High Performance Coding in C/C++

Héctor Ferrada
académico

Instituto de Informática
Universidad Austral de Chile
INFO335 - High Performance Computing

1er Semestre, 2018

HPC in Perspective

- In short, HPC aims to implement efficient methods to speed up processing time in a computer model (CPU/GPU/Cluster architecture, shared/distributed memory).

HPC in Perspective

- In short, HPC aims to implement efficient methods to speed up processing time in a computer model (CPU/GPU/Cluster architecture, shared/distributed memory).
- This course focuses mainly in the study of PARALLEL ALGORITHMS for Multi-core, Cluster, GPU architectures.

HPC in Perspective

- In short, HPC aims to implement efficient methods to speed up processing time in a computer model (CPU/GPU/Cluster architecture, shared/distributed memory).
- This course focuses mainly in the study of PARALLEL ALGORITHMS for Multi-core, Cluster, GPU architectures.
- However HPC is much more than Parallel Computing.

HPC in Perspective

- In short, HPC aims to implement efficient methods to speed up processing time in a computer model (CPU/GPU/Cluster architecture, shared/distributed memory).
- This course focuses mainly in the study of PARALLEL ALGORITHMS for Multi-core, Cluster, GPU architectures.
- However HPC is much more than Parallel Computing.
- In particular, today we address this question. **How can we improve the performance of a code without necessarily changing the paradigm?**

Strategies to Optimize Code

- The current dominant HPC language is C/C++ $\Rightarrow$ *i.-* it's gives the programmer a very high degree of control (use of memory and also low-level operations). *ii.-* It's easily for GPU computing likethan OpenCL and CUDA.

Strategies to Optimize Code

- The current dominant HPC language is C/C++ $\Rightarrow$ *i.-* it's gives the programmer a very high degree of control (use of memory and also low-level operations). *ii.-* It's easily for GPU computing likethan OpenCL and CUDA.
- Integer arithmetic using bitwise operators $\Rightarrow$ shift bits, AND, OR, XOR, NOT.

Strategies to Optimize Code

- The current dominant HPC language is C/C++ $\Rightarrow$ *i.-* it's gives the programmer a very high degree of control (use of memory and also low-level operations). *ii.-* It's easily for GPU computing likethan OpenCL and CUDA.
- Integer arithmetic using bitwise operators $\Rightarrow$ shift bits, AND, OR, XOR, NOT.
- Function calls $\Rightarrow$ *i.-* to avoid include recursive routines, *ii.-* to pass arguments by reference.

Strategies to Optimize Code

- The current dominant HPC language is C/C++ $\Rightarrow$ *i.-* it's gives the programmer a very high degree of control (use of memory and also low-level operations). *ii.-* It's easily for GPU computing likethan OpenCL and CUDA.
- Integer arithmetic using bitwise operators $\Rightarrow$ shift bits, AND, OR, XOR, NOT.
- Function calls $\Rightarrow$ *i.-* to avoid include recursive routines, *ii.-* to pass arguments by reference.
- Individual data storage $\Rightarrow$ for instance, to use the precise required memory to allocate all the memory once if it is possible.

Strategies to Optimize Code

- The current dominant HPC language is C/C++ $\Rightarrow$ *i.-* it's gives the programmer a very high degree of control (use of memory and also low-level operations). *ii.-* It's easily for GPU computing likethan OpenCL and CUDA.
- Integer arithmetic using bitwise operators $\Rightarrow$ shift bits, AND, OR, XOR, NOT.
- Function calls $\Rightarrow$ *i.-* to avoid include recursive routines, *ii.-* to pass arguments by reference.
- Individual data storage $\Rightarrow$ for instance, to use the precise required memory to allocate all the memory once if it is possible.
- Decision strategies for local routines $\Rightarrow$ for instance, to use additional (very small) storage to acceletare the process.

Strategies to Optimize Code

- The current dominant HPC language is C/C++ $\Rightarrow$ *i.-* it's gives the programmer a very high degree of control (use of memory and also low-level operations). *ii.-* It's easily for GPU computing likethan OpenCL and CUDA.
- Integer arithmetic using bitwise operators $\Rightarrow$ shift bits, AND, OR, XOR, NOT.
- Function calls $\Rightarrow$ *i.-* to avoid include recursive routines, *ii.-* to pass arguments by reference.
- Individual data storage $\Rightarrow$ for instance, to use the precise required memory to allocate all the memory once if it is possible.
- Decision strategies for local routines $\Rightarrow$ for instance, to use additional (very small) storage to acceletare the process.
- Store precalculated values in variables to speed up final computations.

Strategies to Optimize Code

- The current dominant HPC language is C/C++ $\Rightarrow$ *i.-* it's gives the programmer a very high degree of control (use of memory and also low-level operations). *ii.-* It's easily for GPU computing likethan OpenCL and CUDA.
- Integer arithmetic using bitwise operators $\Rightarrow$ shift bits, AND, OR, XOR, NOT.
- Function calls $\Rightarrow$ *i.-* to avoid include recursive routines, *ii.-* to pass arguments by reference.
- Individual data storage $\Rightarrow$ for instance, to use the precise required memory to allocate all the memory once if it is possible.
- Decision strategies for local routines $\Rightarrow$ for instance, to use additional (very small) storage to acceletare the process.
- Store precalculated values in variables to speed up final computations.
- And any strategy that improves the current performance of the code (even if we have to change the paradigm).

Strategies to Optimize Code

- The current dominant HPC language is C/C++ $\Rightarrow$ *i.-* it's gives the programmer a very high degree of control (use of memory and also low-level operations). *ii.-* It's easily for GPU computing likethan OpenCL and CUDA.
- Integer arithmetic using bitwise operators $\Rightarrow$ shift bits, AND, OR, XOR, NOT.
- Function calls $\Rightarrow$ *i.-* to avoid include recursive routines, *ii.-* to pass arguments by reference.
- Individual data storage $\Rightarrow$ for instance, to use the precise required memory to allocate all the memory once if it is possible.
- Decision strategies for local routines $\Rightarrow$ for instance, to use additional (very small) storage to acceletare the process.
- Store precalculated values in variables to speed up final computations.
- And any strategy that improves the current performance of the code (even if we have to change the paradigm).

We must bear in mind that, in general, it's always is possible to improve a code.

Bitwise Operators in C/C++

- & (AND) Takes two numbers and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
- $|$ (OR) Takes two numbers and does OR on every bit of two numbers. The result of OR is 1 any of the two bits is 1.
- $\wedge$ (XOR) Takes two numbers and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
- $<<$ (left shift) Takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
- $>>$ (right shift) Takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
- $\backsim$ (bitwise NOT) Takes one number and inverts all bits of it

Bitwise Operators in C/C++. Example

Operator	Ese	Example
&	bitwise AND	1101 & 1001 = 1001
$\mid$	bitwise OR	$1010 \mid 1001 = 1011$
$\wedge$	bitwise XOR	$1101 \wedge 0011 = 0110$
$<<$	left shift	$0101 << 1 = 1010$
$>>$	right shift	$1101 >> 2 = 0011$
$\backsim$	Ones's compliment	$\backsim 1001 = 0110$

Read/Write Integers Using as few Bits as Possible

- The idea behind this is that using the exact memory can dramatically speed up the performance of a routine $\Rightarrow$ operating in smaller segments of memory which even works in cache.

Read/Write Integers Using as few Bits as Possible

- The idea behind this is that using the exact memory can dramatically speed up the performance of a routine $\Rightarrow$ operating in smaller segments of memory which even works in cache.
- Even sometimes it is prohibited to waste a full memory word per value $\Rightarrow$ being necessary to use secondary storage, which is orders of magnitude slower.

Read/Write Integers Using as few Bits as Possible

- The idea behind this is that using the exact memory can dramatically speed up the performance of a routine $\Rightarrow$ operating in smaller segments of memory which even works in cache.
- Even sometimes it is prohibited to waste a full memory word per value $\Rightarrow$ being necessary to use secondary storage, which is orders of magnitude slower.
- For instance. Suppose that we are working with long int numbers (8 bytes in C++).
  $\Rightarrow$ long int $*X$ = new long int[n];

Read/Write Integers Using as few Bits as Possible

- The idea behind this is that using the exact memory can dramatically speed up the performance of a routine $\Rightarrow$ operating in smaller segments of memory which even works in cache.
- Even sometimes it is prohibited to waste a full memory word per value $\Rightarrow$ being necessary to use secondary storage, which is orders of magnitude slower.
- For instance. Suppose that we are working with long int numbers (8 bytes in C++).
  $\Rightarrow$ long int $*X$ = new long int[n];
  but the maximum value for any x_i is M (a treadhold for the numbers in X) $\Rightarrow$ we do not need all the bytes to store $X[\,]$.

Read/Write Integers Using as few Bits as Possible

- The idea behind this is that using the exact memory can dramatically speed up the performance of a routine $\Rightarrow$ operating in smaller segments of memory which even works in cache.
- Even sometimes it is prohibited to waste a full memory word per value $\Rightarrow$ being necessary to use secondary storage, which is orders of magnitude slower.
- For instance. Suppose that we are working with long int numbers (8 bytes in C++).
  $\Rightarrow$ long int $*X$ = new long int[n];
  but the maximum value for any x_i is M (a treadhold for the numbers in X) $\Rightarrow$ we do not need all the bytes to store $X[\,]$.
  $\Rightarrow$ each x_i fits in $u = 1 + \lfloor \lg M \rfloor$ bits.

Read/Write Integers Using as few Bits as Possible

- The idea behind this is that using the exact memory can dramatically speed up the performance of a routine $\Rightarrow$ operating in smaller segments of memory which even works in cache.
- Even sometimes it is prohibited to waste a full memory word per value $\Rightarrow$ being necessary to use secondary storage, which is orders of magnitude slower.
- For instance. Suppose that we are working with long int numbers (8 bytes in C++).
  $\Rightarrow$ long int $*X$ = new long int[n];
  but the maximum value for any x_i is M (a treadhold for the numbers in X) $\Rightarrow$ we do not need all the bytes to store $X[\,]$.
  $\Rightarrow$ each x_i fits in $u = 1 + \lfloor \lg M \rfloor$ bits.
- How to read/write this $x_1 \cdots x_n$ numbers using only u bits per cell?

Write an Integer

```
// set the number x as a bitstring sequence in *A. In the range of bits
// [ini, .. ini+len-1] of *A. Here x has len bits
void setNum64(ulong *A, ulong ini, uint len, ulong x) {
   ulong i=ini>>BW64, j=ini-(i<<BW64);
   if ((j+len)>W64){
      ulong myMask = ~(~0ul >> j);
      A[i] = (A[i] & myMask) | (x >> (j+len-W64));      // OR: 1|0 = 0|1 = 1|1 = 1
      myMask = ~0ul >> (j+len-W64);
      A[i+1] = (A[i+1] & myMask) | (x << (WW64-j-len)); // AND: 1&1 = 1
   }else{
      ulong myMask = (~0ul >> j) ^ (~0ul << (W64-j-len)); // XOR: 1^1=0^0=0; 0^1=1^0=1
      A[i] = (A[i] & myMask) | (x << (W64-j-len));
   }
}
```

Where ...

const uint W64 = 64; // Word's bits

const uint BW64 = 6; // pow of two for W64

const uint WW64 = 128; // 2*W64

Write an Integer

```
// set the number x as a bitstring sequence in *A. In the range of bits
// [ini, .. ini+len-1] of *A. Here x has len bits
void setNum64(ulong *A, ulong ini, uint len, ulong x) {
   ulong i=ini>>BW64, j=ini-(i<<BW64);
   if ((j+len)>W64){
      ulong myMask = ~(~0ul >> j);
      A[i] = (A[i] & myMask) | (x >> (j+len-W64));      // OR: 1|0 = 0|1 = 1|1 = 1
      myMask = ~0ul >> (j+len-W64);
      A[i+1] = (A[i+1] & myMask) | (x << (WW64-j-len)); // AND: 1&1 = 1
   }else{
      ulong myMask = (~0ul >> j) ^ (~0ul << (W64-j-len)); // XOR: 1^1=0^0=0; 0^1=1^0=1
      A[i] = (A[i] & myMask) | (x << (W64-j-len));
   }
}
```

Where ...

const uint W64 = 64; // Word's bits

const uint BW64 = 6; // pow of two for W64

const uint WW64 = 128; // 2*W64

but... What is that? :(

Write an Integer

```
// set the number x as a bitstring sequence in *A. In the range of bits
// [ini, .. ini+len-1] of *A. Here x has len bits
void setNum64(ulong *A, ulong ini, uint len, ulong x) {
   ulong i=ini>>BW64, j=ini-(i<<BW64);
   if ((j+len)>W64){
      ulong myMask = ~(~0ul >> j);
      A[i] = (A[i] & myMask) | (x >> (j+len-W64));      // OR: 1|0 = 0|1 = 1|1 = 1
      myMask = ~0ul >> (j+len-W64);
      A[i+1] = (A[i+1] & myMask) | (x << (WW64-j-len)); // AND: 1&1 = 1
   }else{
      ulong myMask = (~0ul >> j) ^ (~0ul << (W64-j-len)); // XOR: 1^1=0^0=0; 0^1=1^0=1
      A[i] = (A[i] & myMask) | (x << (W64-j-len));
   }
}
```

Where ...

const uint W64 = 64; // Word's bits
const uint BW64 = 6; // pow of two for W64
const uint WW64 = 128; // 2*W64

but... What is that? :(... don't worry and see the whiteboard :)

Read the Integer

```
// return (in a unsigned long integer) the number in A from
// the bit position 'ini' to 'ini+len-1'
ulong getNum64(ulong *A, ulong ini, uint len){
   ulong i=ini>>BW64, j=ini-(i<<BW64);
   ulong result = (A[i] << j) >> (W64-len);

   if (j+len > W64)
      result = result | (A[i+1] >> (WW64-j-len));

   return result;
}
```

Now... Working with the code readWriteIntegers.cpp

Working in Class: Binary Search

Consider the following code that performs a binary seach:

```
// binary search for x on X[]
bool binarySearch(ulong *X, ulong n, ulong x, ulong *idx){
   if (x < X[0] || x > X[n-1])
      return 0;

   ulong l, r, m;

   l=0;
   r=n-1;
   m = r/2;

   while (l<=r){
      if (x==X[m]){
         *idx = m;
         return 1;
      }

      if (x<X[m])
         r=m-1;
      else
         l=m+1;

      m=l+(r-l)/2;
   }
   return 0;
}
```

Can we do better than that ?

Binary Search - Optimizations

So we can write a function, bool scanBSearch(ParProg *par, ulong x, ulong *idx), which implements some optimizations (you use the base tamplate binarySearch.cpp):

1. Note that, $m = r/2;$ and $m = l + (r - l)/2;$ can be write as: $m = r >> 1;$ and $m = (l + r) >> 1;$

Binary Search - Optimizations

So we can write a function, bool scanBSearch(ParProg *par, ulong x, ulong *idx), which implements some optimizations (you use the base tamplate binarySearch.cpp):

1. Note that, $m = r/2$; and $m = l + (r - l)/2$; can be write as: $m = r >> 1$; and $m = (l + r) >> 1$;
2. Note that the probability $x == X[m]$ is much lower than $x < X[m]$. So we change the code because a query is more expensive than a compute.

Binary Search - Optimizations

So we can write a function, bool scanBSearch(ParProg *par, ulong x, ulong *idx), which implements some optimizations (you use the base tamplate binarySearch.cpp):

1. Note that, $m = r/2$; and $m = l + (r - l)/2$; can be write as: $m = r >> 1$; and $m = (l + r) >> 1$;
2. Note that the probability $x == X[m]$ is much lower than $x < X[m]$. So we change the code because a query is more expensive than a compute.
3. To use the exact memory for the input array $A[1..n]$

Binary Search - Optimizations

So we can write a function, bool scanBSearch(ParProg *par, ulong x, ulong *idx), which implements some optimizations (you use the base tamplate binarySearch.cpp):

1. Note that, $m = r/2$; and $m = l + (r - l)/2$; can be write as: $m = r >> 1$; and $m = (l + r) >> 1$;
2. Note that the probability $x == X[m]$ is much lower than $x < X[m]$. So we change the code because a query is more expensive than a compute.
3. To use the exact memory for the input array $A[1..n]$
4. Additionally, we can change the method to look for x. If we create a table that enable us to perform a reduced binary search or a scan of a maximum number of cells in order to determine if x is in the set or is not.

Binary Search - Optimizations

So we can write a function, bool scanBSearch(ParProg *par, ulong x, ulong *idx), which implements some optimizations (you use the base tamplate binarySearch.cpp):

1. Note that, $m = r/2$; and $m = l + (r - l)/2$; can be write as: $m = r >> 1$; and $m = (l + r) >> 1$;
2. Note that the probability $x == X[m]$ is much lower than $x < X[m]$. So we change the code because a query is more expensive than a compute.
3. To use the exact memory for the input array $A[1..n]$
4. Additionally, we can change the method to look for x. If we create a table that enable us to perform a reduced binary search or a scan of a maximum number of cells in order to determine if x is in the set or is not.
5. Then, we perform a binary search if the amount of values in a fix segment is greater than a parameter $CELLS$; otherwise, we simply scan the segment from the sample value.

Experiments

We also execute expleriments that compute the average query-time for many repetitions (parameter REPET in the code; see the routines for experiments):

- The code will write a points file with the resume for each configuration of the program.
- We write a shell file to call the program and also to plot the points from the resume file (see the shell binarySearch.sh)
- We test two ways to create the input array $X[1..n]$. 1.- With a Normal distribution and 2.- With an uniform distribution.
- *As an additional task, we can calculate what is the minimum number of repetitions that we need to execute and ensure an error lower than 5% $\Rightarrow$ to do that we can compute the media online.

Other strategy (Homework)

In base of our code, we can coding the following strategy:

1. We store $X[]$ in differential way (called Gap-Encoding): $X'[1..n] = x_1, x_2 - x_1, x_3 - x_2, x_4 - x_3, ..., x_n - x_{n-1}$, where MAX is the maximum gap in $X' \Rightarrow$ we require only $\log MAX$ bits per element.
2. We include a sample table $S[1..n/s]$, which stores for each $s = O(\log n)$ (for instance $s = \frac{(\log n)^2}{2}$) positions a sample value of X.
3. Then, we perform a binary search on S to find the correct segment where should be x, after that we scan that segment to determine if x is in the set.