In order to use effectively computers, you must first understand basics of hardware (which can change rather fast, so you have to keep up!). This is also useful in avoiding marketing tactics of computer manufacturers ....

The basic components of modern workstations, for our purpose here, are

i) The CPU (Central processing unit), which is the fastest part of the computer, having a number of very high-speed memory units called registers.

ii) The cache, a very fast bit of memory, that holds instructions, addresses, and data in their passage between the CPU and the slower RAM memory.

iii) The RAM (Random access memory), which can be accessed randomly (hence its name) is where the program resides while it is being processed.

iv) The hard disk, at the bottom of the memory pyramid, is the slowest but largest (and permanent) repository of data.

Now, there are some basic timescales and sizes you can keep in mind to see the difference between different components (the absolute scale in these numbers changes quite rapidly, but it gives you a feeling for what's going on in relative terms):

i) CPU: $5 \times 10^6$ transistors in a $0.1 \text{cm}^2 \times 100 \text{nm}$ layers. Clock speed ~ few GHz. Communicates with rest of the system through the front side bus (FSB) typically (now) half-third of clock speed.

ii) L2 Cache: works @ CPU speed, typically ~ MB in size. Typically made out of static RAM (SRAM): "flip-flap" circuitry that...
allows current to move randomly on transistors - it is faster than
DRAM (used for RAM), but uses more power and generates more heat.
Typical time scale here is about 1-2 few nano seconds, located @ CPU.

iii) RAM: Typically dynamic (DRAM), must be refreshed ~100s
of times per second, based on capacitors. Now most popular is
SDRAM, S stands for synchronous, it has a clock that
synchronizes input and output processes. Typically now it is
also DDR (double data rate) which theoretically doubles bandwidth
by transmitting data on rising and falling of clock signal.
Typical time scale here is 5-10 nano seconds, and typical
size are 1GB to few GB's, do serious computing. Connects
to CPU through memory bus, an important issue is how wide
is the bus, larger than 32-bit (e.g. 128-bit) is becoming common
as computers move from 32-bit to 64 bit addressing. For example, a
400 MHz memory 128-bit wide gives a bandwidth of ~ 0.4 x 16 GB/s = 6.4 GB/s

iv) Hard disk: Typically magnetic disks, and hundreds of MB to
terabytes TB, access time of few milliseconds (ms), due to
basically seek time (time that takes to go on any to some spot in disk).
Once you are there you can transfer @ ~ 60 Mb/s (like Gigabit
Ethernet, notice here b = bit, B = byte = 8 bits). Thus transfer
to disk is about 50 times slower than to RAM, but when you
are in transferring mode, if you go back and forth, then
you have to add access time and track hills, you because you
are dealing with ms rather than ~ nano seconds.

Performance of your code will depend on system architecture, CPU
clock speed, FSB, Cache, Memory bandwidth, MOST LXXXI on
the type of code (and how it is written). In order to avoid
dealing with the slowest parts of the system, we have to
understand a bit more about how CPU works, in particular
Pipeline

A pipeline takes advantage of the fact that many operations can be divided into stages that can proceed concurrently. In other words, one begins executing an instruction before completing the previous one:

```
in  1 2 3 4 5  

results
```

In this five-stage pipeline, an operation entering on the left proceeds on its own for 5 clock ticks before emerging on the right. Given that pipeline stages are independent of one another, up to 5 operations can be in-flight at a time. Without pipelining, it would have taken 5 clock ticks to get a single result, a pipeline produces (after some initial time) one result every clock tick.

Pipelines are used in many ways in computers, e.g., they include:

- Instruction processing, memory references, floating point arithmetic operations, etc.

For example, an instruction (although we usually think of them as a single step) are pipelined into several stages, schematically (in five stages):

1) Instruction Fetch: the CPU fetches an instruction from memory
2) Decode: the instruction is recognized or decoded
3) Operand Fetch: the processor fetches operands the instruction needs, these "ops" may be in registers or in memory
4) Execute: instruction gets executed
5) Write back: results are written back to where it's supposed to go 

Note that:

- Individual instructions are no faster, but throughput is higher.
- Potential speedup is (roughly) determined by the number of stages.
- Tiling and draining the pipeline limits speedup
- Rate through pipe is limited by slowest stage
- Less work per stage implies faster clock

Ideally, for a pipeline that is full all the time, one would like to have as many stages as possible. However, this is very risky for several reasons, e.g.

i) Structural hazards: attempt to use the same resource two different ways at the same time (e.g., two instructions try to read the same memory at the same time)

ii) Data hazards: attempt to use something before it's ready, e.g., when an instruction depends on result of prior instruction still in the pipeline: \[ y = a + b \quad y = c - x \]

iii) Control hazards: attempt to make a decision before condition is evaluated, e.g., branch instructions: it is not immediately known which instruction executes after a branch.

When these conflicts appear, an obvious solution is to stall (do nothing) until conflict is resolved: this introduces bubbles into the pipeline. Note that the longer the pipeline, the more it takes the "bubble" to propagate through, and thus the more wasted cycles.

To improve this situation, CPUs use branch prediction and speculative operation techniques to reduce bubbles: a prediction is made (based on static and dynamic lookup tables) and if the prediction turns out to be right, a pipeline bubble is avoided. Modern processors can predict about 90-95% of the time.

Also note that long pipelines need a lot more hardware resources to keep them full, thus the gain in adding stages is not linear in the number of stages (the compiler reduces the cost of data and
Control hazards by loading delay slots and making branches predictable.

Some advice and things to avoid...

The description above gives you a very basic picture of what's going on; basically, you want to keep all your code in the fastest region of the computer, i.e., CPU, avoiding going to memory, and of course, hard drive (except when you need to write), and when you are at CPU, the main goal is to keep the pipeline full all the time.

To have good use of the cache, you would like to use arrays as locally as possible - when something is grabbed from main memory to cache, a whole line is transferred (not just the element you ask for), thus if in the next operation you ask for some element close to previous one, it will be already in cache. Beware C and Fortran store arrays in opposite ways. For example, this is good in Fortran (but bad in C)

```fortran
do j = 1,N
    do i = 1,N
        ans = ans + A(i,j)  \( \text{access } A(1,1), A(1,2), \ldots \text{they are all sequentially stored in cache.} \)
    end do
end do
```

In C you should have the i-loop be the outer one.

Each time you ask for something in cache and it's not there, it has to be found in main memory, this is known as cache misses. Notice that if your code requires working with large arrays accessed in random ways, having a cache can be a penalty since each time you ask for something, it grabs a lot of additional stuff that you are not going to ask soon for.
have ways of turning off cache if your code is like that.

- Another way of keeping memory local is to declare things
that you use at some time continuously in memory space.
By if you declare

\[ A(10), B(100), C(100) \]

and then do something like

\[ \text{do } i=1, 10 \]

\[ A(i) = B(i) + 3 \]

\[ \text{enddo} \]

A and C are far away from each other, better to have C
declared next to A instead of having B in between...

A good compiler will probably figure out this for you.

- Avoid conditional statements inside loops (if...) - Also
avoid subroutine calls inside loops. This is to improve pipelining.

- Also when nested loops whose order can be interchanged, the
longest loop should always be the interior loop.

- If possible, use indexed loops (do-loops) instead of conditional
loops (while-loops).

- Loop unrolling helps performance because it shortens up a loop with
calculations that can be done in parallel (pipelined). For example:

\[ \text{do } i=1, N \]

\[ A(i) = B(i) + B(i) * C \]

\[ \text{enddo} \]

\( (\text{assume } N \text{ is a multiple of } 4) \)

Otherwise need to include

a pre-conditioning loop for
the rest)

Compilers typically do this for you (check the man pages, for -03)
- Keep in mind the strength of operations, normally +, -, *, /, ** is the order in which they are faster. Also raising a real number to an integer power, better to do a**3 than a*3. Again, the compiler may take care of this for you.

- When mixing different types, e.g. integers with reals, keep the different types as together as possible, by grouping. For example, if you do

\[ A = I * B * J * K \]

where I, J are integers and B, C reals, it is better to do

\[ A = (I*J) * (B+C) \]

- Also, try to avoid calling unnecessary functions, e.g.

\[ \text{if } (x^2+y^2+z^2 < r^2) \text{ then } \ldots \]

it is better and the same as

\[ \text{if } (\sqrt{x^2+y^2+z^2} < r) \text{ then } \ldots \]

- It is important if you want to improve your code, where most of the time is spent on, First to time your code you can use the simple "time" command before everything you would normally type in the command line to execute your code. After execution you will find a bunch of numbers, e.g.

```
1414.760 u  4.890 s  27:51.09  24.9 %  0+0k  2+210  0+0w
```

seconds of user

seconds of system

elapsed time

ratio of elapsed to CPU time

% of block input operations

% of block output operations

# of page faults

same, unshared memory in kb

avg. amount of shared memory in kb

CPU time

time devoted to process

time devoted to process

user time
To know which part of your code is taking most time, you should profile it.

For this you compile with -p and then execute and then use e.g. gprof *.exe to see what part of your code is taking most of the time. This is where you need to improve.

[show example]