Techniques for portable high performance

Matteo Frigo

Quanta Research Cambridge

December 2, 2012

Message of this talk

Don't focus on machine details.

For many problems, portable programs exist that run on different machines as fast as programs tuned to each machine.

Portability is not hard.

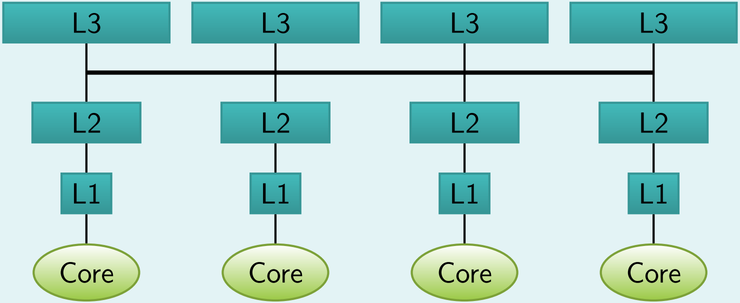
Such portable programs are easier to write than machine-tuned programs.

Outline

1. Portability and the memory hierarchy
2. Portability and parallelism
3. Autotuning
4. Conclusion

Modern CPU architecture

Xeon E312XX Sandy Bridge (oversimplified).



Programming challenges:

- Which cache(s) do you optimize for?
- Does the answer change if your program uses multiple cores?
- Will the answer change next year?

Cache-oblivious algorithms [FLPR99]

Goal:

Use the cache “optimally” without knowing the cache size.

Corollary:

- Simultaneously optimal at all levels of the memory hierarchy.
- Robust against shared caches, whose “effective size” varies.

Cache-oblivious Matrix Multiplication

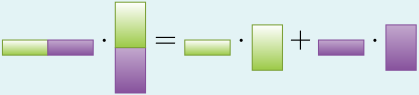
Base case:

If all matrices are 1×1, multiply them.

Recursive case:

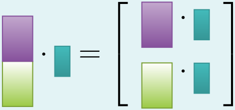
Otherwise, cut the largest dimension in half:

Case 1:



Case 2:

Case 3:



Analysis of cache misses

- At some point the problem becomes small enough to fit into cache.
  - This happens when $n^2 \approx$ cache size.
  - Yet, the algorithm does not know when this happens.
- Such small problems load each matrix element once into cache.
  - n^3 FLOPs for n^2 cache misses.
  - Or, $\sqrt{\text{cache size}}$ flops per cache miss.
- Thus, total cache misses $=$ work$/\sqrt{\text{cache size}}$.
- Matching lower bound [HK81].

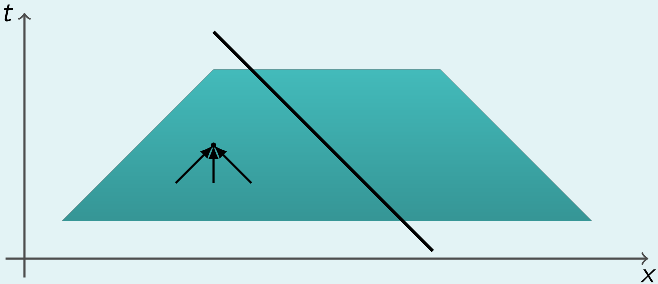
Cache-oblivious stencils

Three-point stencil:

$$u_x^{(t+1)} = K\left(u_{x-1}^{(t)},\ u_x^{(t)},\ u_{x+1}^{(t)}\right).$$

Cache-oblivious [Frigo and Strumpen 2005]:

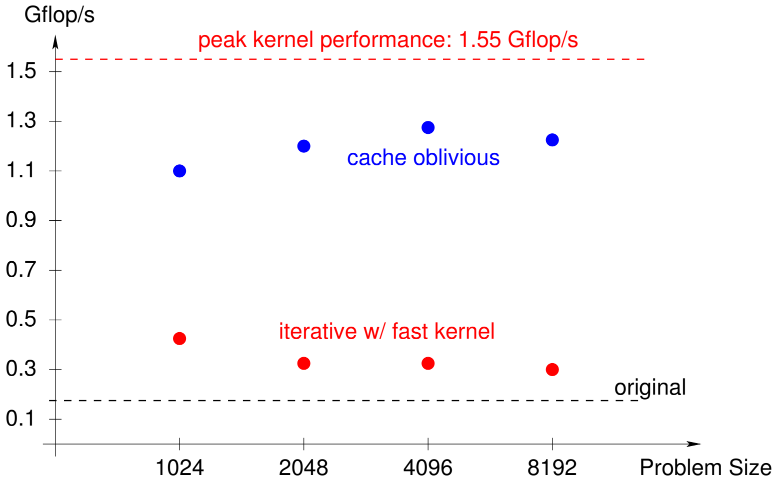
Recursive traversal of trapezoidal regions of spacetime.



LBMHD

- Lattice Boltzmann Magneto-HydroDynamics.
- Computes distribution of velocities of particles in a grid.
- 2D toroidal space.
- 13-point stencil.
- 27 double precision numbers per point.
- About 350 flops per stencil update.

Performance of LBMHD



One Power4+ processor, 1.45GHz, 32 KB L1, 1.5 MB L2, 32 MB L3, 32 GB main memory. Work done at IBM Austin Research Lab.

Other Cache-Oblivious Algorithms

Matrix Transposition/Addition $\Theta(1 + mn/\mathcal{B})$

Straightforward recursive algorithm.

Strassen's Algorithm $\Theta(n + n^2/\mathcal{B} + n^{\lg 7}/\mathcal{B}\mathcal{M}^{(\lg 7)/2-1})$

"Straightforward" recursive algorithm.

Fast Fourier Transform $\Theta(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n))$

Variant of Cooley-Tukey [CT65] using cache-oblivious matrix transpose.

Used in FFTW for register allocation.

LUP-Decomposition $\Theta(1 + n^2/\mathcal{B} + n^3/\mathcal{B}\mathcal{M}^{1/2})$

Recursive algorithm by Sivan Toledo [T97].

Sorting $\Theta(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n))$

Recursive $\sqrt{n}$-way mergesort via cache-oblivious "funnel" merger.

Etc.

Cholesky factorization, stencils, convolution, etc.

Cache-Oblivious Data Structures

Ordered-File Maintenance $O(1 + (\lg^2 n)/\mathcal{B})$

INSERT/DELETE anywhere in the file while maintaining $O(1)$-sized gaps. Amortized bound [BDFC00], later improved in [BCDFC02].

B-Trees

INSERT/DELETE: $O(1 + \log_{\mathcal{B}} n + (\lg^2 n)/\mathcal{B})$

SEARCH: $O(1 + \log_{\mathcal{B}} n)$

TRAVERSE: $O(1 + k/\mathcal{B})$

Solution [BDFC00] with later simplifications [BDIW02], [BFJ02].

Priority Queues $O(1 + (1/\mathcal{B}) \log_{\mathcal{M}/\mathcal{B}}(n/\mathcal{B}))$

Funnel-based solution [BF02]. General scheme based on buffer trees [ABDHMM02] supports INSERT/DELETE.

Moral of the story

Resist the urge of writing loops.

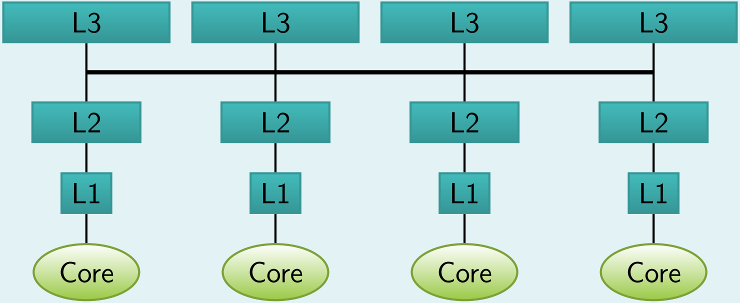
A recursive decomposition of the problem generally makes effective use of the memory subsystem.

Outline

1. Portability and the memory hierarchy
2. **Portability and parallelism**
3. Autotuning
4. Conclusion

No such thing as "number of cores"

Xeon E312XX Sandy Bridge (oversimplified).



Cores run at varying speeds:

- Sharing of caches.
- Nonuniform distance to caches.
- Unpredictable virtual memory mapping.
- Hyperthreading.
- Interrupts.
- System daemons.

Performance variability

2D 10000x10000 heat equation (5-point stencil)

	loop	**cache oblivious**
One process	17.5 s	10.4 s
Four concurrent processes	76.3 s	10.9 s
Saturating memory bus	277 s	19.9 s

(Xeon E31230, 4-core 3.2 GHz Sandy Bridge, 2×DDR3 1333)

Composable parallel software

How many cores should your target?

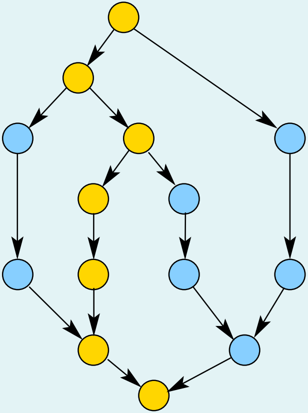
- You have 4 cores.
- You write your FFT library to use 4 threads.
- Your user calls your library from four different threads.
- Everything runs slow. (And you have wasted memory.)

Moral:

- Even if the hardware were perfect, you still cannot assume a given number of cores.

A simple theory of parallelism

Dependency graph:



Measures:

- T_P = execution time on P processors
- T_1 = work
- T_∞ = span

Maximum speedup:

$$\text{speedup} = T_1/T_P \leq T_1/T_\infty = \text{parallelism}.$$

"Reasonable" scheduler:

$$T_P \approx T_1/P + T_\infty.$$

“Processor-oblivious” programming

“Reasonable” scheduler:

$$T_P \approx T_1/P + T_\infty .$$

Corollary:

If the span T_∞ is small, then

$$T_P \approx T_1/P .$$

Moral:

- Use a reasonable scheduler.
- Express much more parallelism than you have cores. (That is, minimize the span.)
- Don’t worry about P.

The Cilk language and runtime system

Fibonacci in the Cilk language.

```
int fib(int n)
{
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n - 1);
    y = fib(n - 2);
    sync;
    return x + y;
  }
}
```

spawn is cheap:

- About 2–5× the cost of a procedure call.
- Cost of sync: about 0.

Work-stealing scheduler:

- Theoretically “optimal”.
- Efficient in practice.

Recommended parallel programming systems

Intel Cilk Plus:

- C/C++ support for fork/join parallelism.
- Cilkscreen for accurate detection of determinacy races.
- Cilkview for analyzing parallelism.
- Reducers for resolving certain race conditions in a lock-free manner.
- Matlab-style array notation for vector parallelism.
- Ships with the Intel Parallel Building Blocks.
- Also available in experimental gcc branch.

Other possibilities:

- Intel TBB.
- OpenMP tasks.

Outline

1. Portability and the memory hierarchy
2. Portability and parallelism
3. **Autotuning**
4. Conclusion

Algorithms for FFT of 16 points

$$\mathsf{FFT}(16) = \text{fastest of} \begin{cases} 2 \times \mathsf{FFT}(8) + 8 \times \mathsf{FFT}(2) \\ 4 \times \mathsf{FFT}(4) + 4 \times \mathsf{FFT}(4) \\ 8 \times \mathsf{FFT}(2) + 2 \times \mathsf{FFT}(8) \\ \text{maybe copying into contiguous buffer} \\ \text{maybe precomputing sin, cos} \\ \text{maybe using fused multiply-add } a \cdot b + c \\ \text{etc.} \end{cases}$$

Autotuners

Automatic search of the algorithmic space

- FFTW: Fourier transforms.
- SPIRAL: signal processing.
- ATLAS: matrix multiplication, LU.
- Sparsity: sparse matrix kernels.
- Berkeley stencil autotuner.

"When in doubt, use brute force."

Autotuning in FFTW

Search space:

- A transform of size $n = p \cdot q$ decomposes into multiple transforms of size p and q.
- Search the space of factorizations of n.
- Try different orders of execution of the subproblems.

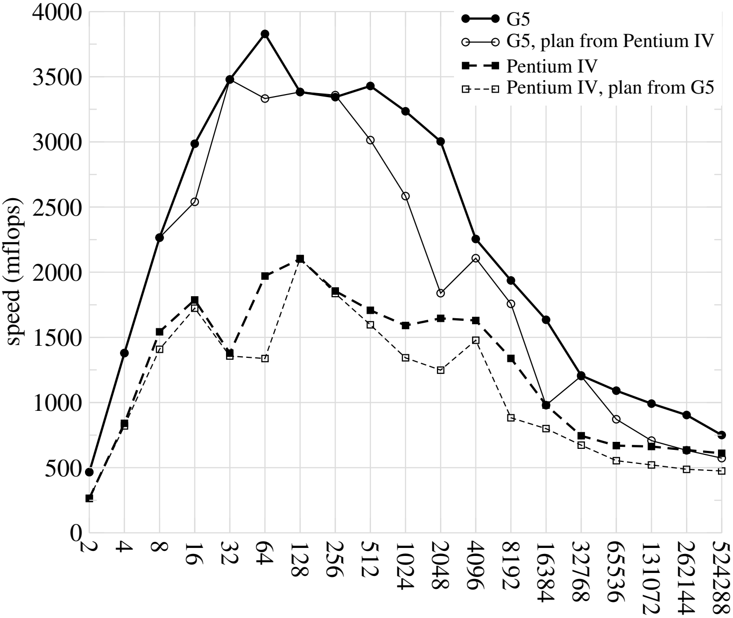
At compile time:

- A special-purpose compiler generates many variants of FFT "codelets" of small size.
- Performs various optimizations, including cache-oblivious scheduling for register allocation.

At run time:

- Measure multiple combinations of codelets, select the fastest.
- Purely empirical—no performance model.

Effect of autotuning in FFTW



Summary

Don’t target a specific memory hierarchy

Write cache-oblivious algorithms.

Don’t target a specific number of cores

Write processor-oblivious programs using Cilk or similar systems.

Don’t waste time tweaking low-level details

Write a code generator and search the tuning space automatically.