Performance of serial C programs

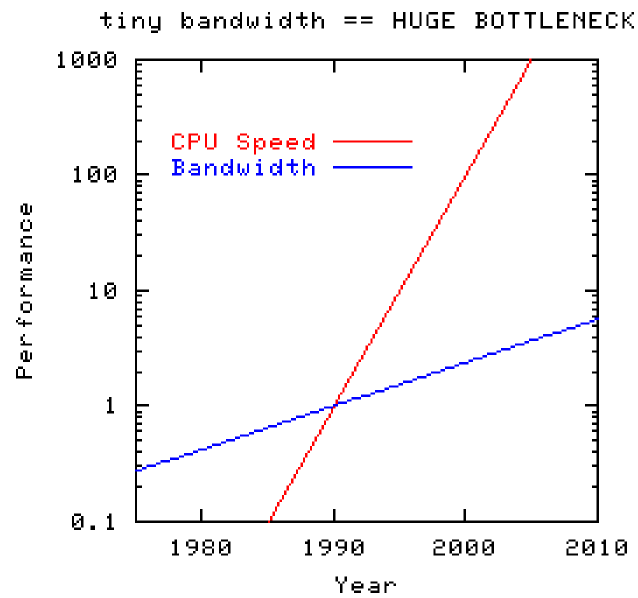
Motivations

- In essence, parallel computations consist of serial computations that are executed on multiple computing units, plus necessary "collaboration" in between
- The overall performance of a parallel program depends on the performance of the serial parts and the collaboration cost
- Effective serial computing on a single processor (core) is fundamental
- This lecture looks at several performance-affecting factors and their implications for typical scientific computations

FLOPS

- FLOPS — floating-point operations per second
- A commonly used metric for processor performance
  - megaflops: 10^6 flops
  - gigaflops: 10^9 flops
  - teraflops: 10^{12} flops
  - petaflops: 10^{15} flops
  - exaflops: 10^{18} flops
- As of November 2013, world's fastest computer—"Tianhe-2": a hybrid cluster of multicore CPUs and many-integrated-core coprocessors—has 54.92 petaflops theoretical peak performance, using 3.12 million cores
- Achieving peak performance is often impossible, relying on full memory performance and full utilization of instruction-level parallelism

Memory is the bottleneck for performance



http://www.streambench.org/

- Time to run a code: cycles spent on performing instructions, cycles spent on transferring data from/to memory
- Scientific computations are often memory intensive
- Memory speed (i.e. bandwidth and latency) is lagging behind the CPU clock frequency
- Memory size is another limiting factor

Example of memory bandwidth requirement

- Suppose we want to sum up an array of `double` values

```
double sum = 0.;
for (i=0; i<LENGTH; i++)
  sum += a[i];
```

- Each iteration reads 8 bytes (one `double` value) from memory
- For example, a memory read bandwidth of 2.9 GB/s (measured on Intel Xeon L5420 2.5GHz processor) only gives 2.9/8 = 0.37 GFLOPS for the above example.
  `http://browse.geekbench.ca/geekbench2/view/81731`
- Realistic situations may be even worse
  - more memory reads and writes per operation
  - memory writes can be slower than memory reads

Cache – a remedy for memory latency/bandwidth

- Memory latency is another limiting factor
  - Read/write a value from/to main memory typically takes $10 \sim 100$ clock cycles
- Cache is a small but fast buffer that duplicates a subset of the main memory
  - closer to CPU
  - small capacity
  - higher bandwidth than memory
  - usually several levels of cache (L1, L2 and possibly also L3)
- When CPU needs a value from main memory, the lowest-level cache is checked first, if not the next-level cache is checked, and so on

More about cache (1)

- Storage of data in a cache is organized as cache lines
- Each cache line is typically 32 bytes $\sim$ 128 bytes
- One entire cache line is read/written from/to memory
- Cache miss happens when CPU requests data that is not available in cache, the opposite is called cache hit

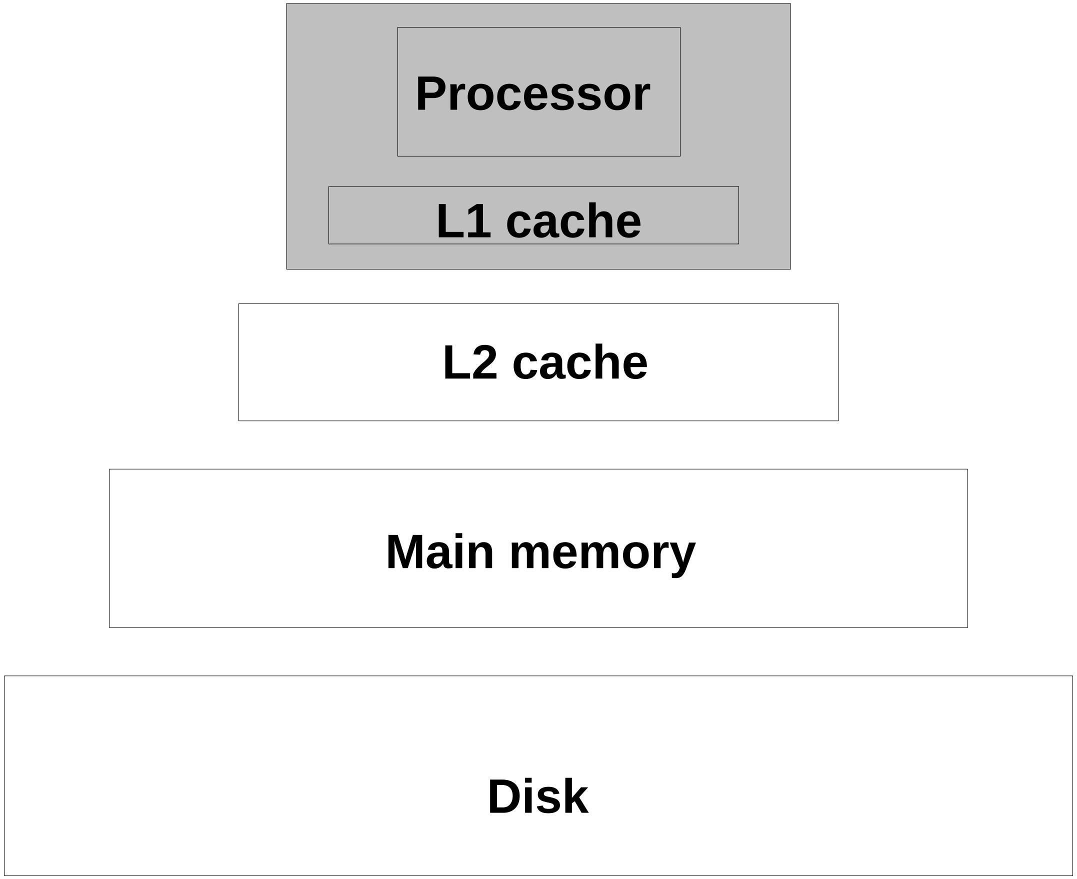
More about cache (2)

- On which cache line should a data block from main memory be placed?
  - fully associative
  - m-way associative
  - direct map
- Cache line replacement strategy for associative caches
  - least recently used (LRU)
  - FIFO
  - random
- How are data written back to main memory?
  - write-through (each store results in a memory write)
  - write-back (memory is updated only when the an entire cache line is to be evicted)

More about cache (3)

- The key to efficiency – reuse the data in cache as much as possible
- Spatial locality – neighboring data items in the main memory are used together in computations
  - one cache line can hold several consecutive data items
  - physically close data items are more likely to be in cache at the same time
- Temporal locality – data items used in the current operation are to be used in immediately upcoming operations

Storage hierarchy



Remark: modern CPUs can have 3 cache levels

How to secure single-core performance?

- Effective use of cache
  - smart design of data structures
  - correct traversal of arrays
  - the aim is good temporal and spatial locality
- Effective use of instruction-level parallelism
  - capable hardware
  - powerful compiler
  - good programming style may also be helpful
- Optimization
  - manual
  - compiler-enabled

Data locality lowers the pressure on memory

- A good code should take advantage of temporal and spatial locality, i.e., good data re-use in cache
- Spatial locality – if location `X` in memory is currently being accessed, it is likely that a location near `X` will be accessed next
- Temporal locality – if location `X` in memory is currently be accessed, it is likely that location `X` will soon be accessed again

Instruction-level parallelism

Several operations simultaneously carried out on a single processor (core) – "*parallel computing on a single core*"

- Pipelining – execution of multiple instructions partially overlapped
- Superscalar execution – using multiple execution units
- Data prefetching
- Out-of-order execution – making use of independent operations
- Speculative execution
    - branch prediction is very important

Loop optimizations (1)

Loop fusion

```
for (i=0; i<ARRAY_SIZE; i++)
  x = x * a[i] + b[i];
for (i=0; i<ARRAY_SIZE; i++)
  y = y * a[i] + c[i];
```

```
for (i=0; i<ARRAY_SIZE; i++) {
  x = x * a[i] + b[i];
  y = y * a[i] + c[i];
}
```

Loop overhead is reduced, better chance for instruction overlap

Loop optimizations (2)

Loop interchange

```
for (k=0; k<10000; k++)
  for (j=0; j<400; j++)
    for (i=0; i<10; i++)
      a[k][j][i] = a[k][j][i] * 1.01 + 0.01;
```

```
for (k=0; k<10; k++)
  for (j=0; j<400; j++)
    for (i=0; i<10000; i++)
      a[k][j][i] = a[k][j][i] * 1.01 + 0.01;
```

Assume that the data layout of array `a` has changed accordingly

Loop optimizations (3)

Loop collapsing

```
for (i=0; i<500; i++)
  for (j=0; j<80; j++)
    for (k=0; k<4; k++)
      a[i][j][k] = a[i][j][k] + b[i][j][k]*c[i][j][k];
```

```
for (i=0; i<(500*80*4); i++)
  a[0][0][i] = a[0][0][i] + b[0][0][i]*c[0][0][i];
```

Assume that the 3D arrays `a`, `b` and `c` have contiguous underlying memory

Loop optimizations (4)

Loop unrolling

```
t = 0.0;
for (i=0; i<ARRAY_SIZE; i++)
  t = t + a[i]*a[i];
```

```
t1 = t2 = t3 = t4 = 0.0;
for (i=0; i<ARRAY_SIZE-3; i+=4) {
  t1 = t1 + a[i+0]*a[i+0];
  t2 = t2 + a[i+1]*a[i+1];
  t3 = t3 + a[i+2]*a[i+2];
  t4 = t4 + a[i+3]*a[i+3];
}
t = t1+t2+t3+t4;
```

Purpose: eliminate/reduce data dependency and improve pipelining

Loop optimizations (5)

Improving ratio of F/M

```
for (i=0; i<m; i++) {
  t = 0.;
  for (j=0; j<n; j++)
    t = t + a[i][j]*x[j];  /* 2 floating-point operations & 2 loads */
  y[i] = t;
}
```

```
for (i=0; i<m-3; i+=4) {
  t1 = t2 = t3 = t4 = 0.;
  for (j=0; j<n-3; j+=4) { /* 32 floating-point operations & 20 loads */
    t1=t1+a[i+0][j]*x[j]+a[i+0][j+1]*x[j+1]+a[i+0][j+2]*x[j+2]+a[i+0][j+3]*
    t2=t2+a[i+1][j]*x[j]+a[i+1][j+1]*x[j+1]+a[i+1][j+2]*x[j+2]+a[i+1][j+3]*
    t3=t3+a[i+2][j]*x[j]+a[i+2][j+1]*x[j+1]+a[i+2][j+2]*x[j+2]+a[i+2][j+3]*
    t4=t4+a[i+3][j]*x[j]+a[i+3][j+1]*x[j+1]+a[i+3][j+2]*x[j+2]+a[i+3][j+3]*
  }
  y[i+0] = t1;
  y[i+1] = t2;
  y[i+2] = t3;
  y[i+3] = t4;
}
```

Loop optimizations (6)

Loop factoring

```
for (i=0; i<ARRAY_SIZE; i++) {
  a[i] = 0.;
  for (j=0; j<ARRAY_SIZE; j++)
    a[i] = a[i] + b[j]*d[j]*c[i];
}
```

```
for (i=0; i<ARRAY_SIZE; i++) {
  a[i] = 0.;
  for (j=0; j<ARRAY_SIZE; j++)
    a[i] = a[i] + b[j]*d[j];
  a[i] = a[i]*c[i];
}
```

Loop optimizations (7)

Further improvement of the previous example

```
t = 0.;
for (j=0; j<ARRAY_SIZE; j++)
  t = t + b[j]*d[j];

for (i=0; i<ARRAY_SIZE; i++)
  a[i] = t*c[i];
```

Loop optimizations (8)

Loop peeling

```
for (i=0; i<n; i++) {
  if (i==0)
    a[i] = b[i+1]-b[i];
  else if (i==n-1)
    a[i] = b[i]-b[i-1];
  else
    a[i] = b[i+1]-b[i-1];
}

a[0] = b[1]-b[0];
for (i=1; i<n-1; i++)
  a[i] = b[i+1]-b[i-1];
a[n-1] = b[n-1]-b[n-2];
```

Loop optimizations (9)

- The smaller the loop stepping stride the better
- Avoid using `if` inside loops

```
for (i=0; i<n; i++)
  if (j>0)
    x[i] = x[i] + 1;
  else
    x[i] = 0;

if (j>0)
  for (i=0; i<n; i++)
    x[i] = x[i] + 1;
else
  for (i=0; i<n; i++)
    x[i] = 0;
```

Loop optimizations (10)

Blocking: A strategy for obtaining spatial locality in loops where it's impossible to have small strides for all arrays

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    a[i][j] = b[j][i];
```

```
for (ii=0; ii<n; ii+=lot)   /* square blocking */
  for (jj=0; jj<n; jj+=lot)
    for (i=ii; i<min(n,ii+(lot-1)); i++)
      for (j=jj; j<min(n,jj+(lot-1)); j++)
        a[i][j] = b[j][i];
```

Other rules of optimization (1)

Factorization

```
xx = xx + x*a[i] + x*b[i] + x*c[i] + x*d[i];
```

```
xx = xx + x*(a[i] + b[i] + c[i] + d[i]);
```

Other rules of optimization (2)

Common expression elimination

```
s1 = a + c + b;
s2 = a + b - c;
```

```
s1 = (a+b) + c;
s2 = (a+b) - c;
```

Make it recognizable by compiler optimization

Other rules of optimization (3)

Strength reduction

- Replace floating-point division with inverse multiplication (if possible)
- Replace low-order exponential functions with repeated multiplications

```
y=pow(x,3);
```

```
y=x*x*x;
```

- Use of Horner's rule of polynomial evaluation

```
y=a*pow(x,4)+b*pow(x,3)+c*pow(x,2)+d*pow(x,1)+e;
```

```
y=(((a*x+b)*x+c)*x+d)*x+e;
```

Efficiency in the large

- What is efficiency?
- *Human efficiency* is most important for programmers
- *Computational efficiency* is most important for program users

Premature optimization

- “Premature optimization is the root of all evil” (Donald Knuth)
- F77 programmers tend to dive into implementation and think about efficiency in every statement
- “80-20” rule: “80” percent of the CPU time is spent in “20” percent of the code
- Common: only some small loops are responsible for the vast portion of the CPU time

Don’t think too much about efficiency before you have a thoroughly debugged and verified program!

Example of solving 1D heat equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

- Spatioal solution domain: $0 < x < 1$
- Temporal solution domain: $0 < t < T$
- Initial condition: $u(x,0) = I(x)$ prescribed
- Boundary condition: $u(0,t) = u(1,t) = 0$ always

An explicit finite difference scheme

- $M+2$ uniformly spaced spatial points: $x_0 = 0$, $x_{M+1} = 1$, $x_i = \frac{i}{M+1}$
- $u_i^\ell \approx u(x_i, \ell\Delta t)$
- Discretization:

$$\frac{u_i^{\ell+1} - u_i^\ell}{\Delta t} = \frac{u_{i-1}^\ell - 2u_i^\ell + u_{i+1}^\ell}{\Delta x^2}$$

Implementing 1D explicit heat equation solver

- Computation during one time step:
  $u_i^{\ell+1} = \rho(u_{i-1}^\ell + u_{i+1}^\ell) + (1-2\rho)u_i^\ell$ for $i = 1, 2, \ldots, M$, $\rho = \Delta t/\Delta x^2$
- We need two 1D arrays in a computer program: `u` refers to the $u^{\ell+1}$ vector, `u_prev` refers to the u^ℓ vector
- Enforce the initial condition

```
u_prev[0] = u_prev[M+1] = 0.0;

x = dx;
for (i=1; i<=M; i++) {
  u_prev[i] = I(x);
  x += dx;
}
```

Implementing the time loop

```
t = 0;

while (t<T) {
  t += dt;

  for (i=1; i<=M; i++)
    u[i] = rho*(u_prev[i-1]+u_prev[i+1])+(1.0-2.0*rho)*u_prev[i];

  /* boundary condition enforcement */
  u[0] = u[M+1] = 0.;

  /* data copy before next time step */
  for (i=0; i<=M+1; i++)
    u_prev[i] = u[i];
}
```

Optimizations

- We can avoid repeated computations of $1 - 2\rho$

```
double c_1_2rho = 1.0-2.0*rho;
/* ... */

for (i=1; i<=M; i++)
  u[i] = rho*(u_prev[i-1]+u_prev[i+1])+c_1_2rho*u_prev[i];
```

- We can avoid the copy between `u_prev` and `u` by simply switching the two pointers

```
double *tmp_pointer;
/* ... */

tmp_pointer = u_prev;
u_prev = u;
u = tmp_pointer;
```

Saving u to file

- Binary format

```
FILE *fp = fopen("u.bin","wb");
fwrite(u, sizeof(double), M+2, fp);
fclose(fp);
```

  File size: $8 \times (M + 2)$ bytes

- ASCII format

```
FILE *fp = fopen("u.txt","w");
for (i=0; i<=M+1; i++) {
  fprintf(fp, "u[%d]=%g\n",i,u[i]);
fclose(fp);
```

- The binary data file is both smaller in size and much faster to write and read!