Introduction to Bioinformatics for Computer Scientists

Knowledge Quiz Slides
A review of the Questionnaire

- HPC
- Algorithms
HPC: some remarks

• HPC is nice
  → better algorithms are nicer
  → … and more energy-efficient

• Before parallelizing
  → optimize the sequential code first
  → frequently not done when comparing x86 code to some GPU implementation
  → learn the basics, before playing around with parallel HW
Program Optimization

- “The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet.”

- “Premature optimization is the root of all evil”
Profiling

- Profiling: collecting statistics from example executions
  - Provides an idea which routines/functions are critical
  - Common profiling approaches:
    - Instrument manually/automatically all procedure call/return points (potentially expensive)
    - Sampling PC every X milliseconds -- so long as program run is significantly longer than the sampling period, the accuracy of profiling is pretty good
- Profiling output

<table>
<thead>
<tr>
<th>Routine</th>
<th>% of Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>function_a</td>
<td>60%</td>
</tr>
<tr>
<td>function_b</td>
<td>27%</td>
</tr>
<tr>
<td>function_c</td>
<td>4%</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>function_zzz</td>
<td>0.01%</td>
</tr>
</tbody>
</table>

- Often over 80% of the time spent in less than 20% of the code (80/20 rule)
  - Code locality principle
  - Data locality principle
- More accurate profiling is possible with on-chip built-in HW counters and analysis tools
Using Gprof

- GNU profiler
- Uses PC sampling technique
- Add the `-pg` flag to the compiling step!
- Add the `-pg` flag to the linking step!
- Then just run the program as before
  - It will execute slower though!
- Slowdown for dense matrix-matrix multiplication (`mmult`):
  - Less than 1%
- Then type “gprof mmult”
## Gprof on \texttt{mmult}

Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td></td>
</tr>
<tr>
<td>99.83</td>
<td>156.90</td>
<td>156.90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.17</td>
<td>157.17</td>
<td>0.27</td>
<td>11184640</td>
<td>24.14</td>
</tr>
<tr>
<td>0.00</td>
<td>157.17</td>
<td>0.00</td>
<td>18</td>
<td>0.00</td>
</tr>
</tbody>
</table>

% the percentage of the total running time of the program used by this function.

.............
Using Valgrind

- Open source tool to find C/C++ memory leaks and a lot of other stuff!
- When compiling add `-g` to the compiler flags such that you can see the source lines where valgrind detects errors
- To look for memory leaks type: `valgrind --tool=memcheck --leak-check=yes -v` and then just the executable as you called it so far
- Also works with Pthreads-based programs
- Valgrind slowdown for `mmult` almost factor 10 (188 -> 1451 secs)!
- A comment about
  - `malloc(size_t size);`
  - `size_t` is an unsigned integer or an unsigned long integer depending on the system!
  - Frequently you may be multiplying 32-bit signed integers with each other in `malloc(int1 * int2 * int3)` and obtain a number of bytes that can not be represented by a signed integer any more
Valgrind on mmult

Total time 1451.554931
--13304-- REDIR: 0x40cd4b0 (memset) redirected to 0x4023d50 (memset)
==13304==
==13304== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 1)
--13304--
--13304-- supp: 13 dl-hack3-1
==13304== malloc/free: in use at exit: 0 bytes in 0 blocks.
==13304== malloc/free: 12,264 allocs, 12,264 frees, 134,264,640 bytes allocated.

Deliberate bug: c[matrixSize][matrixSize-1] = 0.0;

==14585== Invalid read of size 4
==14585== at 0x80488AC: main (mmult.c:124)
==14585== Address 0x41aa148 is 0 bytes after a block of size 64 alloc'd
==14585== at 0x4022AB8: malloc (vg_replace_malloc.c:207)
==14585== by 0x8048625: main (mmult.c:73)
==14585==
==14585== Invalid write of size 8
==14585== at 0x80488BC: main (mmult.c:124)
==14585== Address 0x78 is not stack'd, malloc'd or (recently) free'd
Optimization Cycle

- Look at code
- Profile code
- Change code manually
- Re-design algorithm?
- Play with compiler flags
- Execute code & measure time
- Confirm correctness

iterate
Optimization Cycle

- Look at code
- Profile code
- Change code manually
- Re-design algorithm?
- Play with compiler flags
- Execute code & measure time
- Confirm correctness

- Tricky for floating point numbers: 
  \[ a+b+c+d \neq b+d+a+c \]
BLAS: Basic Linear Algebra Subprograms

- Interface definition for linear algebra routines
- Several highly optimized implementations exist!
- Code optimization by using optimized libraries!
- Level 1 routines
  - Vector-vector ops
- Level 2 routines
  - Vector-matrix ops
- Level 3 routines
  - Matrix-matrix ops
- C-interface:
  - Functions preceded by `cblas_`, e.g., `cblas_dgemm()`
  - Arrays need to be contiguous in memory
  - Passed as pointers, not pointers of pointers
BLAS

- Implementations
  - GOTO-BLAS $\rightarrow$ usually the fastest one
  - ATLAS-BLAS $\rightarrow$ easier to install
  - Intel MKL Math Kernel Library
  - AMD AMCL
- Interfaces: matrix linearization:
  - $\rightarrow$ by rows or
  - $\rightarrow$ by columns?
- ATLAS has a C interface
Instruction Level Parallelism

- All processors since around 1985 use pipelining to overlap (and hence parallelize and accelerate) the execution of instructions.
- Goal: increase amount of parallelism and hence speed at a low (mostly HW) level.
- Two distinct approaches:
  - HW-based & dynamic
    - RISC: Reduced Instruction Set Computer/Computing
  - SW (compiler)-based very long instruction words VLIW & static branch prediction
    - Intel Itanium processor
- ILP represents very fine-grained low-level parallelism.
- When tuning performance you must keep in mind that there is a pipeline processor!
Levels of Parallelism

- ILP in a single pipeline
- SSE & AVX vector instructions on a single core
  - has anybody ever used vector intrinsics?
  - compilers are supposed to vectorize automatically
  - often this doesn't work
  - our experience: plain, good C code versus manually vectorized code: speedup factor 4.5 (with 256 bit AVX instructions)
- Simultaneous multi-threading/Hyper-threading
- Thread-based (OpenMP/Pthreads) on shared memory
  - multi-cores
  - mostly
    - loop-level parallelism
    - Fork-join paradigm
  - shared memory supercomputers
- MPI across nodes
  - high bandwidth/low latency network interconnect
  - current state-of-the art: Infiniband technology
  - closely coupled parallel codes
- Distributed computing, e.g., seti@home → no need for fast interconnect
NUMA

- Taxonomy of shared memory computers based on RAM address access times
- UMA: Uniform Memory Access
- NUMA: Non-Uniform Memory Access
  - all multi-cores
  - we actually have distributed shared memory!
NUMA

Distributed RAM blocks

Multi-core CPU

Interconnect
Get data at address x!
NUMA

Get data at address x!
We need to think about data locality when programming on a NUMA architecture.

Current systems use *first-touch* policy.

Page gets allocated to RAM block closest to the first core that does a writing access.

→ there also exist performance issues associated to *thread-to-core* pinning.
# Top500 list

- List of top 500 supercomputer systems in the world
- Benchmarking based on LINPACK linear algebra package

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DOE/NNSA/LLNL United States</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IEM</td>
</tr>
<tr>
<td>2</td>
<td>RIKEN Advanced Institute for Computational Science (AICS) Japan</td>
<td>K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu</td>
</tr>
<tr>
<td>3</td>
<td>DOE/SC/Argonne National Laboratory United States</td>
<td>Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IEM</td>
</tr>
<tr>
<td>4</td>
<td>Leibniz Rechenzentrum Germany</td>
<td>SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR iEM</td>
</tr>
<tr>
<td>5</td>
<td>National Supercomputing Center in Tianjin China</td>
<td>Tianhe-1A - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 NUDT</td>
</tr>
<tr>
<td>6</td>
<td>DOE/SC/Oak Ridge National Laboratory United States</td>
<td>Jaguar - Cray XK6, Opteron 6274 16C 2.20GHz, Cray Gemini interconnect, NVIDIA 2090 Cray Inc.</td>
</tr>
<tr>
<td>7</td>
<td>CINECA Italy</td>
<td>Fermi - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IEM</td>
</tr>
<tr>
<td>8</td>
<td>Forschungszentrum Juelich (FZJ) Germany</td>
<td>JuQUEEN - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IEM</td>
</tr>
<tr>
<td>9</td>
<td>CEA/TGCC-GENCI France</td>
<td>Curie thin nodes - Bullx B510, Xeon E5-2680 8C 2.700GHz, Infiniband QDR Bull</td>
</tr>
<tr>
<td>10</td>
<td>National Supercomputing Centre in Shenzhen (NSCS) China</td>
<td>Nebulae - Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050 Dawning</td>
</tr>
</tbody>
</table>
OpenMP

- **Open** specifications for **Multi-Processing**
- Easy semi-automatic parallelization of codes for shared memory systems
- Just insert so-called **pragmas** into the code to parallelize loops (mostly)
- Based upon PThreads
- Needs a dedicated OpenMP-enabled compiler
- In comparison to direct usage of PThreads
  - faster implementation
  - provides the programmer less control over what is happening
OpenMP Hello World

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int id,i;
    omp_set_num_threads(4);

    #pragma omp parallel for private(id)
    for (i = 0; i < 4; ++i)
    {
        id = omp_get_thread_num();

        printf("Hello World from thread %d\n", id);
    }

    return 0;
}
The Memory Gap

CPU Speed 40% per year

Memory Speed 9% per year

1980  2007
What we hope for

Small fast memory & big slow memory → ideally looks like a big fast memory

Given spatial & temporal code & data locality

System view

User view
Principles of Cache Memories & Paging Algorithms

- What I wanted to hear:
  - Temporal locality of data accesses
  - Spatial locality of data accesses
- The term cache comes from the French verb cacher (hiding)
  - Caches hide the memory latency
- If temporal and spatial data locality are not given, caches have no effect
  - random memory access patterns
  - e.g., accessing hash tables!
Cache Coherence

- In a shared memory system with multiple caches

- How do we make sure that copies of the same address $x$ in RAM that may reside in two or more caches are consistent when one cache copy is changed?
  - $\rightarrow$ cache coherency protocols

- We often talk about ccNUMA
  - $\rightarrow$ cache-coherent NUMA
Measuring Parallel Performance

- **Speedup**: $S = \frac{T(1)}{T(p)}$
- It can't be stated frequently enough
  - $T(1)$ actually refers to the fastest sequential implementation/algorithm available, everything else should be reported as **relative** speedup!
- **Scalability**: Up to how many CPUs do we get good speedups?
- We distinguish between:
  - *weak scaling*: scale up the problem size as we add cores
  - *strong scaling*: keep problem size fixed as we add cores
Super-linear speedups

- Reducing the per-CPU memory footprint of parallel programs via data distribution can yield super-linear speedups due to increased cache efficiency due to increased total cache size.

- Below: RAxML on an AMD Barcelona multi-core system:

![Graph showing super-linear speedups](image-url)
Example

- Sequential code total memory footprint of 20 MB
- Assume a machine with 2 cores and two caches of 10MB each
  → if we use one core, the working set of 20MB will not fit into cache
  → if we use two cores, the working set of 20 MB will entirely fit into the two caches
Amdahl's Law

At some point execution times will be dominated by the sequential part of the code.

Solutions:
- Let's pray for cache efficiency
- Optimize sequential part
- Parallel I/O
Amdahl's Law

- Scalability to large number of processors is limited by sequential part of program
- Every program has a sequential portion, even if it is just the time needed to start all the threads or MPI processes!
- More formally: Speedup ≤ \( \frac{1}{f + \frac{(1-f)}{p}} \) where \( f \) is the fraction of the program that must be executed sequentially
- Thus for \( p \rightarrow \infty \) the maximum speedup \( S_{\text{max}} \leq \frac{1}{f} \)
- If the fraction \( f \) is 0.01 we get \( S_{\text{max}} \leq 100 \) :-(
- IMPORTANT: here we are assuming linear speedups for the part that can be parallelized!

Solutions
- Hope that \( f \) is small
- Make \( f \) small
- May be counter-balanced by cache-efficiency!
Who cares about Amdahl's law anyway:

2. Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.

It is quite difficult to obtain high performance on a complete large-scale scientific application, timed from beginning of execution through completion. There is often a great deal of data movement and initialization that depresses overall performance rates. A good solution to this dilemma is to present results for an inner kernel of an application, which can be souped up with artificial tricks. Then imply in your presentation that these rates are equivalent to the overall performance of the entire application.
Twelve Ways to Fool the Masses when Giving Performance Results on Parallel Computers by David H. Bailey

Who cares about Amdahl's law anyway:

2. Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.

It is quite difficult to obtain high performance on a complete large-scale scientific application, timed from beginning of execution through completion. There is often a great deal of data movement and initialization that depresses overall performance rates. A good solution to this dilemma is to present results for an inner kernel of an application, which can be souped up with artificial tricks. Then imply in your presentation that these rates are equivalent to the overall performance of the entire application.

Full list at:
MPI API for parallelizing Code on Distributed Memory Architectures

- **Message Passing Interface**
- Send-Receive Paradigm
- Point to Point Communication
- Collective Communication
Communication Schemes

Direct/point-to-point

P0 -> P1

P0 -> P2

P0 -> P3

broadcast/multicast

indirect/reduction

P1 -> P0

P2 -> P0

P3 -> P0
Wave-Front Parallelism

- Dynamic Programming algorithms
Wave-Front Parallelism

- Dynamic Programming algorithms

Fill borders
Wave-Front Parallelism

- Dynamic Programming algorithms

Fill cells
Wave-Front Parallelism

- Dynamic Programming algorithms
Wave-Front Parallelism

- Dynamic Programming algorithms
Wave-Front Parallelism

- Dynamic Programming algorithms
Wave-Front Parallelism

- Dynamic Programming algorithms
Wave-Front Parallelism

• Dynamic Programming algorithms
Wave-Front Parallelism

- Dynamic Programming algorithms

Number of cells that can be computed in parallel proceeds like a wave-front through the matrix.
Hash Tables

- Map a universe of keys $U$ to a much smaller integer-based index table
- Lookup of elements in $O(1)$
- Challenge: define hash function
  - such that: it is fast to compute
  - such that: it does not map all keys to the same integer
- Handling collisions: two distinct keys are mapped to the same integer
  - resolve collisions by chaining
  - resolve collisions by re-hashing
- A sequence of hash table lookups will generally produce a sequence of random memory accesses
Binary Searches

• Search by comparison & bisection
• Able to find an element in $O(\log n)$
Kruskal & Prim

• Both invented a minimum spanning tree algorithm!