Bull. Nov. Comp. Center, Comp. Science, 22 (2005), 23–40

Acceleration of the numerical simulation programs*

A. Cherkasov, M. Gorodnichev, S. Kireev, V. Markova, A. Merkulov

Abstract. The basic means and methods to increase the efficiency of the sequential numerical simulation programs are considered. The main components of the modern hardware and software which affect the programs efficiency are discussed. The most important guidelines and advices for application programmers are outlined.

1. Introduction

Many researchers, when developing the numerical simulation programs, miss the possibility to use simple but effective means to increase the efficiency of their programs. Here are factors that affect the program performance: the architecture of the target microprocessor; the existence and the use of specialized libraries; tuning of the compilation process. In this paper, the most productive and effective ways for taking into account the above factors are shown. Note, that we put aside an important question of choosing the algorithm of problem solution. Only technical aspects of the algorithm implementation are considered. This paper is based on the experience of the researches of the ICM&MG, the Supercomputer Software Department, and the Siberian Supercomputer Center in the field of development and optimization of numerical simulation programs.

The paper consists of four parts. The first part contains a brief description of the main components and features of the microprocessor architecture with Opteron 240 and Alpha 21264 processors used for illustration. The second part is devoted to programming, the main "manual" optimization techniques being given. The usage of specialized libraries is discussed in the third part. The fourth part describes compilation of programs and automatic optimization with compilers.

2. The architecture of the superscalar microprocessors

A set of instructions of superscalar processors contains no explicit instructions for parallel processing. Detection of independent instructions and their scheduling among execution units are carried out at a hardware level during

*Supported by the grants: the Dutch-Russian Project "High Performance Simulation on the Grid", Contract NWO-RFBS 047.016.007; the Dutch-Russian Project "Project Investigation of Plasma Induced Cluster Formation and Thin Film Deposition", Contract NWO-RFBS 047.016.018, RFBR 03-07-90302 and financed by the SSCC.

the run time. The processor core contains a number of execution units to organize parallel execution of independent operations. Each execution unit is able to carry out a certain class of operations. Units are staged and can also be several units bearing the same functions. The practice shows that the quantity of executing commands is less, as a rule, at any moment, than that of the execution devices. This is caused by the following reasons:

- the low rate of loading the new commands and the data from memory and storing the results of the work;
- the data and the control flow dependencies between neighboring commands.

The gap between processor speeds and the speed of memory access is tremendous and is the problem for all the existent computational systems. However, the size of this gap differs through the systems and this must be taken into account when planning hardware acquisitions. On the other hand, the problem of slow memory access is always and mainly alleviated by caching the most used data. And the caching system is the most important thing to take into account when optimizing your program. To eliminate data and control flow dependencies, the so-called "dynamic" execution of instructions is used, and the programmer should help the processor to do this work.

2.1. Dynamic execution of commands

Superscalar processing is possible due to the dynamic execution of commands and consists of the following:

- *Branch prediction.* When a conditional branch is encountered, the further direction of the control flow is predicted not waiting for the evaluation of the condition.
- *Speculative execution.* Commands are fetched and executed from the predicted branch.
- *Registers renaming.* This technique allows us to remove data and register dependencies invisible to the software.
- *Out-of-order execution.* All the instructions are dispatched to execution units as soon as their operands are loaded and execution units are free.

2.2. The memory subsystem organization

The hierarchical memory architecture is based on the locality principle of the memory references with respect to space and time, which allows most frequently used data to be placed into the fast memory of small volume. The

Parameter	Alpha 21264	Opteron 240
Architecture	RISC, 4 IPC	CISC with RISC core 3×86 PC
L1 ICache	64 Kb, 2 way-set associative, 64 b line	64 Kb, 2 way-set associative, 64 b line
L1 D-Cache	64 Kb, 2 way-set associative, 3 cycles	64 Kb, 2 way-set associative, 3 cycles
L2 Cache (shared)	1 Mb, external, joint, direct mapped, 16 cycles	1 Mb, on-die, joint, 16 way-set associative, 16 cycles
TLB	128 entries, fully associative	40 entries, fully associative
Program registers	32 unteger, 32 float	16 integer, 8 float, 16 SSE
Register life	72 float, 80 integer registers, 4 read, 6 write ports	120 integer, 9 read, 8 write ports, 120 float, 5 read 5 write ports
Frequency	833 MHz	1400 MHz
System bus	64 Mb, 3.2 Mb/s	128 b, 6.4 Mb/s
Functional units	2 integer, 2 float, 2 load/store	3 integer, 3 float, 3 load/store

result of this is that the memory access average time exceeds the time of access to this fast memory just a little if the programmers do not violate this principle. Alpha 21264 and Opteron 240 microprocessors have a convectional memory organization: a register file, separate level 1 data and instruction caches, the joint level 2 cache, the RAM and disk memory (parameters of the microprocessors are specified in the table).

The first level data caches of these processors are 2-way set-associative, of 64 K size, with cache-line of 64 b (Figure 1). This means that the cache consists of 512 sets with 2 lines per set, and the physical address space is broken into blocks of 64 bytes. Each memory reference to some memory location causes the block including this location, to be entirely loaded into the cache. Nine low bits of a block number determine the set, into one of two lines of which a given block can be placed. It is obvious, that many memory

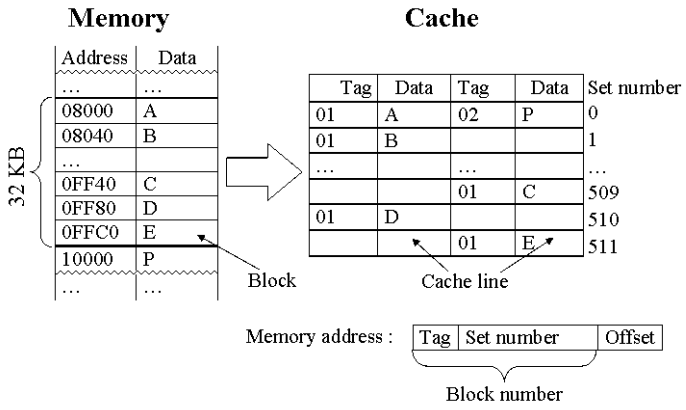

Figure 1. Cache memory organization

blocks correspond to the same set and the stride between these blocks equals 512×64 bytes. Which of two cache lines is used for a particular block load is determined by a certain replacement algorithm [1]. A tag, constituting of higher bits of the block number, is used to identify which of the possible memory blocks is now taking the cache line.

The caching system uses the buffer of associative translation (TLB) for the fast translation of virtual addresses. The TLB stores the numbers of memory pages, which have been recently accessed. The size of the buffer determines the optimal quantity of actively used pages.

As it follows from the description of processor and memory architectures and their implementation (it is known, for example, that memory implementations provide special mechanisms to speed-up, particularly, sequential accesses to memory locations) the following architectural parameters considerable affect the program performance:

- Quantity of program registers, i.e., registers into which local variables of the program are mapped.
- Parameters of the cache memory (length of cache-line, size of the cache, size of the TLB).

It is obvious that if the programmer ignores these parameters and the principles (such as locality of references) which are put into the basis of the architecture design, the program cannot run as fast as it could otherwise.

As it is hard for beginners to deal with the architectural features and physical implementation of memory in the best way, our purpose is to provide guidelines for optimal programming that should be simple and technological.

In the following sections, the most effective recommendations of program optimization are considered.

3. The effective use of arrays

Processing of large amounts of data is a distinctive feature of numerical modeling programs. Reducing the time spent for memory operations can be attained by the careful work with arrays in programs, i.e., good data layout and access order.

3.1. The effective use of cache memory. Taking into account the hierarchical memory organization and, particularly, cache parameters in most cases allows one to notably speed-up the program. First of all, a program should use memory by blocks not larger than the cache size in each separate block of the code (loop, procedure). Here is an example:

Matrix multiplication	Block matrix multiplication

`for(i=0; i<n; i++)`
`  for(j=0; j<n; j++)`
`  for(k=0; k<n; k++)`
`  c[i][k] += a[i][j]*b[j][k];`
	`for(ii=0; ii<n-1; ii=ii+nb)`
	`  for(jj=0; jj<n-1; jj=jj+nb)`
	`  for(kk=0; kk<n-1; kk=kk+nb)`
	`  for(i=ii; i<ii+nb; i++)`
	`  for(j=jj; j<jj+nb; j++)`
	`  for(k=kk; k<kk+nb; k++)`
	`  c[i][k] += a[i][j]*b[j][k];`

Three outer loops in the block matrix multiplication program choose nb×nb blocks of matrices, block by block, and three inner loops multiply these blocks. Speed-up can be expected when the sum of the sizes of three matrix blocks is smaller than the cache size. For example, when n = 500 (the element type is double) and the block size is 50×50 (the sum of the sizes is about 59KB) the block algorithm runs faster than the left one by 3% on Alpha and by 68% on Opteron. The example given demonstrates the general principle of optimization only and, surely, a more effective block algorithm could be proposed. The arrangement and addressing order does not much influence the speed while the program works with data located entirely in the cache. Hereinafter we mainly consider the effective processing of data, whose size exceeds that of the cache memory.

3.2. Arrangement of arrays in memory. The following recommendations are necessary to follow when arranging the data in arrays:

- place the data used in a computational block (loop, subroutine) as densely as possible;
- place elements of the basic types to the addresses that are divisible by the element size.

The first recommendation obviously follows from the memory hierarchy organization. In particular, the cache memory exchanges data with the main memory on a line-by-line basis. Thus, for the best efficiency each cache line, placed into the cache from the main memory should be entirely used. That is, if one element is used, the adjacent ones should be used as well (ideally, all the elements of the cache line). The second recommendation is called "natural alignment of data". It follows from the memory addressing peculiarities. Addressing of unaligned data can greatly decrease the performance. The ratio between the access speed of aligned and unaligned data differs for different processors and data types. For example, reading an unaligned array of doubles (8 bytes) is 60–70 times slower than that of an aligned array on the Alpha processor, and only about 10% on Opteron. Both the dense data allocation and data alignment are significant for the time of execution. However, the data alignment has a greater impact, so compilers always perform the alignment of elements of the basic types by moving them to the

next aligned address. As a result, there can be unused intervals in memory. In some cases, the aggregate size of these unused intervals can be rather big. Aligning data by a compiler may be disabled, but the time of execution can greatly increase. These are recommendations, helping one to keep alignment and dense allocation of data:

- When declaring local variables or structure elements in C or elements of a common block in Fortran, place them in descending order according to their sizes;
- When using an array of structures in C, make the size of the structure to be multiple of the size of its biggest unit. One can reach this goal in two ways:
  - Split the structure to the several sub-structures of optimized size;
  - Extend the structure to the required size with additional elements. This way is only meaningful if additional elements are not only extend the structure (this can be done automatically by a compiler), but are meaningfully used.

There are two rather artificial ways of how unaligned data can appear. It is unlikely that the ordinary programmer will use such constructs in programs but he ought to know them:

- In C, the explicit operations with pointers are allowed. Moving the pointer to the address that is not a multiple of the size of a unit and its subsequent usage will lead to unaligned memory access;
- In Fortran-77, one can have data unaligned passing the array of units of a smaller size to the subroutine and accepting it in a subroutine as array of units of a greater size.

3.3. Going through data in memory. It is a principle of data locality that among others, common principles determine the effectiveness of successive processing of large sequences of data elements (when elements are fetched one by one, processed and/or written back and the like). In particular, the more elements of the loaded cache line are used, the higher is the effectiveness. Moreover, if the cache lines are fetched sequentially from main memory, the hardware prefetch mechanism starts to work, loading the cache lines beforehand and the access time reduces. Thus, the best (simple enough and efficient) way to walk through the array of elements in memory is to access them sequentially in straight order. In some cases, the access time can be more decreased using the software prefetch. The general recommendations on the accessing arrays are:

- access to the elements of arrays in the straight sequential order as they are placed in memory;

- avoid the cache trashing.

Further we will consider these recommendations in detail and outline the most common cases.

3.4. Sequential access to elements of array. Applying the straight sequential access to the elements of one-dimensional array is obvious: elements are accessed from the first to the last with Step 1. Ambiguity arises when considering a multi-dimensional array. It is clear that the order of accesses to elements is determined by the order of loops nesting. For C language, the order of nesting of loops should be the same as the corresponding dimensions of an array; for Fortran language, the order should be reverse:

In C	In Fortran

`for(i=0; i<N; i++)`
`  for(j=0; j<M; j++)`
`  X[i][j] = 1;`
	`do j=1,M`
	`  do i=1,N`
	`  X(i,j) = 1`
	`  enddo`
	`enddo`

Here are the two most useful ways of making the access to elements of multidimensional array be sequential:

- Loop interchange. If allowed by the algorithm, loops should be interchanged in such a way that allows one to perform a straight sequential access to the array elements;
- Changing the order of elements in the array. If the loop interchange is impossible, the reordering of array dimensions may help.

3.5. The cache trashing. Sometimes, the sequential access to the elements of an array is not possible due to the algorithm peculiarities. So, it is important to know special cases of access order that affect performance. One of the most significant cases is cache trashing. It arises when the elements placed with a "bad" stride are accessed in program nearby, and their count is more than the number of cache lines in a set in cache memory. The stride is "bad" if it is divisible by the number of sets in the cache multiplied by the cache line size, or, which is the same, it is divisible by a cache size divided by the number of lines in the set. For example, in Alpha and Opteron processors, the sequential access to more than two data items placed with a stride multiple of 32 KB will cause cache trashing. There are some the most common examples when the cache trashing takes place:

- For several arrays, a distance between elements with the same indexes may be "bad":

```
double a[4096], b[4096], c[4096];
for(i=0; i<4096; i++) c[i] = a[i] + b[i];
```

- In a multidimensional array, the size of the first dimension, i.e., a distance between the neighbors of the other dimension may be "bad":

```
double a[1000][4096]; for(i=1; i<999; i++)
  for(j=1; j<4095; i++)
    a[i][j]=a[i][j]+a[i-1][j]+a[i+1][j]+a[i][j-1]+a[i][j+1];
```

For the majority of processors, a "bad" size equals power 2. Thus, it is a reason for paying a special attention to the arrays, whose dimension equals power 2, as they are a potential cause of the cache trashing. Simplest ways to avoid the cache trashing are the following:

- changing the order in which elements are accessed;
- changing the distance between elements.

Methods of changing the order of accesses were considered in the beginning of this section. The distance between conflicting elements can be changed by the following:

- If the elements of distinct arrays conflict, a distance can be changed by inserting other variables or arrays between them;
- If the elements of one multidimensional array conflict, the size of one of the dimensions can be increased. Additional elements may be used for keeping other program data.

Note that the best change of the distance is a cache line size (or greater).

4. Optimization of computations in loops

Computations in loops take up most of the time in numerical simulation programs. Hence, a modification of the loop body can result in a notable change of execution time of the program. In the sequel, the ways of loop optimization are considered.

4.1. Common optimization techniques. There are optimization techniques that are applicable to any computations, not only to loops. Within loops, their significance can greatly increase.

1. *Reducing the number of operations.* Two most frequently used methods are the following:

   - arithmetical transformations;
   - removing of redundant computations.

   It is necessary to say that many numerical methods use coefficients in the formulas, and these coefficients may be computed just once and then used many times.

2. *Replacing slow operations with faster ones.* Different operations take different processor time. For example, addition and subtraction take less time than multiplication, and multiplication takes less time than division and square root calculation. In certain cases long operations can be replaced with faster ones, thus attaining the same result in lesser time.
3. *Eliminating data dependence between operations.* To fully load all the functional units of the processor, the adjacent operations must be independent and able to be simultaneously computed. Changing the order of calculations in program can help making the adjacent operators independent.
4. *Redundancy reduction in control flow.*

In many cases the time spent for organization of the control flow, i.e., conditional and unconditional branches can be decreased due to only slight variations in algorithm.

4.2. Techniques for optimization of computations in loops. There are some optimization methods that are typical of loops. The latter take into account such important features as repeated computations. Let us consider the most widespread methods applicable in most cases of the loop optimization.

Common loop optimization techniques.

1. *Reducing the number of variables used in a loop.* During the loop execution, accesses to the same several variables happen repeatedly. To avoid unnecessary references to memory, it is preferable that all variables be placed in registers. As the count of program registers is limited, the count of variables in loop has not to exceed it.
2. *Eliminating subroutine calls.* Calling a subroutine and returning from it are not trivial operations that take a lot of processor time. As they are performed repeatedly, the time can significantly increase. If a subroutine is rather small, its code can be inserted in the place of a call. The present day compilers can do such an optimization automatically when the most aggressive optimization levels are used.
3. *Reducing the number of array accesses in the loop body.* In most cases, elements of an array are indexed with loop counters. And often the nearest neighbors are used in each loop iteration. In this case, the array elements are read from memory more than once. Consider the way to avoid unnecessary references by saving values in local variables:

Variant 1	Variant 2

`for(i=1; i<n-1; i++)`
`  y[i] = a*x[i-1]+b*x[i]+c*x[i+1];`
	`x1=a[1]; x2=a[2];`
	`for (i=1; i<n-1; i++) {`
	`  x3=a[i+1];`
	`  y[i]=a*x1+b*x2+c*x3;`
	`  x1=x2; x2=x3;`
	`}`

Read operations: $3(n-2)$	Read operations: n

4. *Loop unrolling.* The optimization duplicates the loop body several times and, respectively, reduces the number of iterations. Program speed-ups due to a decrease of the number of conditional branches, and a larger quantity of instructions become contiguous thus allowing the compiler and the processor to reorder them in a proper manner. Also, some additional optimizations may become available. Here is an example:

Variant	Code	Reads

Source variant	`do i=1,n`
	`  do j=1,n`
	`  z(j,i) = a*x(i)+b*y(j)`
	`  enddo`
	`  enddo`
		$2n^2$

Inner loop unrolled 4 times	`do i=1,n`
	`  do j=1,n,4`
	`  axi = a*x(i)`
	`  z(j,i) = axi+b*y(j)`
	`  z(j+1,i) = axi+b*y(j+1)`
	`  z(j+2,i) = axi+b*y(j+2)`
	`  z(j+3,i) = axi+b*y(j+3)`
	`  enddo`
	`enddo`
		$n^2 + n^2/4$

Outer loop unrolled 4 times	`do i=1,n,4`
	`  do j=1,n`
	`  byj = b*y(j)`
	`  z(j,i) = a*x(i)+byj`
	`  z(j,i+1) = a*x(i+1)+byj`
	`  z(j,i+2) = a*x(i+2)+byj`
	`  z(j,i+3) = a*x(i+3)+byj`
	`  enddo`
	`enddo`
		$n^2 + n^2/4$

The inner loop unroll variant is preferable as elements of the array are sequentially accessed.

5. *Avoiding the dependence between loop iterations.* When the data dependence between iterations is present, the compiler is unable to perform most of effective optimizations (software pipelining, vectorization, parallelization, etc.). For example:

Variant 1	Variant 2

`do i=2,n`
`  x(i) = x(i-1) + x(i)`
`enddo`
	`t=x(1)`
	`do i=2,n`
	`  t = t + x(i)`
	`  x(i) = t`
	`enddo`

   In the source loop (left), every iteration may start execution only if the preceding iteration is finished, i.e., the *write* operation is performed. In the optimized loop (right), the data dependence is avoided, so addition in every iteration can be performed regardless of *write* operations in the preceding iterations.

The inner loop optimization. The inner loop is the most important one in the loop nest, as its body executes the greatest amount of times. So, the lesser execution time of inner loop, the lesser notably total time of program execution. These aspects have the most influence on the execution time of the inner loops:

1. *The size and complexity of the inner loop.* Enough big loop body gives the compiler and the dynamic processor core more freedom in reordering commands, thus bringing about the more effective use of the processor pipeline. But a too complex loop body is difficult for the compiler to understand and optimize
2. *The order of accesses to memory.* When an array is being accessed in the inner loop, the order of accesses has the strongest impact on performance. Hence it is obligatory to sequentially access arrays in the inner loops.

There are typical inner loop optimizations:

1. *Data prefetching* is moving it from the main memory to the cache memory before it is actually needed. Prefetching data may be done at the same time as some long computation in loop body. As the data needed for the computations on the current iteration have to be read before, prefetching data for the next iterations can be performed during computation. The optimal prefetch distance is 3–4 cache lines. The best way is to unroll the loop for the entire cache line be used per loop iteration. The ways of making data prefetching:

- Using software prefetch operations. Advantages: does not need to use additional variables; data out of the required array may be prefetched. Disadvantages: need to use assembly commands or special language extensions. Note that some compilers automatically insert the software prefetch operations into code when using special compiling options.
- Explicit reading an element from the required cache line to the local variable. The whole cache line would be prefetched. Advantages: standard language operations are used. Disadvantages: additional variables are required; data out of the required array can not be prefetched (several iterations must not perform prefetching in such way).

2. *Loop vectorization.* This optimization is platform dependent and can be realized only if a processor supports vector instructions. Each vector instruction performs the same (or not the same) operation on several data items. The inner loops are usually to be vectorized if their iterations are independent. The ways of performing vectorization:

   - automatic vectorization by compiler;
   - using low-level vector instructions in a high-level language extension;
   - using an assembly language.

4.3. Loop transformation. When performing many of the given optimization techniques standard loop transformations are often used. Some of the modern (mainly commercial) compilers may perform such transformations. Making automatic optimization greatly depends on complexity of code. Hence the Fortran compilers usually have greater abilities in transforming loops as the Fortran language is much simpler than C. The mostly used loop transformations listed below:

1. *Loop splitting and fusion.* When splitting a loop its body splits into several parts, each one being placed in its own loop. Fusion is a reverse operation;
2. *Loop distribution.* When distributing the loop splits into several loops with same bodies, which iterations joined compound the iterations of the source loop without repeating. Sometimes the reverse operation is also useful;
3. *Loop permutation* — changing the order of loops in a program;
4. *Loop interchange* — changing the nesting order of loops;
5. *Loop reversal* — making iterations in the reverse order.

5. Libraries

Using libraries is very advantageous: first, development time is decreased due to the use of the existing code, and second, the resulting programs are more effective since producers of libraries pay much attention to optimizing their subroutines. As an example, let us describe the BLAS (Basic Linear Algebra Subprograms) subroutine package (www.netlib.org/blas). The BLAS is a set of subroutines performing the basic vector and matrix operations. They constitute the basis of many libraries that provide solutions of linear algebra tasks, and the effectiveness of such libraries depends on the effectiveness of the BLAS implementation in use. The BLAS routines are divided into three levels: the first level routines operate on vectors, the second level routines operate on a vector and a matrix, and the third level routines operate on two matrices. A set of subroutines and their names are standardized in 2001 due to efforts of the BLAS Technical Forum. Producers of microprocessors develop their own mathematical libraries and include in these libraries implementations of the BLAS specially tuned to particular processors. The Intel offers the so-called Intel Math Kernel Library for its processors, there exists a Compaq Extended Math Library for the Alpha processors, and the AMD provides the AMD Core Math Library for Opteron processors. One can use the ATLAS package (www.netlib.org/atlas) to build the BLAS implementation automatically tuned to a chosen target processor. For the moment, the most effective implementations of the BLAS subroutines for many contemporary processors were done by Kazushige Goto (University of Texas-Austin), a particular attention being paid to elimination of the TLB misses.

Figure 2 shows the times of matrix multiplication on Alpha and Opteron with a simple algorithm consisting of 3 loops and with the processor specific BLAS implementations (Goto's implementations, not presented here, are still a little bit faster). This simple example of matrix multiplication demonstrates the effect of using libraries.

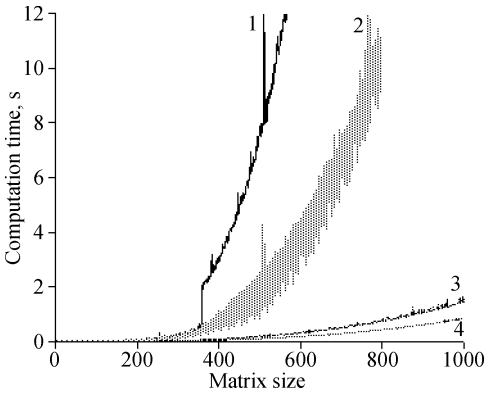

Figure 2. Matrix multiplication time for simplest algorithm on Alpha (1) and Opteron (2), for CXML `dgemm` on Alpha (3), and for ACML `dgemm` on Opteron (4)

The functions of mathematical libraries are wider than just the BLAS. Note that the programs which use functions other than the BLAS are generally not portable between processors of different producers. Here follows a short list of functions provided by the mathematical libraries of AMD, Compaq and Intel.

AMD Core Math Library version 2.5 (http://www.developwithamd.com/acml/):

- BLAS, LAPACK;
- Fourier Transform;
- Exponent, logarithm, sine, cosine functions of the vector elements (vector operations).

Compaq Extended Math Library (http://h18000.www1.hp.com/math/):

- BLAS, LAPACK;
- Direct and iterative solvers for sparse matrix SLAEs.
- Signal processing: 1–3D Fourier transforms, filters, convolution.
- Array-math library vector operations: the square root of N elements of an array, cosine, sine, exponent, logarithm of N elements of an array.
- Random number generators. Uniform and normal distributions.
- Sorting.

Intel Math Kernel Library version 7.2 (http://www.intel.com/software/products/mkl):

- BLAS, LAPACK;
- 1–7D Fourier transforms;
- Vector Math Library;
- Vector Statistical Library, vector random number generator, many distribution functions;
- Sparse SLAE solvers.

A short list of other famous libraries follows.

Free software:

1. GSL — GNU Scientific Library. A large set of subroutines of different areas of computations (www.gnu.org/software/gsl/).
2. GLPK — GNU Linear Programming Kit (http://www.gnu.org/software/glpk/glpk.html).
3. FFTW — Fastest Fourier Transform in the West, multidimensional (http://www.fftw.org).
4. UMFPACK — asymmetric sparse SLAE solvers (http://www.cise.ufl.edu/research/sparse/umfpack/).
5. SPARSEKIT — basic sparce matrix computations (http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html).
6. PETSc — PDE solvers (http://www-unix.mcs.anl.gov/petsc/petsc-2/).

Commercial libraries:

1. NAG Fortran 77 library. A large set of subroutines of different areas of computations (http://www.nag.co.uk).
2. IMSL Fortran Numerical Library. A large set of subroutines of different areas of computations (http://www.absoft.com/Products/Libraries/imsl.html).

6. Optimizing with compilers

Compiling is a process of program translation from the high-level language to the language of machine instructions. Compiling is performed by special programs named compilers. Unless we write assembly code, we are always using a compiler. Up-to-date compilers have a wide range of facilities for optimization of a generated code and tuning it to concrete architectures. Thus, it is possible to have various implementations of the same program which will have strongly different execution times. Usually it is much better to employ the compiler for optimization. In this section, we consider the most convenient and effective methods for program's acceleration by means of compilers.

6.1. Optimization options. For reducing the program run time, the compiler can make the following manipulations on the code being generated: integration of functions into their callers, global register allocation; elimination of an unreachable code; instructions reordering; automatic parallelization; elimination of the computation redundancy; loops conversion (unrolling + pipelining, vectorization, etc.); replacement of slow operations

by the fast ones (e.g. replacement of multiplication and division by $2N$ with a bitwise shift), optimization for a particular microprocessor architecture. One has to use the corresponding options to let the compiler know what kind of optimization should be applied. A typical compiler has hundreds of flags/options. Most of them are never used or are not related to optimization. For an exact option syntax, see the compiler documentation. It is important that the programmer can find the best variant of compilation of his program only by means of searching in different combinations of the optimizing techniques.

6.2. General optimization. The compilers has a great number of different optimizing techniques, each of them can be activated by switching-on a special option. At the same time, there are some general options enabling sets of optimization techniques at once. Such general optimizations are divided into several levels. Each following level includes all the techniques from the previous level, and some more complex and aggressive techniques. As a rule, zero level means no optimization. The default level of general optimization is different for different compilers. Moreover, it may be the zero level. As was shown in our experiments, the highest level of optimization does not always provide the best program performance. It is important to know, that the employment of optimizing techniques may affect your results. Therefore, a program should be tested and checked for correctness after any optimizations are applied. For the most compilers under the UNIX-like operating systems, the level of optimization can be enabled by the `-O`n option, where n is the number of the level. For the ccc compiler, n may be a value from 0 to 4, for the gcc compiler, it lies in the interval from 0 to 3. There are some other options enabling sets of optimization techniques, such as `-fast` and `-tune`, which can also help to generate a program with a good performance. Figure 3 demonstrates how the time needed to solve the Poisson equation (an explicit scheme is used) depends on the applied optimization level. N is the size of the square mesh.

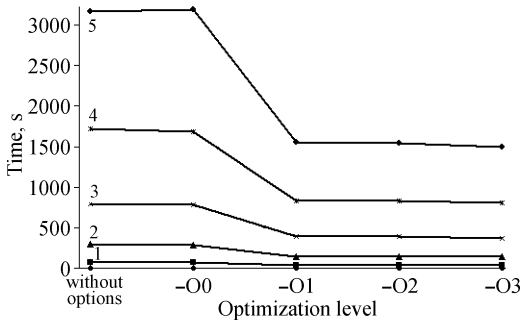

Figure 3. Dependence of Poisson equation solving time on compilation level: 1) $N = 201$, 2) $N = 301$, 3) $N = 401$, 4) $N = 501$, 5) $N = 601$

6.3. GNU C/C++ compiler (gcc). Generation of a code exclusively tuned to a given processor architecture is activated by the option `-march=` *target_arch*, where the *target_arch* is to be chosen from `i386`, `i486`, `i586`, `i686`, `pentium`, `pentium-mmx`, `pentiumpro`, `pentium2`, `pentium3`, `pentium4`. If the processor and the compiler support special sets of instructions (MMX, SSE, etc.), one can use additional options for the greater speedup. For instance, `-mfpmath=sse` option enables utilization of the SSE extensions.

6.4. Compaq C Compiler (ccc). Use `-arch` *target_arch* option for optimization for a particular processor architecture. The *target_arch* may be chosen from the following set: `generic` (default value), `host` (architecture of the processor, where the program is compiled), `ev4`, `ev5`, `ev56`, `ev6`, `ev67`, `pca56`. In addition to the previous option, one may use flag `-ansi_alias` which informs the compiler that the program uses only legal combinations of the methods of memory access (established by ANSI C standard). Assume that the compiler can produce more a sophisticated optimization conversation. We observe that `-arch` host option slows down the program if used without `-ansi_alias` option. In relation to that fact we recommend use both this options together with each other.

7. Recommendations to programmers

A program code has to satisfy a certain set of requirements to allow effective optimization by a compiler. The most general and effective recommendations are given bellow.

Use the language more accurate for more accurate description of the nature and utilization methods of the programming objects:

- Prefer local variables to global ones. The compiler analysis is more effective with local variables.
- Use standard functions for implementation of the widespread operations. The compiler has effective implementations of many such operations.

Follow the recommendations:

- Do not leave procedures with empty body.
- Group functions used together in the same file.
- Eliminate program blocks which are never used.
- In the file, subroutines should follow each other in the order in which they are supposed to be used (it may be done by compiler if the corresponding option is used).

- Use the compiler's hints and flags/options.
- If your program consists of several files, compile it as a whole at once (not separately).
- Use options allowing optimization between procedures and files.
- Try to eliminate dependencies between loop iterations and between index array elements. Let the compiler know that they are independent using the appropriate options (see compiler documentation).
- Use branch-free loop iterations. Even with sophisticated branch prediction hardware, branches are bad for performance. If you can't eliminate them, at least try to get them out of the critical loops. Try to help compiler, rather than do its job.
- Do not use fetch-commands in advance.
- Do not unroll loops by hand.
- Do not use an assembler language in programs in C, C++, Fortran, etc.
- Try to help the compiler to do its job; write a simpler and obvious code. In particular, do not implement parallelization of computations between functional processor units by means of an assembler language; avoid function calls in loops and computations of the end condition inside the loop body.

References

[1] Wilkinson B. Computer Architecture. Design and Performance. — Cambridge: University Press, 1994.

[2] Sima A., Fountain T., Kacsuk P. Advanced Computer Architectures. A Design Space Approach. – London: Addison Wesley, 1998.

[3] 21264/EV68A Microprocessor Hardware Reference Manual. — ftp://ftp.compaq.com/pub/products/alphaCPUdocs.

[4] Software Optimization Guide for AMD AthlonT 64 and AMD OpteronT Processors 25112 Rev. 3.05. — November, 2004.

[5] Kasperski K. Technic for Program Optimization. — SPb.: BHW-Peterburg, 2003.

[6] http://www.cs.utexas.edu/users/flame/goto/.

[7] Gerber R. The Software Optimization Cookbook. High-Performance Recipes for the Intel Architecture. — Intel Press, 2002.