High Performance Scientific computing
Lecture 12

S. Gopalakrishnan

Compiler Optimizations

FORTRAN and C Compilers

- Freeware
  - The GNU Fortran and C compilers, g77, g95, gcc, and gfortran are popular.
  - The port for window is either MinGW or Cygwin.
- Proprietary
  - Portland Group (http://www.pgroup.com)
  - Intel Compiler (http://www.intel.com)
  - Absoft
  - Lehay

Comparison (http://fortran-2000.com/ArnaudRecipes/CompilerTricks.html)

Array Considerations

In Fortran

```
do i=1,5
 do j = 1,5
  a(i,j)= …
 enddo
enddo
```

```
do j=1,5
 do i = 1,5
  a(i,j)= …
 enddo
enddo
```

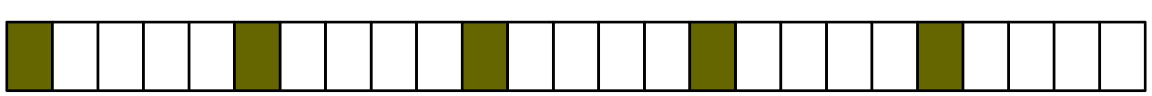
In C/C++

```
for(j=1;j<=5;j++){
 for(i=1;i<=5;i++){
  a[i][j]=…
 }
}
```

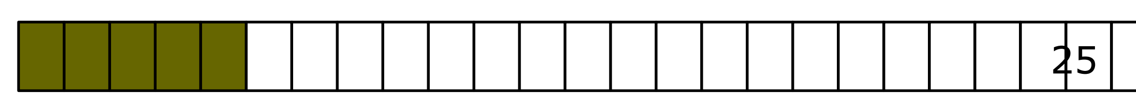
```
for(i=1;i<=5;i++){
 for(j=1;j<=5;j++){
  a[i][j]=…
 }
}
```

Corresponding memory representation

Outer 1 1 1 1 1
Inner 1 2 3 4 5



Outer 1 1 1 1 1
Inner 1 2 3 4 5



Blocking

- Blocking is used to reduce cache and TLB misses in nested matrix operations. The idea is to process as much data brought in the cache as possible

```
do i = 1,n
  do j = 1,n
    do k = 1,n
      C(I,j)=C(I,j)+A(I,k)*B(k,j)
    enddo
   enddo
enddo
```

Blocking

- Blocking is used to reduce cache and TLB misses in nested matrix operations. The idea is to process as much data brought in the cache as possible

```
do i = 1,n
  do j = 1,n
    do k = 1,n
      C(I,j)=C(I,j)+A(I,k)*B(k,j)
    enddo
   enddo
enddo
```

```
do ib = 1,n,bsize
  do jb = 1,n,bsize
    do kb = 1,n,bsize
      do i = ib,min(n,ib+bsize-1)
        do j = jb,min(n,jb+bsize-1)
          do k = kb,min(n,kb+bsize-1)
           C(I,j)=C(I,j)+
                 A(I,k)*B(k,j)
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

Loop Fusion

- The main advantage of loop fusion is the reduction of cache misses when the same array is used in both loops. It also reduces loop overhead and allow a better control of multiple instructions in a single cycle, when hardware allows it.

```
do i = 1,100000
  a = a + x(i) + 2.0 *z(i)
enddo

do j = 1,100000
   v = 3.0*x(j) - 3.314159267
enddo
```

Loop Fusion

- The main advantage of loop fusion is the reduction of cache misses when the same array is used in both loops. It also reduces loop overhead and allow a better control of multiple instructions in a single cycle, when hardware allows it.

```
do i = 1,100000
  a = a + x(i) + 2.0 *z(i)
enddo

do j = 1,100000
   v = 3.0*x(j) – 3.314159267
enddo
```

```
do i = 1,100000
  a = a + x(i) + 2.0 *z(i)
  v = 3.0*x(i) – 3.314159267
enddo
```

Sum Reduction

- Sum reduction is another way of reducing or eliminating data dependencies in loops. It is more explicit than the loop unroll.

```
do i = 1,1000
  a = a + x(i) * y(i)
enddo
```

2000 cycles

Sum Reduction

- Sum reduction is another way of reducing or eliminating data dependencies in loops. It is more explicit than the loop unroll.

```
do i = 1,1000
  a = a + x(i) * y(i)
enddo
```

2000 cycles

```
do i = 1,1000,4
  a1 = a1 + x(i) * y(i)
      + x(i+1)* y(i+1)
  a2 = a2 + x(i+2)* y(i+2)
      + x(i+3)* y(i+3)
enddo

a = a1 + a2
```

751 cycles

Compiler Code Optimizations

- Introduction
  - Optimized code
    - Executes faster
    - efficient memory usage
    - yielding better performance.
  - Compilers can be designed to provide code optimization.
  - Users should only focus on optimizations not provided by the compiler such as choosing a faster and/or less memory intensive algorithm.

Compiler Code Optimizations

- A Code optimizer sits between the front end and the code generator.
  - Works with intermediate code.
  - Can do control flow analysis.
  - Can do data flow analysis.
  - Does transformations to improve the intermediate code.

Compiler Code Optimizations

- Optimizations provided by a compiler includes:
  - Inlining small functions
  - Code hoisting
  - Dead store elimination
  - Eliminating common sub-expressions
  - Loop unrolling
  - Loop optimizations: Code motion, Induction variable elimination, and Reduction in strength.

Compiler Code Optimizations

- Inlining small functions
  - Repeatedly inserting the function code instead of calling it, saves the calling overhead and enable further optimizations.
  - Inlining large functions will make the executable too large.

Compiler Code Optimizations

- Code hoisting
  - Moving computations outside loops
  - Saves computing time

Compiler Code Optimizations

- Code hoisting
  - In the following example (2.0 * PI) is an invariant expression there is no reason to recompute it 100 times.

```
DO I = 1, 100
        ARRAY(I) = 2.0 * PI * I
ENDDO
```

  - By introducing a temporary variable 't' it can be transformed to:

```
t = 2.0 * PI
DO I = 1, 100
        ARRAY(I) = t * I
END DO
```

Minimize number of Operations

- *During optimization, first thing needed to do is reducing the number of unnecessary operations performed by the CPU.*

```
do k=1,10
 do j=1,5000
  do i=1,5000
   a(i,j,k)=3.0*m*d(k)+c(j)*23.1-b(i)
  enddo
 enddo
enddo
```

1250 millions of operations

Minimize number of Operations

- *During optimization, first thing needed to do is reducing the number of unnecessary operations performed by the CPU.*

```
do k=1,10
 do j=1,5000
  do i=1,5000
   a(i,j,k)=3.0*m*d(k)+c(j)*23.1-b(i)
  enddo
 enddo
enddo
```

1250 millions of operations

```
do k=1,10
 dtmp(k)=3.0*m*d(k)
 do j=1,5000
  ctmp(j)=c(j)*23.1
  do i=1,5000
   a(i,j,k)=dtmp(k)+ctmp(j)-b(i)
  enddo
 enddo
enddo
```

500 millions of operations

Compiler Code Optimizations

- Dead store elimination
  - If the compiler detects variables that are never used, it may safely ignore many of the operations that compute their values.

Compiler Code Optimizations

- Eliminating common sub-expressions
  - Optimization compilers are able to perform quite well:

    X = A * LOG(Y) + (LOG(Y) ** 2)

  - Introduce an explicit temporary variable t:

    t = LOG(Y)

    X = A * t + (t ** 2)

  - Saves one 'heavy' function call, by an elimination of the common sub-expression LOG(Y), the exponentiation now is:

    X = (A + t) * t

Function call Overhead

```
do k = 1,1000000
  do j = 1,1000000
    do i = 1,5000
      a(i,j,k)=fl(c(i),b(j),k
    enddo
   enddo
enddo

function fl(x,y,m)
  real*8 x,y,tmp
  integer m
  tmp=x*m-y
  return tmp
end
```

Function call Overhead

```
do k = 1,1000000
  do j = 1,1000000
    do i = 1,5000
      a(i,j,k)=fl(c(i),b(j),k
    enddo
   enddo
enddo

function fl(x,y,m)
  real*8 x,y,tmp
  integer m
  tmp=x*m-y
  return tmp
end
```

→

```
do k = 1,1000000
  do j = 1,1000000
    do i = 1,5000
      a(i,j,k)=c(i)*k-b(j)
    enddo
   enddo
enddo
```

This can also be achieved with compilers inlining options. The compiler will then replace all function calls by a copy of the function code, sometimes leading to very large binary executable.

```
% ifc –ip
% icc –ip
% gcc –finline-functions
```

Compiler Code Optimizations

- Loop unrolling
  - The loop exit checks cost CPU time.
  - Loop unrolling tries to get rid of the checks completely or to reduce the number of checks.
  - If you know a loop is only performed a certain number of times, or if you know the number of times it will be repeated is a multiple of a constant you can unroll this loop.

Compiler Code Optimizations

- Loop unrolling
  - Example:

```
// old loop
for(int i=0; i<3; i++) {
                color_map[n+i] = i;
}
// unrolled version
int i = 0;
colormap[n+i] = i;
i++;
colormap[n+i] = i;
i++;
colormap[n+i] = i;
```

Compiler Code Optimizations

- Code Motion
  - Any code inside a loop that always computes the same value can be moved before the loop.
  - Example:

```
while (i <= limit-2)
do {loop code}
```

where the loop code doesn't change the limit variable. The subtraction, limit-2, will be inside the loop. Code motion would substitute:

```
t = limit-2;
while (i <= t)
do {loop code}
```

Complex Numbers

- *Watch for operations on complex numbers that have imaginary or real part equals to zero.*

```
! Real part = 0
complex *16 a(1000,1000),b
complex *16 c(1000,1000)

do j=1,1000
 do i=1,1000
   c(i,j) = a(i,j)*b
 enddo
enddo
```

6 millions of operations

Complex Numbers

- *Watch for operations on complex numbers that have imaginary or real part equals to zero.*

```
! Real part = 0
complex *16 a(1000,1000),b
complex *16 c(1000,1000)

do j=1,1000
 do i=1,1000
   c(i,j) = a(i,j)*b
 enddo
enddo
```

6 millions of operations

➡

```
real *8 aI(1000,1000)
complex *16 b,c(1000,1000)

do j=1,1000
 do i=1,1000
   c(i,j) = (-IMAG(b)*aI(i,j),
             aI(I,j)*REAL(b));
 enddo
enddo
```

2 millions of operations

Compiler Code Optimizations

- Conclusion
  - Compilers can provide some code optimization.
  - Programmers do have to worry about such optimizations.
  - Program definition must be preserved.

Version Control

Need for Version Control

Scenario 1:

Your program is working
You change "just one thing"
Your program breaks
You change it back
Your program is still broken--*why?*

Has this ever happened to you?

Need for Version Control

Your program worked well enough yesterday
You made a lot of improvements last night...
    ...but you haven't gotten them to work yet
You need to turn in your program *now*

Has this ever happened to you?

Version Control with groups

Scenario:

You change one part of a program--it works

Your co-worker (fellow grad student) changes another part--it works

You put them together--it doesn't work

Some change in one part must have broken something in the other part

What were all the changes?

Version Control with groups

Scenario:

You make a number of improvements to a class

Your co-worker makes a number of *different* improvements to the *same* class

How can you merge these changes?

Version Control System

A version control system (often called a source code control system) does these things:

Keeps multiple (older and newer) versions of everything (not just source code)

Requests comments regarding every change

Allows “check in” and “check out” of files so you know which files someone else is working on

Displays differences between versions

Version Control System: Details of process

Files are kept in a *repository*

Repositories can be local or remote to the user

The user edits a copy called the *working copy*

Changes are *committed* to the repository when the user is finished making changes

Other people can then access the repository to get the new code

Can also be used to manage files when working across multiple computers

Centralised Version Control System

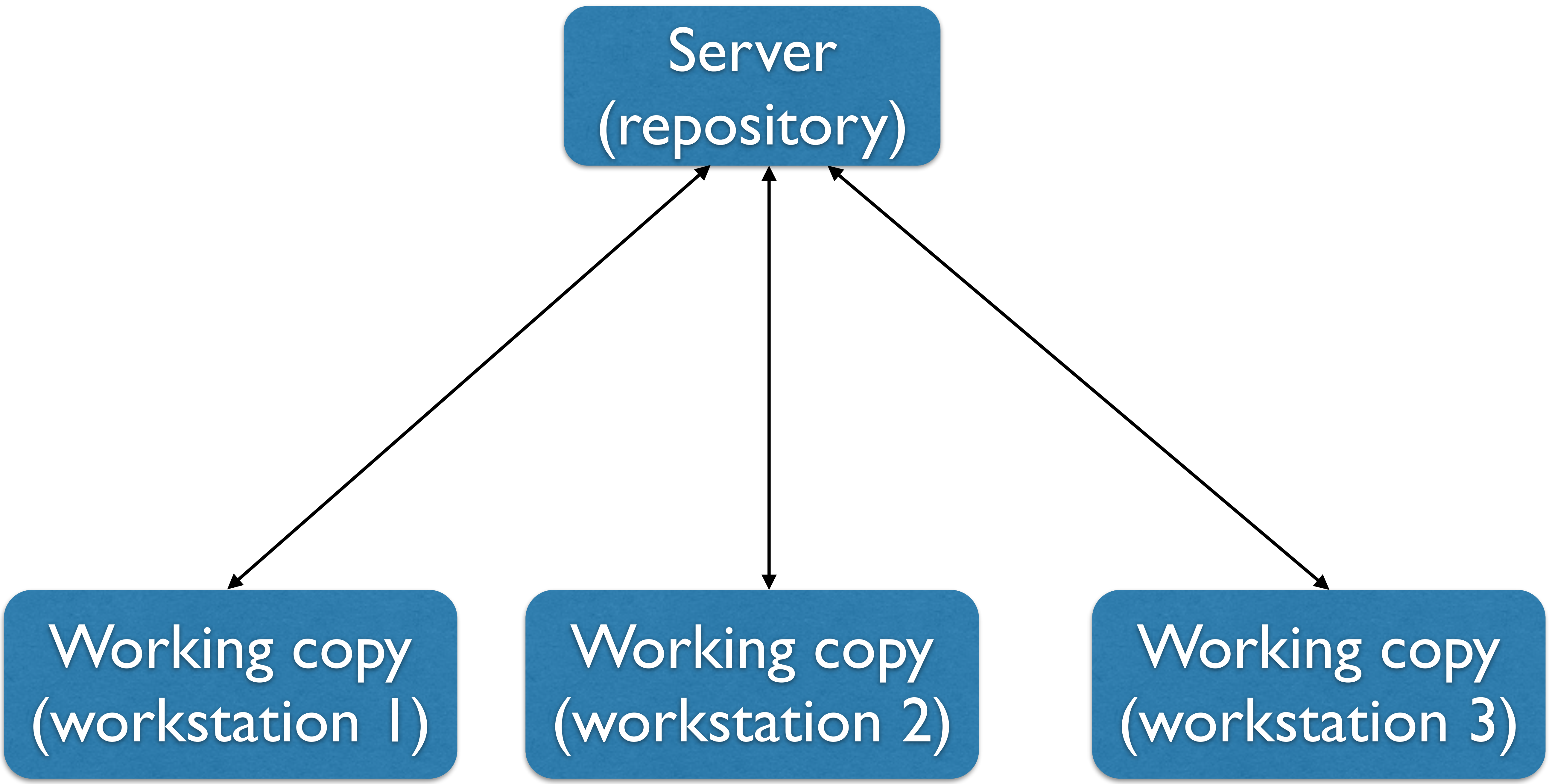
A single server holds the code base

Clients access the server by means of check-in/check-outs

Examples include CVS, Subversion, Visual Source Safe.

Advantages: Easier to maintain a single server.
Disadvantages: Single point of failure.

Centralised Version Control System



Distributed Version Control System

Each client (essentially) holds a complete copy of the code base.

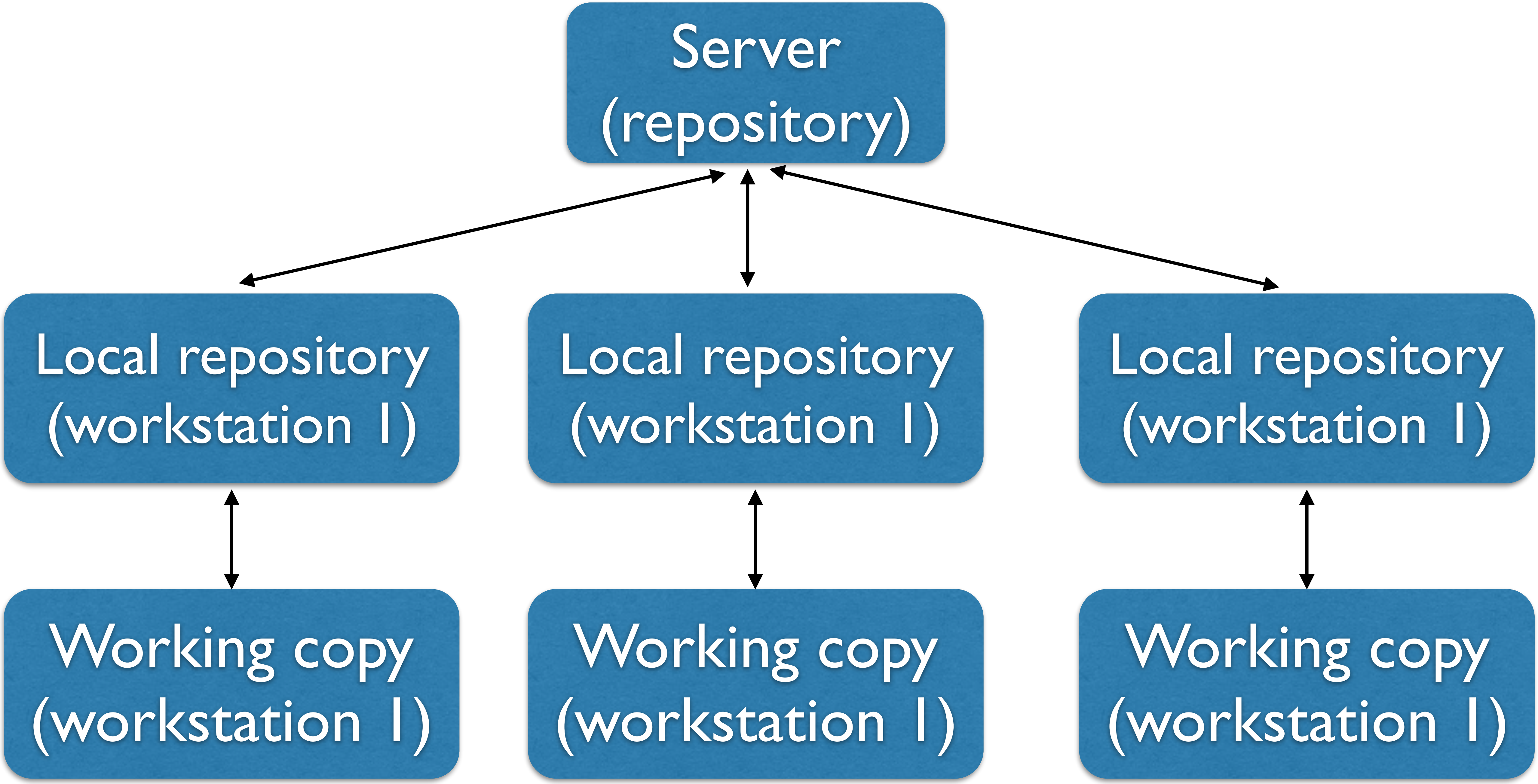
Code is shared between clients by push/pulls

Advantages: Many operations cheaper. No single point of failure

Disadvantages: A bit more complicated!

Examples: Git, Mercurial

Distributed Version Control System



More uses of Version Control System

Version control is not just useful for collaborative working, essential for quality source code development

Often want to undo changes to a file

- start work, realize it's the wrong approach, want to get back to starting point
- like "undo" in an editor…
- keep the whole history of every file and a *changelog*

Also want to be able to see who changed what, when

- The best way to find out how something works is often to ask the person who wrote it

Branching Projects

Branches allows multiple copies of the code base within a single repository.

- Different projects have different requirements
  - Project A wants features A,B, C
  - Project B wants features A & C but not B.
  - Project C wants only feature A.
- Each project has their own branch.

Different versions can easily be maintained