Lecture 3: Performance of serial programs

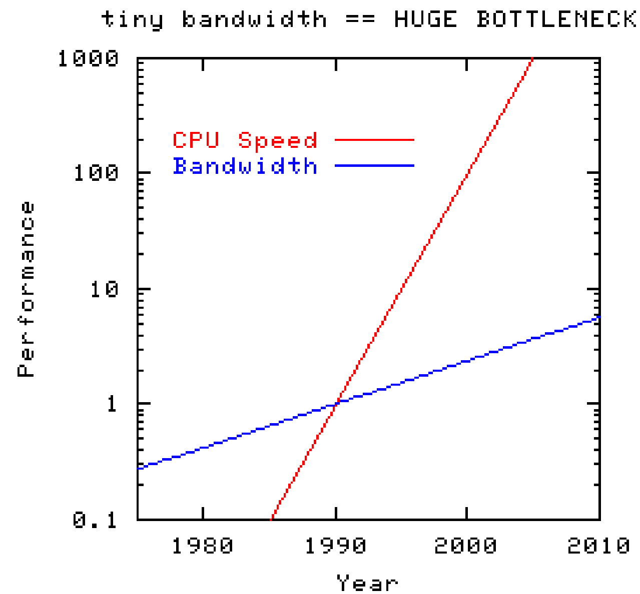
Motivations

- In essence, parallel computations consist of serial computations (executed on multiple computing units) and the needed 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
- In this lecture, we will take a look 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 2009, world's fastest computer—a Cray XT5 system named Jaguar—has 2.3 petaflops theoretical peak performance, making use of 224,162 AMD's Opteron 2.6 GHz processor cores
  - each core has 10.4 gigaflops peak performance
- 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 waiting for data from memory
- Scientific computations are oftem 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 bandwith 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 will be even worse
  - more memory reads and writes per operation
  - memory writes are usually slower than memory reads

Cache – a remedy for memory latency

- 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
  - located on-chip
  - typically of SRAM
  - small capacity
  - usually several levels of cache (L1, L2 and possibly 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

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

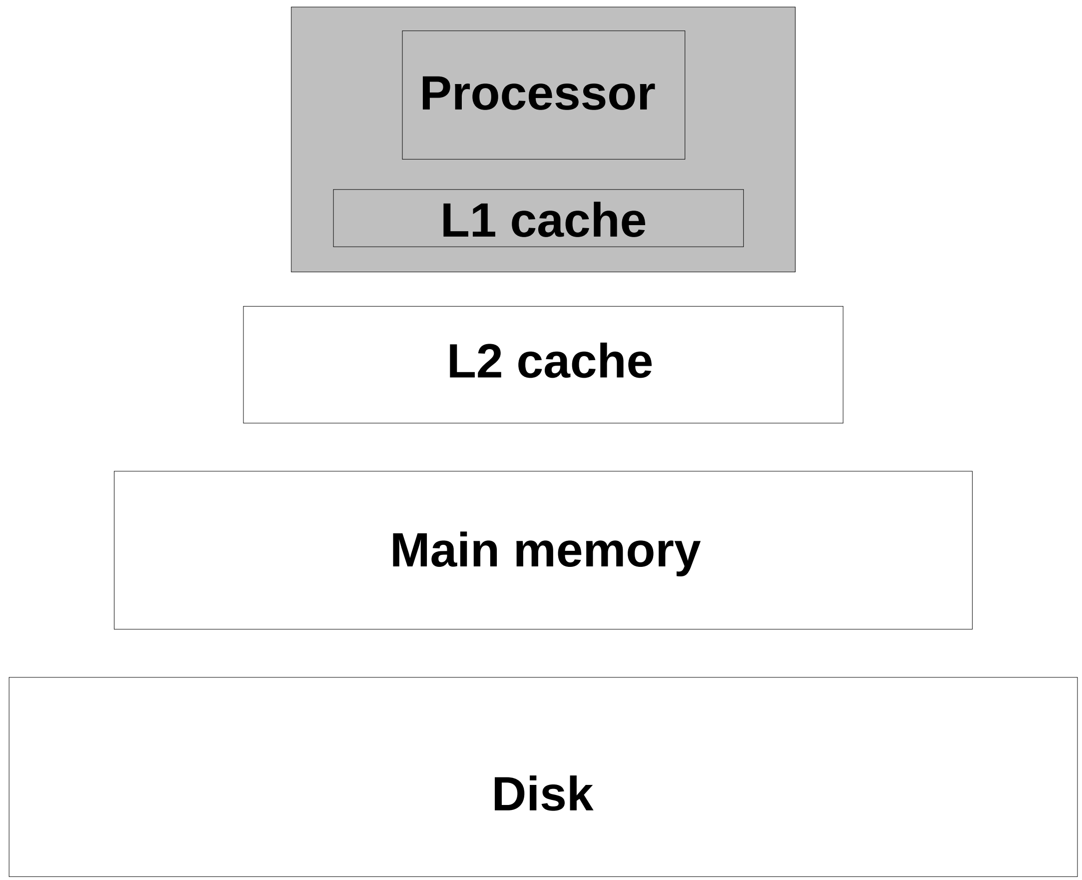
More about cache (2)

- 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



How to secure single-core performance?

- Effective use of cache
  - smart design of data structures (don’t waste memory)
  - 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
- Multithreading – some processors have hardware support to efficiently execute multiple threads on one core

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

Simple rules of efficiency (1)

- 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

Simple rules of efficiency (2)

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

Simple rules of efficiency (3)

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

Simple rules of efficiency (4)

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

Simple rules of efficiency (5)

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

Simple rules of efficiency (6)

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;
}
```

Simple rules of efficiency (7)

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];
}
```

Simple rules of efficiency (8)

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];
```

Simple rules of efficiency (9)

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];
```

Simple rules of efficiency (10)

- 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;
```

Simple rules of efficiency (11)

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];
```

Simple rules of efficiency (12)

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]);
```

Simple rules of efficiency (13)

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

Simple rules of efficiency (14)

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
- C++ and F90 force us to focus more on design

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

Example of solving 1D heat equation

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

- Solution domain: $0 < x < 1$
- Initial condition: $u(x, 0) = I(x)$
- boundary condition: $u(0, t) = u(1, t) = 0$

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
- Implement the initial condition

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

- Implement the main computation

```
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];
  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;
```

Solving 2D heat equation

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

- Solution domain: $(x, y) \in (0, 1) \times (0, 1)$
- Uniform mesh: $x_i = \frac{i}{M+1}$, $y_j = \frac{j}{N+1}$
- $u_{i,j}^{\ell} \approx u(x_i, y_j, \ell\Delta t)$
- Explicit finite difference discretization

$$\begin{aligned}
\frac{u_{i,j}^{\ell+1} - u_{i,j}^{\ell}}{\Delta t} &= \frac{u_{i-1,j}^{\ell} - 2u_{i,j}^{\ell} + u_{i+1,j}^{\ell}}{\Delta x^2} + \frac{u_{i,j-1}^{\ell} - 2u_{i,j}^{\ell} + u_{i,j+1}^{\ell}}{\Delta y^2} \\
u_{i,j}^{\ell+1} &= \rho(u_{i-1,j}^{\ell} + u_{i+1,j}^{\ell}) + \gamma(u_{i,j-1}^{\ell} + u_{i,j+1}^{\ell}) + \nu u_{i,j}^{\ell}
\end{aligned}$$

for $i = 1, 2, \ldots, M$ and $j = 1, 2, \ldots, N$, $\rho = \Delta t/\Delta x^2$, $\gamma = \Delta t/\Delta y^2$, $\nu = 1 - 2\rho - 2\gamma$

Implementing 2D explicit heat equation solver

- Use two 1D arrays

```
u = (double*)malloc((M+2)*(N+2)*sizeof(double));
u_prev = (double*)malloc((M+2)*(N+2)*sizeof(double));
```

- A two-layer `for`-loop for the main computation per time step

```
for (j=1; j<=N; j++)
  for (i=1; i<=M; i++) {
    index = j*(M+2)+i;
    u[index] = rho*(u_prev[index-1]+u_prev[index+1])
                +gamma*(u_prev[index-M-2]+u_prev[index+M+2])
                +nu*u_prev[index];
  }
```

Minor improvements

```
int offset = M+2;
/* ... */
index = offset;
for (j=1; j<=N; j++) {
  for (i=1; i<=M; i++) {
    ++index;
    u[index] = rho*(u_prev[index-1]+u_prev[index+1])
              +gamma*(u_prev[index-offset]+u_prev[index+offset])
              +nu*u_prev[index];
  }
  index += 2;
}
```

Saving u to file

- Binary format

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

  File size: $8(M+2)(N+2)$ bytes

- ASCII format

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

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

Exercises

- Write a simple C program that can be used to measure the size of the highest-level cache (typically L2) and the length of each cache line.
- Write a simple C program that illustrates the speed advantages of reading and writing binary data files, compared with ASCII data files.
- Write a simple C that compares between the handcoded copy operation between two arrays (`for (i=0; i<n; i++) b[i]=a[i]`) and using the standard `memcpy` function.
- Implement the explicit solver of the 3D heat equation $\frac{\partial u}{\partial t} = \kappa \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right)$ in the unit cube, where κ is a constant. You should use two 3D arrays `u[i][j][k]` and `u_prev[i][j][k]` which both have an underlying contiguous storage layout.
- Modify the above code by simply allocating u and `u_prev` as two very long 1D arrays. Do you notice any changes in the performance?

Exercises

- Make a theoretical estimate of the number of floating-point operations needed by the explicit 3D heat equation solver. What is the actual FLOPS rate achieved by your implementation?
- If κ is not constant, but a function $\kappa(x, y, z) = 1 + (x + y + z)/3$, what will the number of floating-point operations be then?
- Enforce a so-called "block" data structure for your explicit 3D heat equation solver. That is, instead of letting the values of u refer to the mesh points in a standard cyclic order, let u be a cyclically ordered sequence of small 3D blocks. In each block the respective u values are ordered cyclically.
  - Find out a mapping from the actual physical coordinates (x_i, y_j, z_k) to `u[index]`. We suppose n_x, n_y, n_z denote the number of mesh points in each spatial direction, and that $m_x \times m_y \times m_z$ is the size of each block.
  - Modify your implemention to use the above block data structure. Do you see any changes in the performance?