CS528
SCO and Tuning

A Sahu

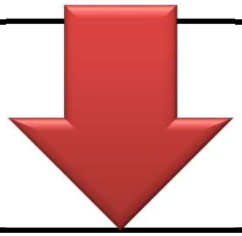
Dept of CSE, IIT Guwahati

Outline

- Intro to Code Optimization
- Machine independent/dependent optimization
- Common sense of Optimization
  - Do less work, avoid expensive Ops, shrink working set
- Simple measure Large impact : simd, branch, comm sub expre
- C++ Optimization
- Scalar Profiling
  - Manual Instrumentation (get_wall_time, clock_t)
  - Function and line based profiling (gprof, gcov)
  - Memory Profiling (valgrind, callgraph)
  - Hardware Performance Counter (oprofile,likwid)

CSO: Loop Jamming

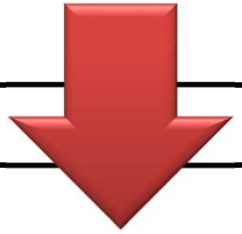
```
for(i=0;i<10000;i++){
     Dostuff(i);//Small Independent work
}
for(i=0;i<10000;i++){
     DoMorestuff(i); //Small Independent work
}
```



```
for(i=0;i<10000;i++){
     Dostuff(i);
     DoMorestuff(i);
}
```

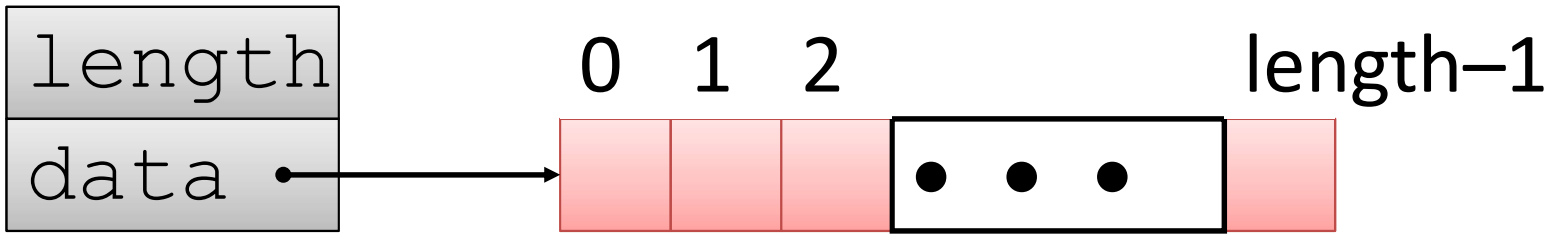
CSO: Function Looping

```
for(i=0;i<10000;i++){
     Func(t,i);
}
Fun (int w, d){//do lots of stuff}
```



```
funn(t);
void funn(w){
 for(i=0;i<10000;i++){//do lots stuffs
  }
}
```

CSO: Example: Vector ADT




```
vec_ptr new_vec(int len)
```

- Create vector of specified length

```
int get_vec_element(vec_ptr v, int
  index, int *dest)
```

- Retrieve vector element, store at *dest
- Return 0 if out of bounds, 1 if successful

```
int *get_vec_start(vec_ptr v)
```

- Return pointer to start of vector data

Optimization Example

- Procedure
  - Compute sum of all elements of vector
  - Store result at destination location
  - What's the Big-O of this code?

```
void combine1(vec_ptr v, int *dest){
   int i;
   *dest = 0;
   for (i=0; i<vec_length(v); i++) {
      int val;
      get_vec_element(v, i, &val);
      *dest += val;
   }
}
```

Move `vec_length` Call Out of Loop

- Value does not change from one iteration to next
- Code motion, `vec_length` requires only constant time, but significant overhead

```
void combine2(vec_ptr v, int *dest){
  int i;
  int length = vec_length(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

Reduction in Strength

```
void combine2(vec_ptr v, int *dest){
 int length = vec_length(v);
 *dest = 0;
  for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

Reduction in Strength

```
void combine3(vec_ptr v, int *dest){
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

Eliminate Unneeded Memory Refs

```
void combine4(vec_ptr v, int *dest){
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

Code Motion Example #2

- Procedure to Convert String to Lowercase
  - Extracted from many beginners' C programs
  - (Note: only works for ASCII, not extended characters)

```
void toLower(char *s) {
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i]>='A' && s[i]<='Z')
        s[i] -= ('A' - 'a');
}
```

Optimization Blocker: Procedure Calls

- *Why couldn't the compiler move* `vec_len` *or* `strlen` *out of the inner loop?*
  - Procedure might have side effects
    - Alters global state each time called
  - Function might not return same value for given arguments
    - Depends on other parts of global state
    - Procedure `lower` could interact with `strlen`

Optimization Blocker: Procedure Calls

- *Why doesn't compiler look at code for* `vec_len` *or* `strlen`*?*
  - Linker may overload with different version
    - Unless declared static
  - Interprocedural optimization is not extensively used, due to cost
- **Warning:**
  - Compiler treats procedure call as a black box
  - Weak optimizations in and around them

Profiling for Serial Code

Profiling for Serial Code

- Manual Instrumentation (get_wall_time, clock_t)
- Function and line based profiling (gprof, gcov)
- Memory Profiling (valgrind, callgraph)
- Hardware Performance Counter (oprofile,likwid)

Manual Instrumentation

- System Status
  - $uptime, $top , $vmstat
  - $systemmonitor, $gnome-system-monitor
- $time ./a.out
  - real time/wall clock time
  - cpu time and system time
  - cputime=sys time+usr time
- Using get_wall_time, clock_t

vmstat : command reports statistics about kernel threads in the run and wait queue, memory, paging, disks, interrupts, system calls, context switches, and CPU activity

This command performs a CPU usage monitoring operation using a number of CPU monitoring counters including the total CPU usage, the user-level CPU usage, the system-level CPU usage, the CPU interrupt time, the CPU interrupt rate, the C1, C2 and C3 low-power CPU states and the CPU frequency.

Uptime : prints the current time, the length of time the system has been up, the number of users online, and the load average

Top : The top command is used for memory monitoring. It works only on Linux platform. The top command produces an ordered list of running processes selected by user-specified criteria, and updates it periodically.

Manual Instrumentation

- $time command and Using get_wall_time,

```
#include <time.h>

int main(){
clock_t t; double Etime;
t = clock();
//Do some Work
t = clock() - t;
Etime=((double)t)/CLOCKS_PER_SEC;
printf("ETime =%f seconds",Etime)
return 0;
}
```

Profiler: Hotspot Analyzer

- Given a program
- Finding out part of the program which takes maximum amount of time
- Optimizing hot-spot area reduce the execution time significantly
- Suppose a program spend 99% of time in a small function/code
  - Optimizing that code will result better performance

Function and line based profiling

- GNU profile (gprof)
  - $gcc –p test.c
  - $./a.out
  - $gprof ./a.out
  - $gprof ./a.out >FPprofile.txt
- GNU coverage (gcov)

Gprof Example

```
#include <stdio.h>
void FunA() {
   int i=0,g=0;
   while(i++<100000)
     {  g+=i; }
}
void FunB() {
   int i=0,g=0;
   while(i++<400000)
     { g+=i; }
}
```

```
int main(){
    int iter=5000;
    while(iter--) {
        FunA();
        FunB();
    }
    return 0;
}
```

Gprof Example: Flat Profile

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
80.26	5.55	5.55	5000	1.11	1.11	FunB
20.94	6.99	1.45	5000	0.29	0.29	FunA

Gprof Example: Call Graph

```
Call graph
index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00    6.99                 main [1]
               5.55    0.00    5000/5000       FunB [2]
               1.45    0.00    5000/5000       FunA [3]
-----------------------------------------------
               5.55    0.00    5000/5000       main [1]
[2]     79.3   5.55    0.00    5000         FunB [2]
-----------------------------------------------
               1.45    0.00    5000/5000       main [1]
[3]     20.7   1.45    0.00    5000         FunA [3]
```

Function and line based profiling

- GNU profile (gprof)
- GNU coverage (gcov)
  - **$gcc -fprofile-arcs -ftest-coverage tmp.c**
  - **$./a.out**
  - **$gcov tmp.c**

File 'tmp.c'

Lines executed:87.50% of 8

Creating 'tmp.c.gcov'

Gcov output

```
#include <stdio.h>
int main (){
   int i, total;
   total = 0;
   for (i = 0; i < 10; i++)
           total += i;
   if (total != 45)
           printf ("Failure\n");
   else printf ("Success\n");
   return 0;
}
```

```
        -:  1:#include <stdio.h>
        1:  2:int main (){
        -:  3:   int i, total;
        1:  4:   total = 0;
       11:  5:   for (i = 0; i < 10; i++)
       10:  6:       total += i;
        1:  7:   if (total != 45)
    #####:8:       printf ("Failure\n");
        1:  9:   else printf ("Success\n");
        1: 10:   return 0;
        -: 11:}
```

Valgrind

- Free tools: **$sudo apt-get install valgrind**
- CallGraph, Profiler, Memory Check…
  - Many more
  - From C code, one can use API of valgrind
- Program analysis tools are useful
  - Bug detectors, Profilers, Visualizers
- **Dynamic binary analysis** (DBA) tools
  - Analyse a program's machine code at run-time
  - Augment original code with **analysis code**

Valgrind

```
void Work1(int n) {
    int i=0,j=0,k=0;
    while(i++<n){
        while(j++<n){while(k++<n);}
    }
}
void Work2(int n) { int i=0;  while(i++<n);}
void Maneger(int n1, int n2){
     Work1(n1); Work2(n2);
}
void Projects1() { Maneger(1000000, 1000);}
void Projects2() { Maneger(100, 1000000);}

int main() {
  Projects1(); Projects2();  return 0;
}
```

Valgrind: How to use

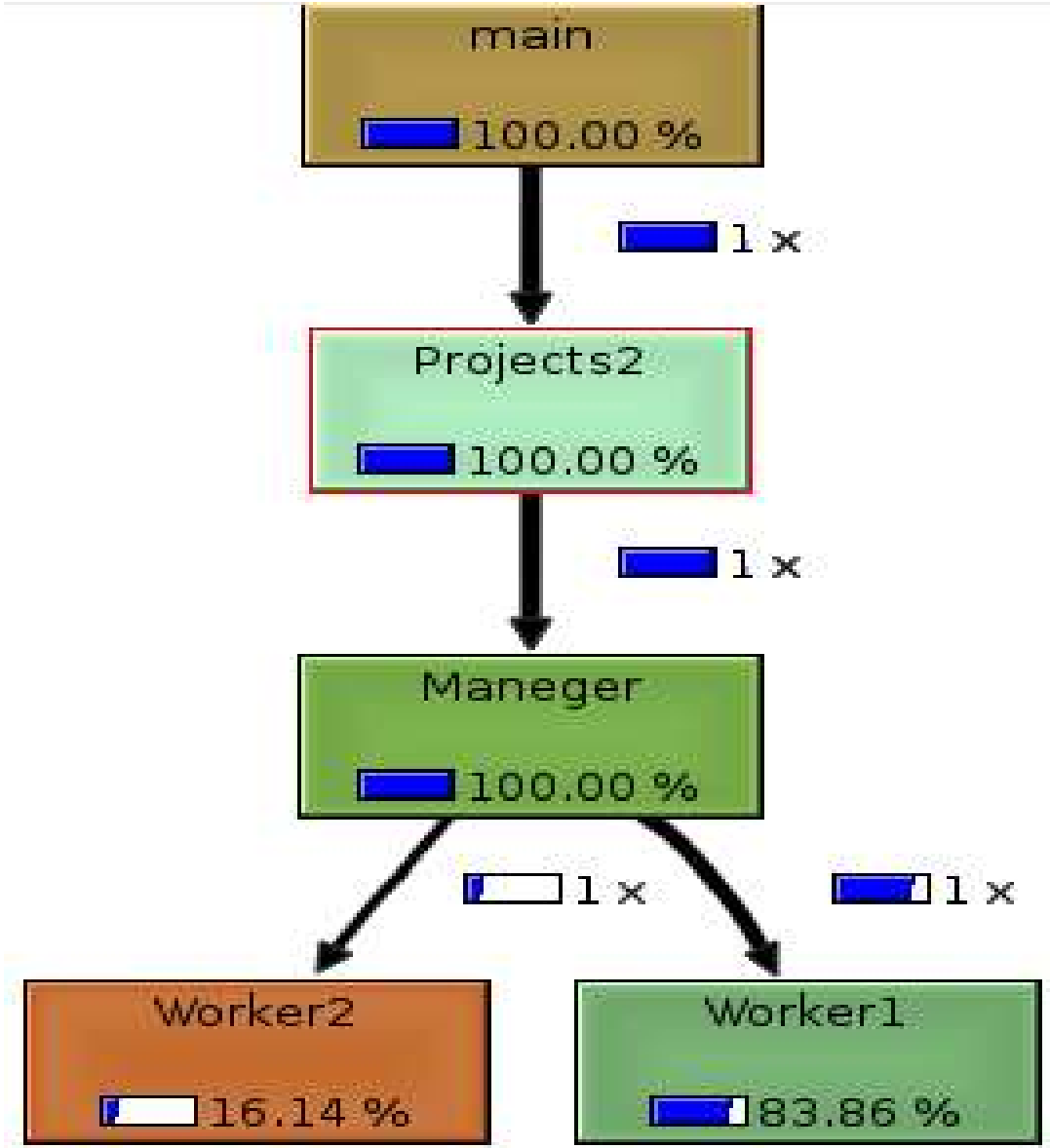
- $gcc -pg -o Valgrindtest Valgrindtest.c
- $valgrind --tool=callgrind ./Valgrindtest
- $ls

  Valgrindtest Valgrindtest.c callgrind.out.**11233**

$kcachegrind `ls -tr callgrind.out.* | tail -1`

pid

Valgrind: Call Graph



Further Optimizations for Serial Code

- Simple measure Large impact : simd, branch, comm sub expre
- C++ Optimization

Simple measures, large impact

- Elimination of Common Sub-expressions
- Avoid Branches:
  - **Code Can be SIMdized by compiler/gcc**
  - **Effective use of pipeline for loop code**
- Use of SIMD Instruction sets
  - **512 bit AVX SIMD in modern processor**
  - **ML/AI app use 8 bit Ops, can be speed up 512/8=64 time by simply SIMD-AVX**

Elimination of Common Sub-expressions

```
//value of s, r, x don't change in this loop
for (i=0; i<ALargeN; i++){
    A[i]=A[i]+s+r+sinx(x);
}
```

```
//value of s, r, x don't change in this loop
Tmp=s+r+sinx(x);
for (i=0; i<ALargeN; i++){
    A[i]=A[i]+Tmp;
}
```

Avoid Branches

```
for (i=0; i<N; i++)
    for(j=0;j<N;j++){
      if(i<j) S=1; else S=-1;
       C[i] =C[i]+S*A[i][j]*B[i];
    }
```

```
for (i=0; i<N; i++){
     for(j=0;j<i;j++)
        C[i] =C[i] -A[i][j]*B[i];
     for(j=i;j<N;j++)
        C[i] =C[i] +A[i][j]*B[i];
}
```