RED HAT :: SAN DIEGO :: 2007

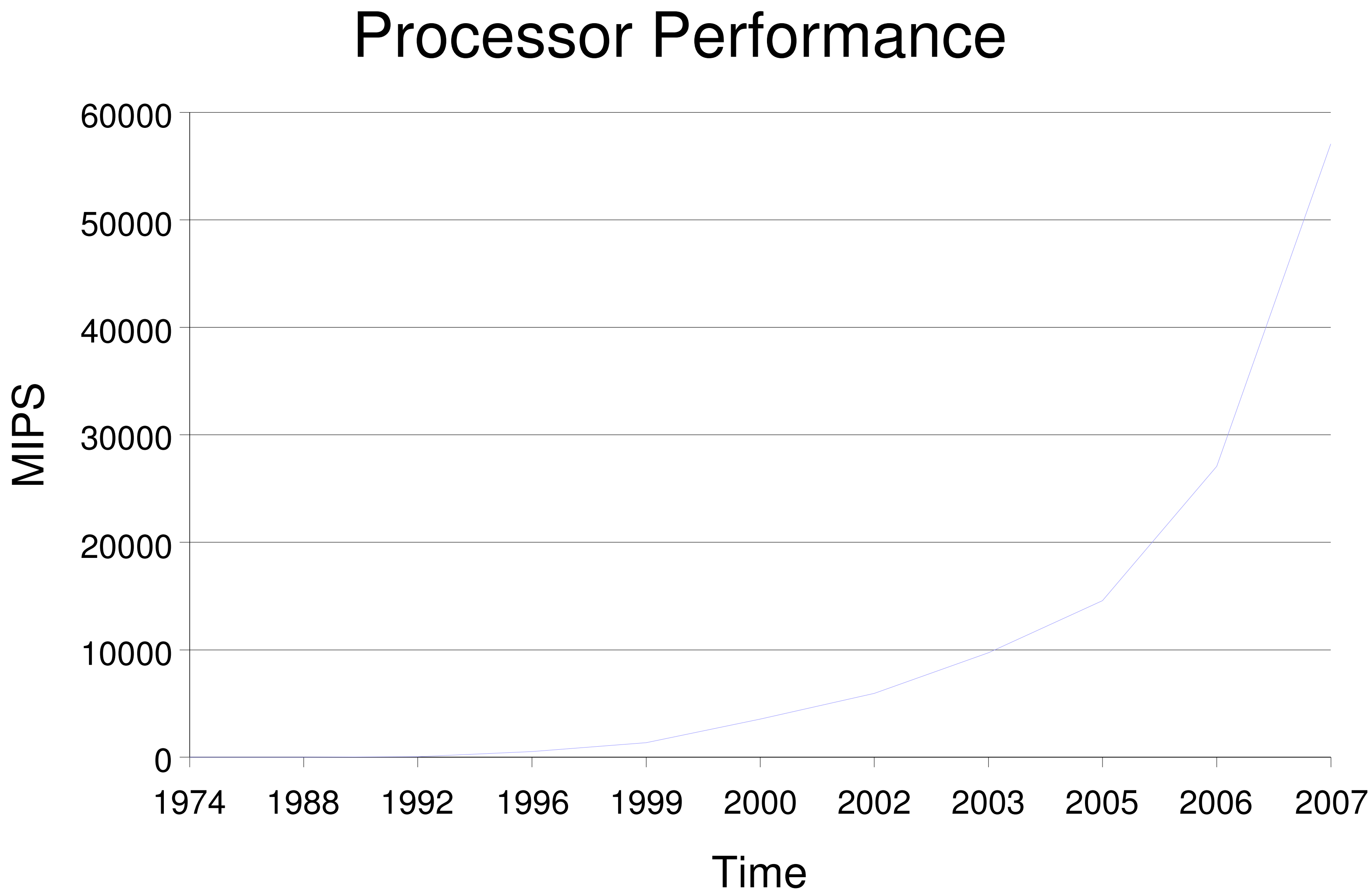
SUMMIT

Programming for tomorrow's high speed processors, today

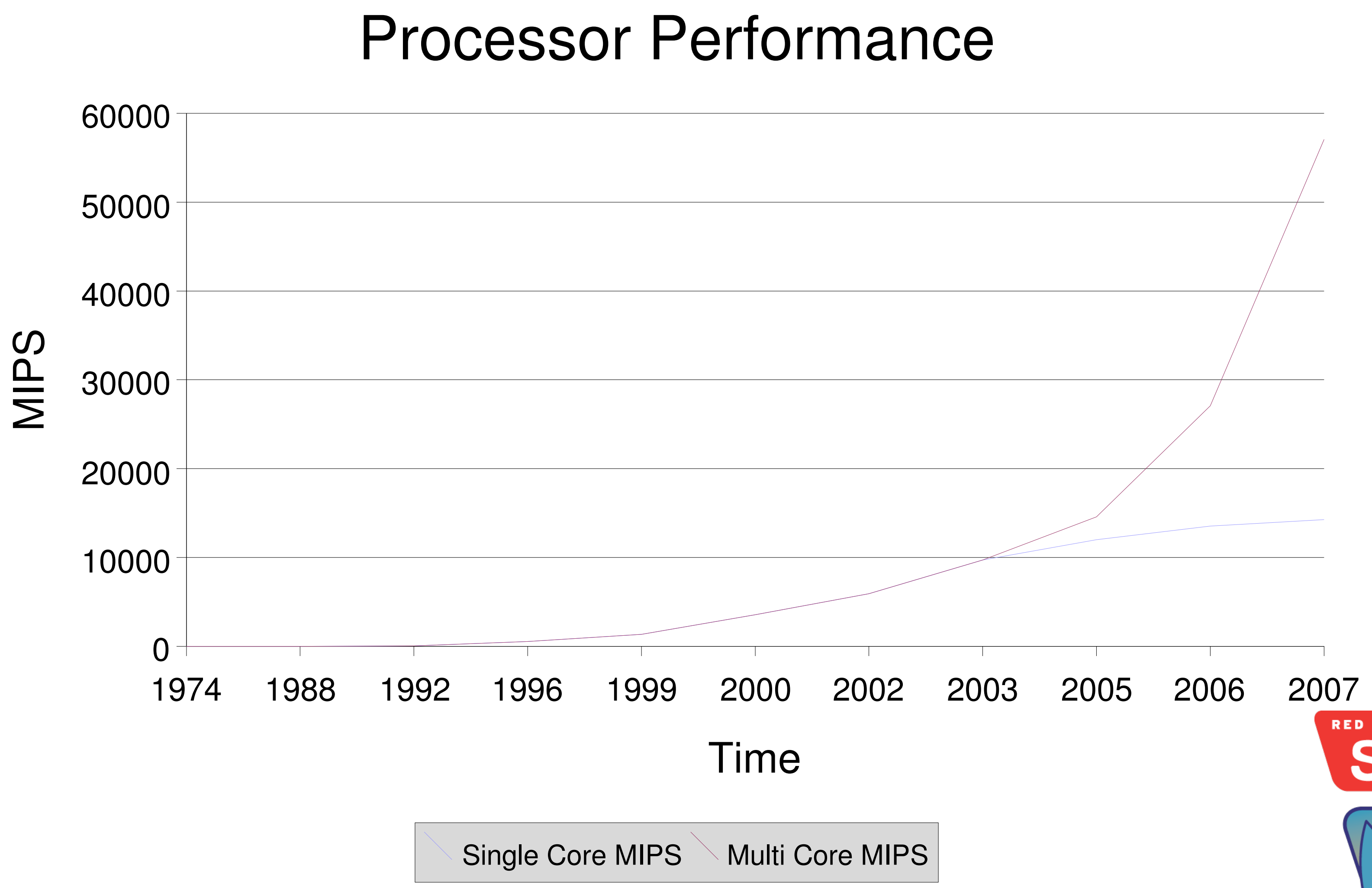
Ulrich Drepper

May 9th 2007

Programmers Do Not Need To Be Smart



The Big Problem of the next years



More Problems

- Numbers are inflated: realistic vs peak performance
- Peak performance only for stream instructions
  - Assuming full utilization of pipeline
  - No stalls due to memory / cache

- More typical:
  - Stream operations at 10% of peak
  - Normal operations at 2% of peak

Moore's Law and Dumb Programmers

- Moore's Law helped programmers so far
  - Almost all programs got faster with new hardware
  - No specific reorganization needed
  - Maybe recompilation for extra boost
- But no more:
  - Performance increases of cores flatten out
  - Hence dumb program increase increase flattens
- **Programmers must get smarter!**

What To Do?

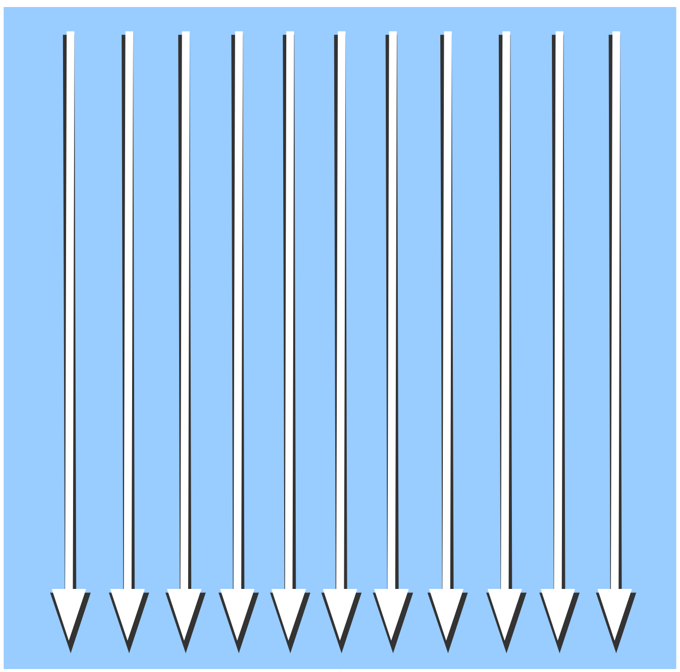
Only increase is parallelism can help:

- Exploit the pipeline
  - Data-parallelism
- Exploit the hyper-threads, cores, processors
  - Control-parallelism
- But: **Parallelism is hard!**
  - Hard to get right
  - Hard to get fast

Data-Parallelism

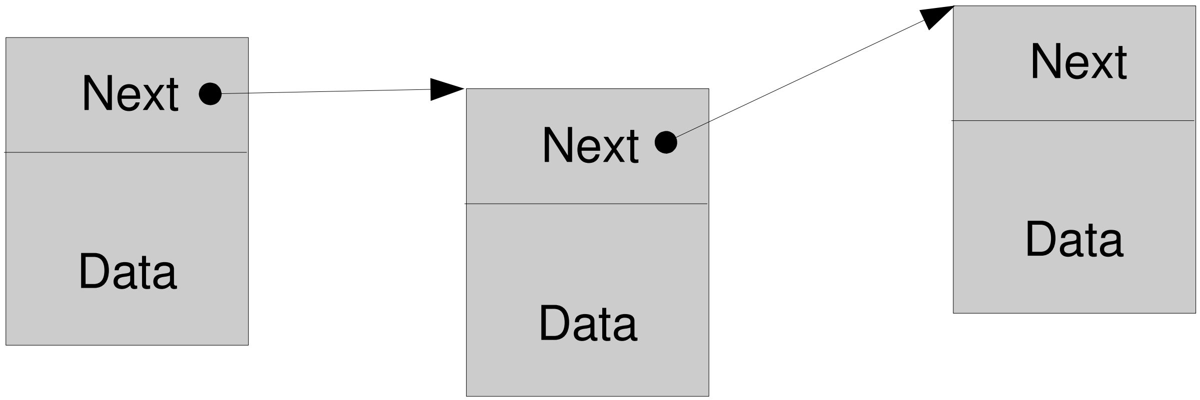
- Use pipelined instructions
  - Complex instructions with latency (multiplication)
  - Stream instructions
- Prerequisites:
  - Data must be available fast enough
  - Results must be written fast enough
- Prefetching must be efficient, cache misses create bubbles
- Data layout important
  - Sequential access in arrays
  - Random access with large lead times for prefetch
  - Efficient cache line usage

Memory Accesses



Consecutive accesses touch different cache lines

Memory Accesses



No locality

➔ No prefetching

Tricky and rarely usable software prefetching

Stream Operations

Simple matrix multiplication:

```
for (i = 0; i < N; ++i)
   for (j = 0; j < N; ++j) {
      double s = 0.0;
      for (k = 0; k < N; ++k)
         s += mul1[i][k] * mul2[k][j];
      res[i][j] = s;
   }
```

Stream Operations

Matrix Multiplication with stream operations:

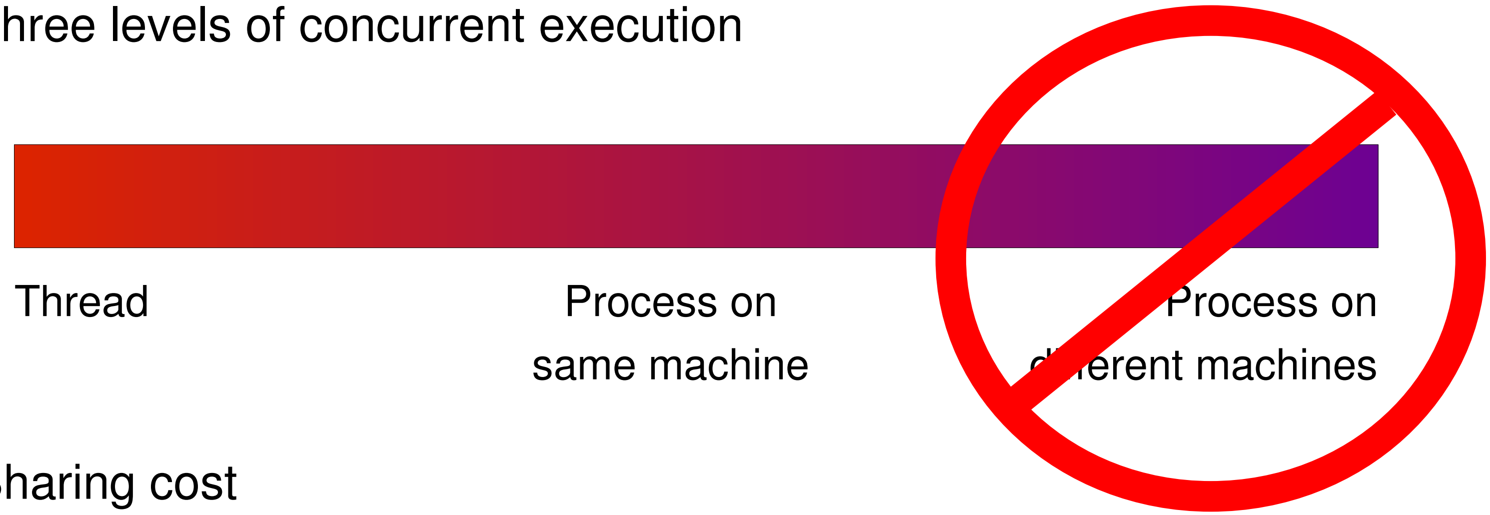
```
for (i= 0; i < N; i += 8)
   for (j = 0; j < N; j += 8)
      for (k = 0; k < N; k += 8)
         for (i2 = 0; i2 < 8; ++i2)
            for (k2 = 0; k2 < 8; ++k2) {
               __m128d m1d = _mm_load_sd(&mul1[i+i2][k+k2]);
               m1d = _mm_unpacklo_pd(m1d, m1d);
               for (j2 = 0; j2 < 8; j2 += 2) {
                  __m128d m2 = _mm_load_pd(&mul2[k+k2][j+j2]);
                  __m128d r2 = _mm_load_pd(&res[i+i2][j+j2]);
                  _mm_store_pd(&res[i+i2][j+j2],
                               _mm_add_pd(_mm_mul_pd(m2,m1d), r2));
               }
             }
```

Best Practices

- Create data types for the working set (alignment, etc)
- Not only for arithmetic operations:
  - Logical operations
  - min/max
  - Comparison
- Rearrange data (temporarily) to array form
- Transpose arrays (temporarily)
- Process arrays in chunk matching cache line sizes

Control-Parallelism

- Three levels of concurrent execution



- Sharing cost

  low ⟷ high

- Communication cost

  low ⟷ high

- Synchronization cost

  low ⟷ high

- Robustness

  low ⟷ high

Concurrency Levels

- Threads:
  - All share the same address space
    - No inter-process communication needed
  - All die together
  - Can scribble over each other's memory
- Processes:
  - Separate address spaces with connections through shared memory
  - Completely separate lifetimes
  - Different address space layout (pointers are problematic)
- Performance:
  - In Linux scheduling about the same
  - Synchronization intra-process will be a bit faster

Use Processes if...

- Amount of modified shared data is limited
  - Read-only data can be mapped in multiple-processes with little cost
  - Fixed size random-access data placed in shared memory
    - Coordinate access
    - Atomic updates
  - Best: data stream
    - Pipes are fast, even faster in RHEL5
      - `vmsplice()`, `splice()`, `tee()` system calls
  - Robustness is key
    - Synchronization possible with robust mutexes

Use Threads if...

- Large amounts of data have to be shared
- Not easy to partition data for different processes
- Frequent creation/destruction of new concurrent control flow
- Equivalent: short-lived concurrency needed

Programming Models

- Processes are mostly single threaded code
  - No special no knowledge needed for that
  - Synchronization only needed for shared resources
    - Synchronization objects in shared memory
    - Atomic operations
- Threads require more work
  - Changes and overhead to old code introduced by POSIX.1c
  - More shared means more synchronization
  - Many problem lure in new and old code
  - Pthread model too complex

Need to find something better...

Parallelism In The Language

- Today: OpenMP
  - No explicit creation of thread
  - Code can be used without threads
    - Or: non-threaded code can be parallelized without many changes
  - Compiler gets told about concurrency
    - Optimizations can take this into account
  - More like parallelism as taught
- Tomorrow: more parallelism constructs in language (Parallel C)
- Alternative: data structure implementations implicitly using parallelism

OpenMP

- Implicit thread creation. Number of threads:
  - Programmer configurable
  - User configurable
  - Dynamic based on hardware and configuration
- OpenMP runtime maintains thread pool (amortized startup)
- Iteratively add more and more directives
- Does not collide with other thread use

OpenMP

Normal C code:

```
void avg(int n, float a[n], float b[n]) {
  int i;

  b[0] = (0 + a[0]) / 2;

  for (i = 1; i < n; ++i)
    b[i] = (a[i – 1] + a[i]) / 2.0
}
```

OpenMP

OpenMP C code:

```
void avg(int n, float a[n], float b[n]) {
   int i;

   b[0] = (0 + a[0]) / 2;
#pragma omp parallel for
   for (i = 1; i < n; ++i)
      b[i] = (a[i – 1] + a[i]) / 2.0
}
```

OpenMP

Normal C code:

```
int fct(int a, int b) {
  int r1, r2, r3;




    r1 = fct1(a);

    r2 = fct2(b);

    r3 = fc3(a, b);

  return r1 + r2 + r3;
}
```

OpenMP

OpenMP C code:

```
int fct(int a, int b) {
  int r1, r2, r3;
#pragma omp parallel sections
  {
#pragma omp section
     r1 = fct1(a);
#pragma omp section
     r2 = fct2(b);
#pragma omp section
     r3 = fc3(a, b);
  }
  return r1 + r2 + r3;
}
```

Future Development

- Co-processors are coming back
  - Intel Geneseo, AMD Torrenza
  - IBM Cell
- Huge performance advantage through specialization:
  - All purpose CPU: 50-60 GFLOPS
  - Cell: 210 GFLOPS
  - NVidia GPU: 500 GFLOPS
- Need special programming

Summary

- Use data-parallelism to reach peak performance
- Encapsulate implementation to allow co-processor use
- Use control-parallelism to benefit from future hardware upgrades
- Use programming models which
  - Provide safest, most robust environment for least cost
  - Helps developers by preventing many bugs

Questions?