observation, that the higher level BLAS routines perform better than the lower level ones, is related to the fact that by suitable loop unrolling one can obtain a better floating point to load/store ratio for the higher level routines than for the lower level ones.

EXERCISE: Calculate the floating point to load/store ratio for a scalar product, a matrix times vector and a matrix times matrix multiplication for the case that all loops are unrolled by a factor of 3. Are there enough registers to unroll a matrix times matrix multiplication by a factor of 4 with respect to all indices?

4.3 Special functions

Special functions, such as divides, square roots and exponentials are very expensive. Table 6 shows the number of cycles required for some common special functions on several processors. A large fraction of the CPU time for the calculation of these function goes actually into the calculation of the last few bits. So relaxing accuracy demands has the potential of considerably speeding up these calculations. Some vendors, such as CRAY, have libraries that calculate special functions with slightly reduced accuracy, but significantly faster (Table 6). Concerning divisions, some compilers have options that allow you to override the IEEE accuracy standards and to replace divisions by the faster sequence of an inverse and a multiply. By using information about the context in which the special functions are called or about the values their arguments take, it is often possible to write your own faster version of a special function.
Floating point performance as a function of the loop length for different processors. The timing also includes the subroutine calling overhead.

4.4 Eliminating Overheads

The completion of floating point or integer operations, responsible for the useful numerical work in a code, is always accompanied by all kinds of extraneous costs, such as loop and subroutine calling overheads that can take a significant fraction of time. In general, you should try to have as little branches as possible in the important inner loops of your program. Branches, which are generated either by if statements or by encountering the last loop iteration, will interrupt the smooth flow of data within the CPU’s pipelines and lead thus to unused cycles. Correct branch prediction can considerably boost performance. Several processors contain hardware that accumulates old branching patterns to predict future branching. We have developed a test program 'peak.f' (listed in the Appendix) to measure the combined loop and subroutine calling overheads on several processors. Since this program has no memory references and contains no dependencies it should run practically at peak speed on any RISC processor in the limit of very long loops. The results are shown in Figure 2.

Both the IBM and SGI processors approach the peak speed within nearly 10 percent,
showing that the asymptotic loop overhead is roughly 1 cycle on the IBM (8 cycles are needed to go through the loop) and roughly 2 cycles on the SGI (with 16 cycles). The very good performance of the SGI processor for very short loops is impressive. In order to get this good asymptotic speed on the SGI, it was however necessary to rewrite the subroutine as shown in the Appendix. On the Cray, we were not able to get close to peak speed, even though we software pipelined the subroutine by hand. The reason for that is that the compiler is undoing some of the software pipelining and it is not always grouping a multiplication and an addition together as it would be necessary to keep both the addition and multiplication units busy. We encountered similar problems on the HP 730 processor, which also has separate and rather long addition and multiplication pipelines. The R10000 has separate addition and multiplication units, but it has the additional capability of out of order instruction execution, thus preventing idle functional units. In addition, its instruction set contains a fused multiply-add.

As a general comment, it seems to be the case that present compiler technology is not able to produce code that is making good use of the instruction level parallelism!

5 A closer look at cache related performance issues

The optimization of the memory access mainly involves minimizing the number of cache misses. Cache misses can be separated into three categories:

- compulsory misses, related to the first reference to a cache line that is not cache resident
- capacity misses, when not all cache lines needed can fit into the cache
- conflict misses, related to set associativity rules for direct mapped caches.

In the following sections cache misses will be discussed in more detail.

5.1 Directly and indirectly mapped caches

When a cache line is loaded from the main memory it can actually be placed in a limited number of locations in cache. Any word in memory can be mapped only to one or a few locations in the cache. In the case of a directly mapped cache, there is only one possible location. Indirectly mapped caches allow for more than one location. If the size of the cache is $n$ words, then the $i$th word in memory can be stored only in the position given by $\mod(i, n)$ in a directly mapped cache. If we have a $m$-way associative cache of size $m \times n$, then any location in memory can be mapped to $m$ possible locations in cache, given by the formula $jn + \mod(i, n)$, where $j = 0, \ldots, m - 1$. The situation for a 4-way associative cache is illustrated in Figure 3.

If in an $m$-way associative cache all the $m$ possible locations for a data set are taken, one of them has to be overwritten. On most processors, the least recently used entry (i.e. the entry that was not used for the longest time) will be overwritten. On the 3-way associative L2 cache of a Cray T3E the entry to be overwritten is chosen at random.
Figure 3: Mapping rules for a 4-way associative cache. Each memory locations can be mapped to 4 cache locations, but of course many memory locations map to the same 4 cache locations.

5.2 Cache trashing

Because of the mapping rules, the effective cache size is usually smaller than the physical cache size, leading to conflict misses. The effective cache size would be equal to the physical cache size only if a data item from the main memory could go into any location in the cache. Since this is not the case, there will frequently be unoccupied "data holes" in the cache, thus reducing the effective cache size. The extreme case, when most of the cache is not available because of these mapping rules, is called cache trashing. Let us look at the following program.

```fortran
program cache_trash
  implicit real*8 (a-h,o-z)
  parameter(nexp=13,nn=2**nexp,nbuf=0*81)
  dimension w(nn+nbuf,6)
  n1=2**10
  nit=(2**22/n1)
  do 30,it=1,nit
    call sub(n1,w(1,1),w(1,2),w(1,3),w(1,4),w(1,5),w(1,6))
  30 continue
end

subroutine sub(n,x1,x2,x3,x4,y1,y2)
  implicit real*8 (a-h,o-z)
  dimension x1(n),x2(n),x3(n),x4(n),y1(n),y2(n)
  do 15,i=1,n-1,2
```

21
We ran this program on an IBM 590, once with the parameter \( nbuf = 0 \) and then with \( nbuf = 81 \) (or any other reasonable nonzero value). In the first case we get a performance of just 3.5 Mflops, in the second case a performance of 67 Mflops is achieved even though we did not modify at all the subroutine that is doing all the numerical work. The reason for the extremely poor performance is that, in the first case, all 6 memory references are mapped to the same 4 slots in cache. So, even though all the data of size \( 6 \times 2^{10} = 6144 \) that are accessed in the subroutine could easily fit into cache, the effective cache size in this case is only \( 4 \times 32 = 128 \) double words, which is not enough to hold all of the 6 cache lines. This example is of course contrived. In most applications the starting elements of these 6 arrays will not be separated be a high power of 2. There are however lots of algorithms, most notably FFTs, fast multi-pole methods and wavelet transforms, where the leading dimensions are typically high powers of 2. Introducing some buffer at the end of the array will fix your performance problem in these cases. It is clear that the likelihood of running into cache trashing is higher for directly mapped caches than for set associative ones. Even if you access several arrays in a subroutine that will not map to cache in a pathological way, there will seldom be perfect mapping. As a rule-of-thumb, your effective cache size is roughly half the size of the physical cache size. If you do not want to loose half of the effective cache size you may have to align all the arrays by hand in memory. This can either be done by copying them in a work array and choosing the starting positions of the sub-arrays in the work array in an optimal way, or by aligning them in a common block. The first method has the advantage that you can choose dynamical starting positions that vary during program execution.

5.3 Performance of RISC processors for semi-local data access

One frequently distinguishes between local, semi-local and non-local data access patterns. Local means that you access your data with stride one and that you are using your data many times before loading other data into cache. There are only compulsory cache misses and they are negligible if the data in cache are many times reused. This is of course the optimal situation on a RISC processor. Non-local access is the worst. You access your data with a large stride while using them only once. In this case, each memory access incurs the penalty of a cache-miss. The intermediate case is the semi-local data access. There you access your data with unit stride, but the data set is too large to fit into cache, and you use it thus effectively only once. You have capacity cache misses, but the cost of one cache miss is distributed over all the words of one cache-line.

In many scientific programs you will find large portions of semi-local data access patterns. So it is interesting to see how well different RISC processors perform in this regime. By calling the BLAS DCOPY subroutine with unit strides from within a program
very similar to the program memory_test listed in the Appendix, we obtained the results shown in Figure 4.

For data sizes that fit into cache, we again note very high performance. This performance is then dropping abruptly as we get out of cache. The lower saturated value is the semi-local performance. There is clearly no correlation between fast cycle time and the asymptotic semi-local memory bandwidth performance. It is also surprising that the Cray T3E processor that is based on the DEC Alpha is so much better in performance than the original DEC Alpha. This is due to the fact that Cray has put additional hardware and software around the CPU to partially overcome the worst bottlenecks.

5.4 TLB misses

RISC workstations are virtual memory machines. This means that they distinguish between a logical and a physical memory address. The logical address is the address you are using in your program. In other words, two array elements, say x(1000) and x(3000), are not necessarily at the physical locations 1000 and 3000. Things are somewhat more complicated. Each address belongs to a page. On the IBM 590 computer one page holds \(2^{11} = 2048\) double-words. So the array element x(1000) would be on the first page and x(3000) on the second page. A page can be stored in any place in memory or even on a disk. A page table keeps track of where all the pages go. The entries of that table, that are most frequently used, are stored in a special cache that is called the table look-aside buffer (TLB). So, if you access a logical memory location that belongs to a logical page whose physical location is not in the TLB, that page location has to be fetched from the much larger page table and something similar to a cache miss, called a TLB miss, will occur. From the point of view of the programmer the effect is exactly the same as if there was an extra cache level, whose cache-line size is equal to the page size. On an IBM 590, the TLB is a 2-way associative cache with \(2 \times 512\) entries each holding 2048 double-words. Using a slightly modified version of the program memory_test.f named tlb_test.f we get for the loads the results shown in Table 7.

Table 7: Average number of cycles needed to load data sets of different size with different strides on an IBM 590 workstation. Both data size and stride are in units of powers of 2.

<table>
<thead>
<tr>
<th>D</th>
<th>17</th>
<th>19.4</th>
<th>18.9</th>
<th>18.6</th>
<th>17.6</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>18</td>
<td>19.3</td>
<td>19.0</td>
<td>18.6</td>
<td>17.6</td>
<td>17.9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>19</td>
<td>19.8</td>
<td>20.0</td>
<td>20.0</td>
<td>20.4</td>
<td>20.4</td>
<td>20.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>20</td>
<td>19.9</td>
<td>20.0</td>
<td>20.4</td>
<td>21.0</td>
<td>20.9</td>
<td>21.0</td>
<td>21.1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>19.9</td>
<td>20.1</td>
<td>19.8</td>
<td>20.1</td>
<td>20.8</td>
<td>20.9</td>
<td>20.4</td>
<td>20.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>22</td>
<td>20.3</td>
<td>20.9</td>
<td>22.4</td>
<td>25.1</td>
<td>30.3</td>
<td>39.2</td>
<td>38.9</td>
<td>39.0</td>
<td>39.8</td>
</tr>
<tr>
<td>I</td>
<td>23</td>
<td>20.1</td>
<td>20.9</td>
<td>22.2</td>
<td>24.8</td>
<td>29.7</td>
<td>39.0</td>
<td>39.2</td>
<td>39.0</td>
<td>39.2</td>
</tr>
<tr>
<td>Z</td>
<td>24</td>
<td>20.0</td>
<td>20.8</td>
<td>22.0</td>
<td>24.8</td>
<td>29.5</td>
<td>39.3</td>
<td>39.0</td>
<td>39.0</td>
<td>38.9</td>
</tr>
<tr>
<td>E</td>
<td>25</td>
<td>20.1</td>
<td>20.6</td>
<td>21.3</td>
<td>24.2</td>
<td>29.5</td>
<td>38.9</td>
<td>39.0</td>
<td>39.0</td>
<td>39.0</td>
</tr>
</tbody>
</table>

We see indeed again exactly the same pattern as when we looked at cache behavior in Table 3, only that the effects are happening now for larger data sizes and larger strides.
Figure 4: Mega copy operations per second for different data sizes (given in powers of 2) on different RISC processors.
We note a clear transition for data sets larger than $2^{22}$ double-words. This confirms that the TLB can hold the physical page addresses of up to $2 \times 512 \times 2048 = 2^{21}$ double-words. For data sets larger than $2^{21}$, the performance decreases slowly with respect to the stride. Saturation is visible for strides larger than $2^{13}$. Once the stride is larger than or equal to this number we therefore encounter a TLB miss in addition to a cache miss for every memory reference. We thus conclude that the page size is $2^{11} = 2048$ (If we had a directly mapped TLB the page size would be $2^{12}$). Concerning the effective TLB size as compared to the physical one, exactly the same remarks as in the case of the cache apply. In particular you can have TLB trashing if accessing several data items whose logical pages all map to the same location in the TLB. When you organize your data access in such a way that no substantial cache trashing occurs, it is also likely that TLB trashing will be avoided.

EXERCISE: Write a pathological program where TLB trashing is more important than cache trashing.

5.5 Blocking strategies

Blocking (or tiling) is a strategy for obtaining better data locality for loop structures where it is not possible to have sequential access for all the referenced arrays. The easiest example is a matrix transposition as shown in the program block.f below.

```
program block
  implicit real*8 (a-h,o-z)
  parameter (nx=1050)
  dimension a(nx,nx),b(nx,nx)

  subroutine rot(n,a,b)
    implicit real*8 (a-h,o-z)
    dimension a(n,n),b(n,n)
    do 100,j=1,n
      do 100,i=1,n-3,4
        b(i+0,j)=a(j,i+0)
        b(i+1,j)=a(j,i+1)
        b(i+2,j)=a(j,i+2)
        b(i+3,j)=a(j,i+3)
      100 continue
    return
  end
```

Obviously it is not possible to have unit stride access for the elements of both arrays $a$ and $b$. Once the data sets are larger than the size of the cache we get therefore a significant performance degradation as shown in Table 8. The fact that we have reasonable performance for small data sets suggests the solution for large data sets. Instead of transposing the whole big matrix in a one big chunk, we subdivide it into smaller sub-matrices and we transpose each of these smaller arrays. This divide-and-conquer strategy is called blocking and it is shown schematically in Figure 5 where the sub-matrices are denoted by different colors.
The FORTRAN implementation of the blocked version is the following:

```
program block
  subroutine rotb(n,a,b,lot)
    c works correctly only if data set has length that is multiple of 4
    implicit real*8 (a-h,o-z)
    dimension a(n,n),b(n,n)
    do 100,jj=1,n,lot
    do 100,ii=1,n,lot
    do 100,j=jj,min(n,jj+(lot-1))
    do 100,i=ii,min(n,ii+(lot-1))-3,4
       b(i+0,j)=a(j,i+0)
       b(i+1,j)=a(j,i+1)
       b(i+2,j)=a(j,i+2)
       b(i+3,j)=a(j,i+3)
 100          continue
    return
  end
```

The blocking parameter \textit{lot} depends on the cache size. If we had the full physical cache size at our disposal, we would put $2 \times \text{lot}^2 = \text{cache-size}$. But because of the mapping rules previously discussed, the effective cache size is smaller than the physical cache size and we have to choose a smaller value of \textit{lot}. The performance for 3 different values of blocking parameters is shown in Table 8. It can be seen that for the larger blocking values we have, for some data sets, problems associated with cache trashing even though the leading dimension is not a high power of 2. The transposition routine DGETM from the ESSL library is anticipating these problems and is doing a very good job for all big data sets.
Table 8: Number of cycles on an IBM 590 workstation needed to do matrix transposition of different sizes $n$. The cache size is 32000 double words. The performance of the unblocked version is compared with the performance of 3 blocked versions and the IBM ESSL library.

<table>
<thead>
<tr>
<th>$n$</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>256</th>
<th>384</th>
<th>512</th>
<th>768</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>no block</td>
<td>1.2</td>
<td>.9</td>
<td>.9</td>
<td>2.1</td>
<td>2.2</td>
<td>20.2</td>
<td>21.0</td>
<td>20.9</td>
<td>22.9</td>
<td>22.7</td>
<td></td>
</tr>
<tr>
<td>block=32</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>2.5</td>
<td>2.5</td>
<td>3.1</td>
<td>3.3</td>
<td>3.1</td>
<td>3.3</td>
<td>3.9</td>
</tr>
<tr>
<td>block=64</td>
<td>1.3</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>2.4</td>
<td>2.4</td>
<td>2.8</td>
<td>2.7</td>
<td>3.0</td>
<td>2.8</td>
<td>21.4</td>
</tr>
<tr>
<td>block=96</td>
<td>1.2</td>
<td>.9</td>
<td>.9</td>
<td>1.0</td>
<td>2.3</td>
<td>2.3</td>
<td>2.7</td>
<td>2.6</td>
<td>20.0</td>
<td>2.7</td>
<td>21.3</td>
</tr>
<tr>
<td>ESSL</td>
<td>1.4</td>
<td>1.0</td>
<td>1.0</td>
<td>1.1</td>
<td>2.2</td>
<td>2.4</td>
<td>2.7</td>
<td>2.6</td>
<td>.31</td>
<td>2.8</td>
<td>4.0</td>
</tr>
</tbody>
</table>

You might wonder whether it might be necessary to block with respect to several levels of the memory hierarchy such as L1 cache, L2 cache and TLB size. According to our experience that is usually not necessary. Level 1 caches are usually very small and blocking with respect to them will lead to very short loops that give poor performance. Blocking with respect to the TLB would be necessary only for very large data sets and TLB misses are in general quite negligible compared to cache misses. So in the case of a 3 level memory hierarchy it will usually only be necessary to block with respect to the L2 cache.

Blocking is error prone and the code will become less legible. Some compilers do the blocking for you when invoked with certain options. On IBM machine for instance you have to use `xlf -qhot -qcache:...`.

6 Parallel programs

An important part of the optimization of a parallel program is the serial optimization discussed previously. In addition, considerations related to the parallel architecture have to be taken into account. In the case of a parallel program, it is particularly important that the developer of a code takes the basic characteristics of the computer into account during the development phase. The layout of the data structures and the assignment of the computational subtasks to the different processors can be easily tailored to the parallel architecture at this stage, but it is very difficult to modify them for a finished large code. As a general remark, obtaining very high performance on advanced parallel systems often requires new algorithmic and software approaches. These new methods can for instance exhibit a higher degree of parallelism and are frequently quite different from the conventional methods implemented for a serial or vector machine. The alternative approach of using tools to parallelize existing serial codes based on conventional algorithms usually leads to very poor results.

6.1 Introduction to parallelism

Parallel computing involves solving your problem on multiple processors. Parallelism emerges as a natural solution for those cases when you need:

- faster turnaround (elapsed) time for accomplishing your computations
- amounts of memory larger than available on one processing element.

There are many hardware arrangements for supporting parallel computing. A convenient way to classify parallel architectures is based on memory access. Along these lines, at one end of the spectrum in the classification are distributed-memory architectures, of which loosely-coupled clusters of workstations are one example. At the other end, we have tightly integrated shared-memory computers. Cluster of workstations are connected by conventional networks such as Ethernet or FDDI. Communication libraries such as MPI or PVM allow you to run parallel program on this hardware configuration. Shared-memory machines, as their name implies, have a global memory image that can be rapidly accessed by the processing elements. In general, such architectures offer a very fast memory access time, a few orders of magnitude faster than for clusters of workstations. The implication that shared-memory machines are better or preferable is not necessarily true. Contention on the unique memory bus through which all inter-processor memory traffic is directed leads to scalability problems. There is a limit on the number of processors that shared-memory architectures can utilize (typically a few dozen processors). In-between clusters of workstations and shared-memory machines, there are numerous other types of parallel architectures of interest. For example, tightly integrated clusters, such as the IBM SP2, in which the interconnecting network is very fast and "switched" (i.e., the network allows for multiple simultaneous communications). Also, a new emerging trend in architectural design is the so-called Distributed Shared-Memory architectures in which the memories are physically distributed, but a global memory view is present via a combination of hardware and software. Distributed Shared-Memory architectures offer a compromise between fast access and scalability. However, for such machines the memory subsystem becomes extremely complicated with a direct impact on the ease of use when performance is of importance.

6.2 Ideal and observed speedup

If \( N \) is the number of processors (or processing elements), the ideal speedup for your problem is:

\[
S_i = N
\]  

The ideal speedup is a theoretical limit, not attainable except for a very small subset of real problems called "embarrassingly parallel". The real speedup \( S_r \) is usually much smaller than the ideal speedup. There are actually several definitions of the real speedup and one has to be very careful concerning claims of very high speedups. Two of these definitions namely strong scalability and weak scalability will be discussed here. Strong scalability means that one enlarges the numbers of processors for a fixed problem size. The workload per processor therefore decreases. Weak scalability means that one increases both the problem size and the number of processors. The workload per processor is therefore constant or can even increase if the overall workload grows faster than linearly with respect to the size of the computational problem. High speedups are easier to obtain for the weak scaling case. In real programs two main factors prevent you from obtaining the ideal speedup \( S_i \). The first one is related to the fact that in any program
non-parallelizable segments are present. Amdahl’s law tells you what kind of speedup you can expect as a function of the fraction of the workload that is parallel. The second limiting factor comes from the fact that the access to global memory involves interprocessor communication and is therefore rather slow. So in practice you cannot even attain the speedup predicted by Amdahl’s law.

If we denote by $f$ the fraction of your code that is parallelizable (i.e., CPU time for parallelizable sections/CPU time for the whole program), Amdahl’s law states that the upper bound for speedup is given by:

$$S_a = \frac{1}{1 - f + f/N}$$  \hspace{1cm} (2)

The Amdahl speedup $S_a$ is plotted for several values of $f$ in Figure 6. Unless a really large fraction of your code can be parallelized, the speedups are limited to fairly small values. In fact, since $S_a < 1/(1 - f)$, $S_a$ will never exceed this value regardless of how many processors are used. If the sequential component of your code is the bottleneck, then the solution is increasing $f$ rather than $N$. This though involves using algorithms better suitable to parallel computation, or at least restructuring your code.

What we really are interested in is the so-called observed Speedup $S_o$, which is obtained by experiment. By measuring the elapsed time of your serial program, and the elapsed
time for its parallel version, we can write:

\[ S_o = \frac{\text{elapsed time for serial version}}{\text{elapsed time of parallel version}} \]  

(3) 

\( S_o \) includes the effect on your code's performance from all real-life factors affecting performance such as communication overheads for data transfer between different processors. The observed speedup \( S_o \) is therefore always lower than the the Amdahl limit. For a hypothetical program this is shown in Figure 6. Note that in the Amdahl case the speedup will saturate at the Amdahl limit \( 1/(1 - f) \), whereas in a real life program the speedup will actually start to degrade if you add too many processors since the communication overhead could become the dominating factor.

Parallel performance can be modeled. This implies that you know your algorithms well and that you are able to quantify the amount of arithmetic being performed. Similarly, the amount of communication needs to be understood and cast in models for communication. The amount of time and effort in coding time saved by utilizing performance models can be considerable. These communication models are however strongly hardware dependent and will therefore not be discussed in this tutorial.

### 6.3 Profiling parallel programs

Fairly sophisticated parallel performance analysis tools are available from several vendors. Their drawback is that they are not too easy to use, have limited capabilities (for instance only interactively but not in batch) and are intrusive (thus considerably modifying the timing profile of your code). The learning curve for being able to use such tools efficiently is quite significant too. Below we present a parallel timing routine that can give you sufficient information to optimize a parallel program, together with a sample calling program:

```fortran
program time_mpi
  c sums the rows and columns of a square matrix in parallel

  implicit real*8 (a-h,o-z)
  logical parallel
  include 'mpif.h'
  c serial or parallel
  parameter(parallel=.true.)
  c repeat to get long enough measuring interval
  parameter(nrep=100000,nrec=100)
  c dimension of matrix
  parameter(npjc=4)
  dimension aa(npjc),bb(npjc)

  if (parallel) then
    c if parallel initialize MPI
    write(6,*) 'start mpi'
    call MPI_INIT(ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, iproc,ierr)
    call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
    write(6,*) 'mpi started', iproc,nproc
    if (iproc.eq.0) write(6,*) 'nproc=',nproc
    if (nproc.ne.npc) stop 'wrong number of processors'
```

30
else
c if run serially
  iproc=0
  nproc=1
endif

c initialize timing routine
  call timing(' ', 'INI', 0, iproc, nproc)
c each processor initializes its column (aa(i,j)=i)
do 10, i=1, nproc
  aa(i)=i
enddo
c each processors sums column
  nflop=nproc*nrep
  call timing('SUMS ', 'STR', nflop, iproc, nproc)
do irep=1, nrep
    sum=0.d0
    do i=1, nproc
      sum=sum+aa(i)
    enddo
  enddo
  call timing('SUMS ', 'END', nflop, iproc, nproc)
write(6,*) 'processor, sum', iproc, sum

c transpose matrix
  nflop=nproc*nrec
  call timing('TRANSP', 'STR', nflop, iproc, nproc)
  if (parallel) then
    do irep=1, nrec
      call MPI_ALLTOALL(aa, 1, MPI_DOUBLE_PRECISION,
                       bb, 1, MPI_DOUBLE_PRECISION,
                       MPI_COMM_WORLD, ierr)
    enddo
  else
    bb(1)=aa(1)
  endif
  call timing('TRANSP', 'END', nflop, iproc, nproc)
c each processors sums column of transposed matrix (row of original matrix)
  nflop=nproc*nrep
  call timing('SUMS ', 'STR', nflop, iproc, nproc)
do irep=1, nrep
    sum=0.d0
    do i=1, nproc
      sum=sum+bb(i)
    enddo
  enddo
  call timing('SUMS ', 'END', nflop, iproc, nproc)
write(6,*) 'processor, T sum', iproc, sum

c finalize timing routine
  call timing(' ', 'FIN', 0, iproc, nproc)
if (parallel) then
  call MPI_FINALIZE(ierr)
endif

end

subroutine timing(category,action,nflop,iproc,nproc)
c subroutine for performance analysis of parallel MPI programs
c possible actions are:
c 'INI' to initialize all the counters at the beginning of program
c 'STR' to (re)start accumulating counters for a certain class
c 'END' to end accumulating counters for a certain class
c 'FIN' to print the performance statistics
c The timing categories are defined by the user in the data statement below.
c Only one category can be active at a time
c NFLOP: number of floating point operations or communication operations
c The performance analysis is written to the file 'time.prc'
C written by S. Goedecker

implicit real*8 (a-h,o-z)
include 'mpif.h'
character*6 category, cats
character*3 action
logical init
parameter (ncat=2)
dimension cats(ncat),flops(ncat),timesum(ncat+1),
  wrktime(ncat+1),wrksquare(ncat),wrkflops(ncat),wrkflop2(ncat)
save cats,time0,init,timesum,flops,total0

c define your performance categories (set NCAT to correct value)
data cats / 'SUMS ', 'TRANSP' /

( SEE APPENDIX FOR THE FULL LISTING OF THIS SUBROUTINE!)

end

The output of this run on 4 IBM SP2 processors is shown below. Since each processor
did exactly the same amount of work, the deviation in the number of FLOPS (Floating
Point Operations) is zero. We note however a small deviation in the execution time on dif-
ferent processors. If a communication step (MPI ALLTOALL) is analyzed with this tool,
then you have to pass the number of data to be sent or received as the input parameter
"nflop". The output file will then print in this category the number of Msends/Mreceives
per second instead of Mflops. The fact that the total execution time (TOTAL) equals
in this example the sum of all timing categories (SUM) means that we have included all
relevant parts of the program in the timing procedure and that the time overhead for the
timing calls was negligible.

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>TIME (sec)</th>
<th>PERCENT</th>
<th>FLOPS</th>
<th>SPEED (Mflop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUMS</td>
<td>.450E-01</td>
<td>.50E-02</td>
<td>42.857</td>
<td>.800E+06</td>
</tr>
</tbody>
</table>
6.4 Message passing libraries

Message passing architectures utilize communication libraries (such as MPI, PVM or P4) for exchanging non-local data (i.e. data residing in the memory belonging to another processor). Communication libraries are built on top of communication protocols, the equivalent of the assembler language for the network. Two main performance numbers are associated with communication. The first one is the latency. By definition the latency is the time needed to send a zero-length message from one node to another. Alternatively, one can also interpret the latency as the start-up cost for a message. Another parameter of great importance in describing message passing is the bandwidth of the network. Bandwidth is usually quoted as the maximum speed of message transfer in Mbytes/second for an infinitely long message. The effective bandwidth (i.e. the bandwidth measured for a finite length message) involves the latency and is therefore message size dependent. A typical curve of the effective bandwidth vs. message size on the IBM SP2 is presented in Figure 7.

![Figure 7: The effective point-to-point bandwidth as a function of the message size on an IBM SP2 and SGI Origin 2000](image)

By using these two parameters, we can express the time $T$ for sending a message as:

$$ T = L + \frac{M}{B} $$

where $L$ is the latency, $M$ is the size of the message and $B$ is the bandwidth. The latency $L$ can thus be obtained from the intercept of $T$ with the y-axis. It is apparent that
for communication patterns involving frequent but short messages latency is the dominant factor in T. Large infrequent messages are bandwidth dominated.

Table 9 contains latency and bandwidth numbers for some widely utilized parallel architectures.

The program utilized to gather the data in Figure 7, bandwidth.c, is available on the anonymous ftp server indicated at the beginning of the Appendix. The program is designed to be interactive. You will be prompted for three lines of input: the minimum and maximum number of processors, the number of iterations (repetitions) for the run and the minimum and maximum message size in KBytes. If minimum number of processors indicated is different from its maximum and/or the minimum message size is different from the maximum size, bandwidth.c will generate an "average random bandwidth" as a measure of the effective network bandwidth. Processors will be paired randomly, messages of random length (within the specified limits) will be sent and an average point-to-point bandwidth calculated. Such bandwidth is a more realistic measure of the real bandwidth, as you always compete for the network with other applications running on the system. For bandwidth as defined above, you just need to specify 2 processors and the desired message size as input to bandwidth.c.

The MPI message passing library has been adopted as a standard by nearly all computer manufacturers. If you write your program using MPI calls, it will be portable and it will give good performance since nearly all manufacturers have an MPI implementation that is optimized for their specific communication hardware. The MPI library contains not only basic point-to-point routines, where just 2 processors are exchanging a message, but it contains also a wide range of global communication routines where all processors are involved in a concerted communication step. Examples of global communication routines are for instance global reduction sums, where all the sub-results of the individual processors are summed up, as well as global data rotations where a matrix is transposed in the global memory.

On a parallel computer where the bandwidth is high, a good practice is to concatenate messages. One can then replace several MPI calls by a single one reducing thus the cost in latency. You should also use the global MPI communication routines rather than constructing them yourself using MPI point-to-point routines.

Another optimization technique is taking advantage of the overlap between computation and communication that the hardware may provide. However, not all parallel machines allow overlap. For instance, overlap is possible on SGI's Origin 2000, but not on the IBM SP2.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Latency (microsec)</th>
<th>Bandwidth (Mbytes/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGI Origin2000</td>
<td>0.4</td>
<td>300</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>15</td>
<td>70</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>6</td>
<td>330</td>
</tr>
</tbody>
</table>

Table 9: Values of latency and bandwidth on several parallel machines

Global communications cannot be cast in a simple two-parameter model. The notion of bisectional bandwidth of a machine comes into play here. It is the cumulative
bandwidth of all network links cut by an imaginary plane which separates the network in two equal halves. The bisectional bandwidth is strongly dependent on the topology of the communication hardware of the machine. The bisectional bandwidth more accurately predicts the performance of the global communication kernels. Modeling the performance of these kernels require knowledge of the specific algorithm that is used, which in turn depends on the network topology. This is beyond the scope of this tutorial.

6.5 Data locality

Data locality is of utmost importance for high parallel performance. All the data locality concepts discussed in the serial part can be taken over in a straightforward way to the parallel case if we just think of the global memory of a distributed memory computer as the highest level in the memory hierarchy of the computer. Having good data locality then means in the case of a distributed memory parallel machine that most of the data that are needed by one processor are available in the memory physically attached to that processor. Since the access to data located on other processors necessitates interprocessor communication, good data locality will minimize this slow component of your code. In general, one should always try to predict whether it is not faster to replicate some result on all the processors instead of sending it around. In the case of an IBM SP2, we see for instance that we can hope under the most favorable circumstances for a bandwidth of approximately 68 MBytes/second. Since the SP2 is running at 66MHz, this means that we can feed one double precision number into the processor every 4 cycles. In many cases the calculation of each number takes less than 4 cycles.

6.6 Load balancing

The concept of load balancing is a rather trivial one, but nevertheless its effect on performance is significant. Ideally, in a parallel program the distribution of the total workload among the different processing elements would be perfectly even. If the distribution is not balanced then some processing elements will sit idle while others are still doing work. For example, if one processing element has a workload that is twice as big as the workload of all the other processing elements then all these processing elements will sit idle half of the time and the overall efficiency is nearly cut into half compared with the case when perfect load balancing can be achieved. The strategy for distributing the workload is usually motivated by the problem to be solved. If you do for instance a calculation on a homogeneous grid you will probably assign a sub-grid (each containing the same number of grid points) to each processing element. If you are doing a molecular dynamics calculation you might want to assign a certain number of atoms to each processor. In this case, it might however turn out that the workload associated with different atoms varies considerably because the number of nearest neighbors could be different for each atom. In this case a so-called pool of tasks administered by a master processor could be more advantageous. The master processor hands out tasks to each worker processor. Once the worker processor sends back its result it immediately gets the next task from the master.
6.7 Coarse-grain parallelism against fine-grain parallelism

A coarse-grain parallel program implies that a relatively large computational subtask is assigned to each processor. Each processing element then will do computations for quite a while before entering in a communication step, or in other words the computation/communication ratio is large. Fine-grain parallelism is the opposite case, involving short computation periods frequently interrupted by communication. Fine-grain parallelism is typically obtained by parallelizing at the loop level. With a fine-grain program you might get reasonable speedup on a shared memory machine, but not necessarily on a distributed memory architecture. If you want to obtain good speedups on a wide range of parallel computers you therefore need a coarse-grain program. Coarse-grain parallelism is essential to obtain good speedups on massively parallel computers (very large number of processors), as this kind of computers are distributed memory machines. The suitability of several parallel computers for fine-grain applications is shown in Figure 8. A large computational task was artificially subdivided into subtasks, whose length is shown on the x-axis. On an ideal parallel computer, where communication is infinitely fast and latency is zero, the total elapsed time shown on the y-axis would be a constant.

![Graph showing the dependency of the total elapsed time for a constant workload on the length of the time interval into which the total time is subdivided. The timings are shown for a 64 processor configurations on a IBM SP2, Cray T3E and SGI Origin computer.](image)

The communication requirements for this program ("grain_mpi.f", listed in the appendix) are really minimal. In realistic programs the communication requirements will be much larger and the smallest possible grain size that gives a reasonable performance in our example will be a lower bound for any realistic application. Very large differences can be seen for the three parallel computers tested. On the IBM SP2 grain sizes smaller than a tenth of a second lead already to a serious performance deterioration, whereas one can go down to grain sizes of 1/1000 of a second on the Cray T3E.
7 Epilog, acknowledgments and disclaimer

This tutorial was not meant to be an introduction to computer science neither to advanced programming. We just tried to give enough information concerning computer architectures and programming techniques in order to allow computational scientists to write efficient code. After having digested the information in this tutorial and after having gained some optimization experience yourself, you should be able to improve the performance of your programs. The techniques presented here will certainly outperform what you would obtain by using automatic preprocessors and parallelization tools without understanding the underlying principles. At the same time you will be able to discern weak points in certain computer architectures and thus become a more educated buyer of hardware.

S.G thanks Armin Schumacher and Armin Burkhardt for their technical help and Juerg Hutter for interesting discussions. A.H. thanks Olaf Lubeck for very useful discussions and comments. We acknowledge the access to the SP2 supercomputers at the Cornell Theory Center and Lawrence Livermore National Laboratory, to the Cray T3E supercomputer of the Max-Planck Rechenzentrum in Garching and to the SGI Origin 2000 at LANL.

We acknowledge the use of computational resources at the Advanced Computing Laboratory, Los Alamos National Laboratory, and support from the U.S. Department of Energy under Contract No. W-7405-ENG-36.

This tutorial was prepared as an account of work partially sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. References herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency herein.
8 Appendices

The programs listed in the Appendices are available via anonymous ftp at the site "parrix1.mpi-stuttgart.mpg.de" in the directory "/pub/outgoing/PERFORMANCE_OPTIMIZATION"

8.1 Timing routine for BLAS library

C Speed of different level BLAS routines. A simple matrix times matrix
C multiplication is done directly and by decomposing it into nd matrix
C times vector multiplications or nd**2 scalar products.

program blaslevels
implicit real*8 (a-h, o-z)
parameter(nd=1000)
dimension a(nd,nd),b(nd,nd),c(nd,nd)
do 15,j=1,nd
do 15,i=1,nd
a(i,j)=1.d0
b(i,j)=2.d0
c(i,j)=3.d0
15 continue
c level 3:
t1=mclock()*1.d-2
call DGEMM('N','N',nd,nd,nd,1.d0,a,nd,b,nd,0.d0,c,nd)
t2=mclock()*1.d-2
time_sec=(t2-t1)
speed_Mflops=(2*nd**3/time_sec)*1.d-6
cycle=time_sec*cyclehertz/(2*nd**3)
write(6,*)'DGEMM RESULTS ------------------'
write(6,*)'SPEED (MFLOPS)',speed_Mflops

c level 2:
t1=mclock()*1.d-2
do i=1,nd
call DGEMV('N',nd,nd,1.d0,a,nd,b(1,i),1,0.d0,c(1,i),1)
enddo
t2=mclock()*1.d-2
time_sec=(t2-t1)
speed_Mflops=(2*nd**3/time_sec)*1.d-6
cycle=time_sec*cyclehertz/(2*nd**3)
write(6,*)'DGEMV RESULTS ------------------'
write(6,*)'SPEED (MFLOPS)',speed_Mflops

c level 1:
t1=mclock()*1.d-2
do j=1,nd
do i=1,nd
c(i,j)=DDOT(nd,a(1,i),1,b(1,j),1)
enddo
enddo

end

8.2 Program for memory testing

program memory_test

    implicit real*8 (a-h,o-z)
    parameter(nexp=22,nn=2**nexp,nrep=10)

    c IBM 590 (peak 265 MFLOPS)
    parameter(cyclehertz=66.d6)

dimension x(nn+81,2),
& time_sec(nexp,nexp),speed_Mflops(nexp,nexp),cycle(nexp,nexp)

do 15,i=1,nn
15   x(i,1)=1.d0
   t1=1.d0
   t2=2.d0
   t3=3.d0
   t4=4.d0
   t5=5.d0
   t6=6.d0
   t7=7.d0
   t8=8.d0

c STORE TESTING-----------------------------------------------

do 100,il=3,nexp
   nl=2**il
   do 100,ij=1,max(il-7,1)
      jump=2**(ij-1)
      nit=nrep*jump*(nn/nl)
      t1=mclock()*1.d-2
   do 10, it=1,nit
      call sub1(nl,jump,x,t12,t34,t56,t78)
10      continue
   t2=mclock()*1.d-2
   time_sec(il,ij)=(t2-t1)
   nflops=nrep*nn
   speed_Mflops(il,ij)=(nflops/time_sec(il,ij))*1.d-6
   cycle(il,ij)=time_sec(il,ij)*cyclehertz/(nflops)
100   continue
write(6,*) 'MEMORY_TEST LOAD results'
write(6,*) 'CYCLES'
do il=3,nexp
21 format(1x,i2,1x,20(1x,f5.1))
write(6,21) il,(cycle(il,ij),ij=1,max(il-7,1))
enddo

c LOAD TESTING----------------------------------------------
do 200,il=3,nexp
   nl=2**il
   do 200,ij=1,max(il-7,1)
   jump=2**(ij-1)
   nit=nrep*jump*(nn/nl)
   t1=mclock()*1.d-2
   do 20, it=1,nit
      continue
   t2=mclock()*1.d-2
   time_sec(il,ij)=(t2-t1)
   nflops=nrep*nn
   speed_Mflops(il,ij)=(nflops/time_sec(il,ij))*1.d-6
   cycle(il,ij)=time_sec(il,ij)*cyclehertz/(nflops)
200 continue

c CALL subs(nl,jump,x(1,1),x(1,2))
write(6,*) 'MEMORY_TEST STORE results'
write(6,*) 'CYCLES'
do il=3,nexp
write(6,21) il,(cycle(il,ij),ij=1,max(il-7,1))
enddo

c COPY TESTING----------------------------------------------
do 300,il=3,nexp
   nl=2**il
   do 300,ij=1,max(il-7,1)
   jump=2**(ij-1)
   nit=nrep*jump*(nn/nl)
   t1=mclock()*1.d-2
   do 30, it=1,nit
      continue
   t2=mclock()*1.d-2
   time_sec(il,ij)=(t2-t1)
   nflops=2*nrep*nn
   speed_Mflops(il,ij)=(nflops/time_sec(il,ij))*1.d-6
   cycle(il,ij)=time_sec(il,ij)*cyclehertz/(nflops)
300 continue

c CALL subs(nl,jump,x(1,1),x(1,2))
write(6,*) 'MEMORY_TEST COPY results'
write(6,*) 'CYCLES'
do il=3,nexp
write(6,21) il,(cycle(il,ij),ij=1,max(il-7,1))
enddo
end
subroutine subs(n,jump,x,t1,t2,t3,t4,t5,t6,t7,t8)
  implicit real*8 (a-h,o-z)
  dimension x(n)

  if (jump.eq.1) then
    do 15,i=1,n-8+1,8
      x(i+0)=t1
      x(i+1)=t2
      x(i+2)=t3
      x(i+3)=t4
      x(i+4)=t5
      x(i+5)=t6
      x(i+6)=t7
      x(i+7)=t8
    continue
  15 continue
  else
    do 10,i=1,n-8*jump+1,8*jump
      x(i+0*jump)=t1
      x(i+1*jump)=t2
      x(i+2*jump)=t3
      x(i+3*jump)=t4
      x(i+4*jump)=t5
      x(i+5*jump)=t6
      x(i+6*jump)=t7
      x(i+7*jump)=t8
    continue
  10 continue

  return
  end

subroutine subl(n,jump,x,t12,t34,t56,t78)
  implicit real*8 (a-h,o-z)
  dimension x(n)
  t12=0.d0
  t34=0.d0
  t56=0.d0
  t78=0.d0
  if (jump.eq.1) then
    do 15,i=1,n-8+1,8
      s1=x(i+0)
      s2=x(i+1)
      t12=t12+s1*s2
      s3=x(i+2)
      s4=x(i+3)
      t34=t34+s3*s4
      s5=x(i+4)
      s6=x(i+5)
      t56=t56+s5*s6
      s7=x(i+6)
  15 continue
s8=x(i+7)
t78=t78+s7*s8
15 continue
else
do 10,i=1,n-8*jump+1,8*jump
s1=x(i+0*jump)
s2=x(i+1*jump)
t12=t12+s1*s2
s3=x(i+2*jump)
s4=x(i+3*jump)
t34=t34+s3*s4
s5=x(i+4*jump)
s6=x(i+5*jump)
t56=t56+s5*s6
s7=x(i+6*jump)
s8=x(i+7*jump)
t78=t78+s7*s8
10 continue
endif
return
end

subroutine subc(n,jump,x,y)
implicit real*8 (a-h,o-z)
dimension x(n),y(n)
if (jump.eq.1) then
do 15,i=1,n-8+1,8
x(i+0)=y(i+0)
x(i+1)=y(i+1)
x(i+2)=y(i+2)
x(i+3)=y(i+3)
x(i+4)=y(i+4)
x(i+5)=y(i+5)
x(i+6)=y(i+6)
x(i+7)=y(i+7)
15 continue
else
do 10,i=1,n-8*jump+1,8*jump
x(i+0*jump)=y(i+0*jump)
x(i+1*jump)=y(i+1*jump)
x(i+2*jump)=y(i+2*jump)
x(i+3*jump)=y(i+3*jump)
x(i+4*jump)=y(i+4*jump)
x(i+5*jump)=y(i+5*jump)
x(i+6*jump)=y(i+6*jump)
x(i+7*jump)=y(i+7*jump)
10 continue
endif
return
end
8.3 Program that should run at peak speed

```fortran
program peak

implicit real*8 (a-h,o-z)
parameter(nexp=14,nn=2**nexp,nrep=400)
parameter(cyclehertz=120.d6)
dimension &
time_sec(nexp),speed_Mflops(nexp),cycle(nexp)
dimension v(4)

do 100,il=0,nexp
v(1)=1.d-26
v(2)=1.d-26
v(3)=1.d-26
v(4)=1.d-26
nl=2**il
nit=nrep*(nn/nl)
t1=mclock()*1.d-2
 t1=second()
do 10,it=1,nit
call sub_1(nl,v(1),v(2),v(3),v(4))
continue
C t2=mclock()*1.d-2
 t2=second()
time_sec(il)=(t2-t1)
nflops=32*nrep*nn
 speed_Mflops(il)=(nflops/time_sec(il))*1.d-6
cycle(il)=time_sec(il)*cyclehertz/(nflops)
continue

write(6,*) 'CYCLES, SPEED (MFLOPS), TIME (sec)'
do il=0,nexp
21 format(1x,i2,1x,i9,f6.2,f8.1,e10.3)
write(6,21) il,2**il,cycle(il),speed_Mflops(il),time_sec(il)
enddo
write(6,*) v
end

subroutine sub_1(n,x1,x2,x3,x4)
c "normally" written version, gives peak speed for instance on IBM POWER2
implicit real*8 (a-h,o-z)
do 10,i=1,n
y1=x1 + .500007500110d-05*x2 + .100001500023d-04*x3
y2=x2 + .500007500110d-05*x1 + .100001500023d-04*x4
y3=x3 + .100001500023d-04*x1 + .500007500110d-05*x4
y4=x4 + .100001500023d-04*x2 + .500007500110d-05*x3
```

x1=y1 + .500007500110d-05*y2 + .100001500023d-04*y3
x2=y2 + .500007500110d-05*y1 + .100001500023d-04*y4
x3=y3 + .100001500023d-04*y1 + .500007500110d-05*y4
x4=y4 + .100001500023d-04*y2 + .500007500110d-05*y3

10 continue

return
end

subroutine sub_2(n,x1,x2,x3,x4)

10 continue

return
end

subroutine sub_2(n,x1,x2,x3,x4)
c hand optimized version giving also peak speed on SGI R10000

implicit real*8 (a-h,o-z)
do 10,i=1,n

y1=x1 + .500007500110d-05*x2
y2=x2 + .500007500110d-05*x1
y3=x3 + .100001500023d-04*x1
y4=x4 + .100001500023d-04*x2

y1=y1 + .100001500023d-04*x3
y2=y2 + .100001500023d-04*x4
y3=y3 + .500007500110d-05*x4
y4=y4 + .500007500110d-05*x3

x1=x1 + .500007500110d-05*y2
x2=x2 + .500007500110d-05*y1
x3=x3 + .100001500023d-04*y1
x4=x4 + .100001500023d-04*y2

x1=x1 + .100001500023d-04*y3
x2=x2 + .100001500023d-04*y4
x3=x3 + .500007500110d-05*y4
x4=x4 + .500007500110d-05*y3

10 continue

return
end

subroutine sub_3(n,x1,x2,x3,x4)
c software pipelined by hand for T3E processor, nevertheless peak speed not attained

implicit real*8 (a-h,o-z)

s1=.500007500110d-05*x2
s2=.500007500110d-05*x1
s3=.100001500023d-04*x1
s4=.100001500023d-04*x2
s5=.100001500023d-04*x3
s6=.100001500023d-04*x4
s7=.500007500110d-05*x4
s8=.500007500110d-05*x3

y1=x1 + s1
\[ y_2 = x_2 + s_2 \\
\[ y_3 = x_3 + s_3 \\
\[ y_4 = x_4 + s_4 \\
\[ y_1 = y_1 + s_5 \\
\[ y_2 = y_2 + s_6 \\
\[ y_3 = y_3 + s_7 \\
\[ y_4 = y_4 + s_8 \\
\]

\[ s_1 = .500007500110d-05 \times y_2 \\
\[ s_2 = .500007500110d-05 \times x_1 \\
\[ s_3 = .100001500023d-04 \times x_1 \\
\[ s_4 = .100001500023d-04 \times x_2 \\
\[ s_5 = .100001500023d-04 \times x_3 \\
\[ s_6 = .100001500023d-04 \times x_4 \\
\[ s_7 = .500007500110d-05 \times x_4 \\
\[ s_8 = .500007500110d-05 \times x_3 \\
\]

\[ x_1 = y_1 + s_1 \\
\[ x_2 = y_2 + s_2 \\
\[ x_2 = .500007500110d-05 \times x_1 \\
\[ x_3 = y_3 + s_3 \\
\[ x_3 = .100001500023d-04 \times x_1 \\
\[ x_4 = y_4 + s_4 \\
\[ x_4 = .100001500023d-04 \times x_2 \\
\]

\[ x_1 = x_1 + s_5 \\
\[ s_5 = .100001500023d-04 \times x_3 \\
\[ x_2 = x_2 + s_6 \\
\[ s_6 = .100001500023d-04 \times x_4 \\
\[ x_3 = x_3 + s_7 \\
\[ s_7 = .500007500110d-05 \times x_4 \\
\[ x_4 = x_4 + s_7 \\
\[ s_8 = .500007500110d-05 \times x_3 \\
\]

do 10, i=2, n-1

\[ y_1 = x_1 + s_1 \\
\[ s_1 = .500007500110d-05 \times y_2 \\
\[ y_2 = x_2 + s_2 \\
\[ s_2 = .500007500110d-05 \times y_1 \\
\[ y_3 = x_3 + s_3 \\
\[ s_3 = .100001500023d-04 \times y_1 \\
\[ y_4 = x_4 + s_4 \\
\[ s_4 = .100001500023d-04 \times y_2 \\
\]

\[ y_1 = y_1 + s_5 \\
\[ s_5 = .100001500023d-04 \times y_3 \\
\[ y_2 = y_2 + s_6 \\
\[ s_6 = .100001500023d-04 \times y_4 \\
\[ y_3 = y_3 + s_7 \\
\[ s_7 = .500007500110d-05 \times y_4 \\
\[ y_4 = y_4 + s_8 \\
\[ s_8 = .500007500110d-05 \times y_3 \\
\]
x1=y1 + s1
s1=.500007500110d-05*x2
x2=y2 + s2
s2=.500007500110d-05*x1
x3=y3 + s3
s3=.100001500023d-04*x1
x4=y4 + s4
s4=.100001500023d-04*x2

x1=x1 + s5
s5=.100001500023d-04*x3
x2=x2 + s6
s6=.100001500023d-04*x4
x3=x3 + s7
s7=.500007500110d-05*x4
x4=x4 + s8
s8=.500007500110d-05*x3

10 continue

y1=x1 + s1
s1=.500007500110d-05*y2
y2=x2 + s2
s2=.500007500110d-05*y1
y3=x3 + s3
s3=.100001500023d-04*y1
y4=x4 + s4
s4=.100001500023d-04*y2

y1=y1 + s5
s5=.100001500023d-04*y3
y2=y2 + s6
s6=.100001500023d-04*y4
y3=y3 + s7
s7=.500007500110d-05*y4
y4=y4 + s8
s8=.500007500110d-05*y3

x1=y1 + s1
x2=y2 + s2
x3=y3 + s3
x4=y4 + s4
x1=x1 + s5
x2=x2 + s6
x3=x3 + s7
x4=x4 + s8

return
end

46
8.4 Program to test suitability of parallel computers for fine grained applications

program grain_mpi
    implicit real*8 (a-h,o-z)
    include 'mpif.h'
    parameter (ntotal=2**24)

c initialize MPI
    write(6,*) 'start mpi'
    call MPI_INIT(ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD,iproc,ierr)
    call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
    write(6,*) 'mpi started',iproc,nproc
    if (iproc.eq.0) write(6,*),'nproc=',nproc
&    write(6,*),'the total workload is always the same, so on a
&    perfect parallel machine the total time would be constant'

    nrep=1
    do 200,isub=1,15
    nrep=nrep*2
    ndat=ntotal/nrep

    tl=mclock()
    do 100,irep=1,nrep

    call MPI_BCAST(ndat,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

    tt=1.d0
    do 10,i=1,ndat
    tt=sqrt(tt+t.d-1)
     10 continue

    call MPI_REDUCE(tt,ttsum,1,
    1 MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,ierr)

    continue
    t2=mclock()
    time=(t2-tl)/100.d0
    if (iproc.eq.0) then
    write(6,*),'repetitions,total time(sec)',nrep,time
    endif
    200 continue

    call MPI_FINALIZE(ierr)

end
8.5 MPI timing routine

subroutine timing(category,action,nflop,iproc,nproc)
c subroutine for performance analysis of parallel MPI programs
c possible actions are:
c 'INI' to initialize all the counters at the beginning of program
c 'STR' to (re)start accumulating counters for a certain class
c 'END' to end accumulating counters for a certain class
c 'FIN' to print the performance statistics
c The timing categories are defined by the user in the data statement below.
c Only one category can be active at a time
nc NFLOP: number of floating point operations or communication operations
nc The performance analysis is written to the file 'time.prc'
c written by S. Goedecker

implicit real*8 (a-h,o-z)
include 'mpif.h'
character*6 category,cats
character*3 action
logical init
parameter(ncat=2)
dimension cats(ncat),flops(ncat),timesum(ncat+1),
1 wrktime(ncat+1),wrksquare(ncat),wrkflops(ncat),wrkflop2(ncat)
save cats,time0,init,timesum,flops,total0

c define your performance categories (set NCAT to correct value)
data cats / 'SUMS' , 'TRANS' /

if (action.eq.'INI') then
  do 11,i=1,ncat
  total0=mclock()
  flops(i)=0.d0
11  timesum(i)=0.d0
  init=.false.
else if (action.eq.'FIN') then
  timesum(ncat+1)=mclock()-total0
  if (nproc.gt.1) then
    call MPI_REDUCE(timesum,wrktime,ncat+1,
1                   MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,ierr)
  else
    do 488,i=1,ncat+1
    wrktime(i)=timesum(i)
  endif
  do 35,i=1,ncat
35    timesum(i)=timesum(i)**2
  if (nproc.gt.1) then
    call MPI_REDUCE(timesum,wrksquare,ncat,
1                   MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,ierr)
  call MPI_REDUCE(flops,wrkflops,ncat,
1                   MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,ierr)
  else
124    do 488,i=1,ncat
    wrksquare(i)=timesum(i)
enddo
wrkflops(i)=flops(i)
endf

do 45,i=1,ncat
flops(i)=flops(i)**2
if (nproc.gt.i) then
  call MPI_REDUCE(flops,wrkflop2,ncat, MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,ierr)
  else
  do 874,i=1,ncat
  endif
wrkflop2(i)=flops(i)
endf

if (iproc.eq.0) then
  open(unit=60,file='time.prc',status='unknown')
  write(60,*)
  &'CATEGORY , TIME(sec) + dev , PERCENT , FLOPS + dev , SPEED (MFlop)'
  format(1x,a,2x,e10.3,e9.2,2x,f6.3,3x,e10.3,2x,e9.2,2x,e10.3)
  sum=0.d0
  do 22,i=1,ncat
  sum=sum+wrktime(i)
  do 20,i=1,ncat
  t1=wrktime(i)
  t2=sqrt(-t1**2+nproc*wrksquare(i))
  t3=wrktime(i)/sum
  t4=wrkflops(i)
  t5=sqrt(-t4**2+nproc*wrkflop2(i))
  t6=1.d-4*wrkflops(i)/wrktime(i)
  write(60,25) cats(i),1.d-2*t1/nproc,1.d-2*t2/nproc,100.d0*t3,1.d0*t4/nproc,t5/nproc,t6
  write(60,25) 'SUM ',1.d-2*sum/nproc
  write(60,25) 'TOTAL',1.d-2*wrktime(ncat+1)/nproc
  write(60,*) nproc,'processors'
  close(60)
endif

else

do 100,i=1,ncat
if (category.eq.cats(i)) then
  ii=i
  goto 200
endif
continue
print*, 'TIMING CATEGORY',category, ' NOT DEFINED'
print*, 'ACTION ',action
stop 'TIMING CATEGORY NOT DEFINED'
continue
if (action.eq.'STR') then
if (init.neqv..false.) then
  print*, cats(ii),': TIMING RESTARTED BEFORE ENDED'
stop
endif

49
time0=mclock()
init=.true.
else if (action.eq.'END') then
  if (init.neqv..true.) then
    print*, cats(ii), ':NOT STARTED BEFORE ENDED'
  stop
endif
flops(ii)=flops(ii)+nflop
timesum(ii)=timesum(ii)+mclock()-time0
init=.false.
else
  stop 'TIMING ACTION UNDEFINED'
endif
endif
end