Why is performance productivity poor on modern architectures?

Dagstuhl Seminar on Petacomputing, Feb 13–17, 2006

Jan Treibig\textsuperscript{1}  
Georg Hager\textsuperscript{2}

\textsuperscript{1}Lehrstuhl für Systemsimulation, FAU Erlangen-Nürnberg
\textsuperscript{2}Regionales Rechenzentrum Erlangen, FAU Erlangen-Nürnberg

February 16, 2006
Contents

1 Introduction
   - Going parallel as a popular pastime
   - The OpenMP story

2 Problems with using new paradigms — a Case Study
   - The C++/OpenMP/ccNUMA mess
   - Implementing first touch the easy way

3 Performance (un)productivity
   - Getting bad performance the easy way
   - Lessons learned from the high-level approach

4 No way out?
   - Getting good performance — the hard way

5 Machine Analysis

6 Example 1: Getting Memory Bandwidth
   - Memcpys
   - Stream Triad Benchmark

7 Example 2: Red-Black Gauss-Seidel Smoother
   - Itanium2: Software Pipelined loops
Parallelization techniques well established among HPC “power users”

Don’t forget: Petacomputing == gigacomputing at the serial level!

New users forced into parallelization with obstacles to take:
- Multi-/many-core
- ccNUMA
- Network topologies
- Efficient I/O
- Strange architectures
- Stupid compilers
- Lack of mature and “simple” tools
- . . . you name it!

Interesting paradigm: “Constellation” clusters
- Motivation: 30/60 TFlop/s SGI Altix 4000 to be installed in Bavaria very soon
- Promising strategy: Choose the “easiest” path and use OpenMP if possible!
OpenMP?

- Incremental parallelism, serial equivalence
- Good language support, at least in theory
  - Lots of bugs, esp. in C++ compilers
- Easy to learn, hard to master
- Flexible enough to emulate “minimalistic MPI”
- DSM variants available (Cluster OpenMP . . . )
- Advanced tools for correctness checking
- Thread safety ↔ performance issues?
C++ is all about objects and templates, and it should stay that way when doing parallel programming.

Problem: Constructors usually called in a serial region. Test case:

class D {
    double d;
public:
    D(double _d=0.0) throw() : d(_d) {}
    inline D operator+(const D& o) throw() {
        return D(d+o.d);
    }
    [...] 
};

D* A = new array[20000];
C++ is all about objects and templates, and it should stay that way when doing parallel programming.

Problem: Constructors usually called in a serial region. Test case:

class D {
    double d;
public:
    D(double _d=0.0) throw() : d(_d) {}  
    inline D operator+(const D& o) throw() {
        return D(d+o.d);
    }
    [...]
};

D* A = new array[20000]; // Locality problem!
The C++/OpenMP/ccNUMA mess

- C++ is all about objects and templates, and it should stay that way when doing parallel programming
- Problem: **Constructors** usually called in a serial region. Test case:

```cpp
class D {
    double d;
public:
    D(double _d=0.0) throw() : d(_d) {}
    inline D operator+(const D& o) throw() {
        return D(d+o.d);
    }
    [...]  
};
[...] 
D* A = new array[20000]; // Locality problem!
```

- Solution: Use parallel “first touch” in allocation
Correct placement by allocating with “first touch”:

```cpp
template <class T> T* pnew(size_t n) {
    [...]
    char *p = new char[len];
    #pragma omp parallel for schedule(static) private(ofs)
    for(i=0; i<pages; ++i) {
        ofs = static_cast<size_t>(i) « PAGE_BITS;
        p[of]=0;
    }
    for(ofs=0; ofs<n; ++ofs) {
        new(static_cast<void*>(p+ofs*st)) T;
    }
    return static_cast<T*>(p);
}
```
Implementing first touch the easy way

- Correct placement by allocating with “first touch”:

```cpp
template <class T> T* pnew(size_t n) { 
  [...] 
  char *p = new char[len];
  #pragma omp parallel for schedule(static) private(ofs)
  for(i=0; i<pages; ++i) {
    ofs = static_cast<size_t>(i) « PAGE_BITS;
    p[ofs]=0;
  }
  for(ofs=0; ofs<n; ++ofs) {
    new(static_cast<void*>(p+ofs*st)) T;
    // placement new
  }
  return static_cast<T*>(p);
}
```
But we want to do *real* C++ and use STL, in a user-friendly way.

Solution: Design a **NUMA-aware STL allocator**

Example (STL NUMA allocator)

```cpp
template <class T> class NUMA_Allocator {

public: [...] 
    T* allocate(size_type n, const void *lH=0) {
        size_type ofs,len = n*sizeof(T);
        char *p = malloc(len);
        #pragma omp parallel for schedule(static) private(ofs)
            [ ... same as before ...]
    }
    void construct(T* p, const T& x) {
        new(p) value_type(x);
    }
    void destroy(T* p) { p->~T(); }
};
```
Getting bad performance the easy way

- Now we can do
  ```
  vector<double, NUMA_ALLOCATOR<double> > A(20000);
  ```
  and use A in OpenMP loops and be happy. Or can’t we?
- **Performance penalties** of 1-thread vector triad with respect to “vanilla” version for out-of-cache data set:

<table>
<thead>
<tr>
<th></th>
<th>S,op</th>
<th>S,it</th>
<th>S,op,O</th>
<th>S,it,O</th>
<th>d,O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel V9 IA64</td>
<td>0.50</td>
<td>0.99</td>
<td>0.25</td>
<td>0.28</td>
<td>0.98</td>
</tr>
<tr>
<td>Intel V9 EM64T</td>
<td>0.78</td>
<td>0.80</td>
<td>0.64</td>
<td>0.79</td>
<td>1.00</td>
</tr>
<tr>
<td>PGI x86_64</td>
<td>0.53</td>
<td>0.90</td>
<td>0.47</td>
<td>0.68</td>
<td>0.79</td>
</tr>
<tr>
<td>Pathscale x86_64</td>
<td>0.87</td>
<td>0.87</td>
<td>0.81</td>
<td>0.87</td>
<td>1.00</td>
</tr>
<tr>
<td>MIPSPro MIPS</td>
<td>0.78</td>
<td>1.00</td>
<td>0.84</td>
<td>0.95</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Legend: S=STL, op=operator[], it=iterator, O=OpenMP
Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
- Should we abandon STL, or even C++, for low-level operations?
Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
- Should we abandon STL, or even C++, for low-level operations?
- Generally speaking, OpenMP is bad for serial performance
Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
- Should we abandon STL, or even C++, for low-level operations?
- Generally speaking, OpenMP is bad for serial performance
  - Not only in cache, but also for streaming
Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`

- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.

- Should we abandon STL, or even C++, for low-level operations?

- Generally speaking, OpenMP is bad for serial performance
  - Not only in cache, but also for streaming
  - Some issues with NT stores on x86, but in general?
Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
- Should we abandon STL, or even C++, for low-level operations?
- Generally speaking, OpenMP is bad for serial performance
  - Not only in cache, but also for streaming
  - Some issues with NT stores on x86, but in general?
- Should we start from scratch and dump the shared memory approach altogether?
Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
- Should we abandon STL, or even C++, for low-level operations?
- Generally speaking, OpenMP is bad for serial performance
  - Not only in cache, but also for streaming
  - Some issues with NT stores on x86, but in general?
- Should we start from scratch and dump the shared memory approach altogether?
- Are we tackling tomorrow’s performance challenges with yesterday’s tools?
Observation:

The memory wall is not the only problem we are facing. After algorithmic and data layout changes the mapping of the high level language to the ISA is an important issue.

Compilers have difficulties to utilize the performance of modern CPUs. Reasons are:

- Still the well-known issue that caches are not transparent with regard to performance
- Developements in modern CPU architectures:
  - Prefetching
  - SIMD
  - Special instructions: e.g. Non temporal stores
- General code quality (address calculation, register scheduling)
## Architectural Overview

<table>
<thead>
<tr>
<th>Feature</th>
<th>Intel Prescott</th>
<th>AMD Athlon64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock speed</td>
<td>3.2 GHz</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>Cacheline length</td>
<td>64(128) Byte</td>
<td>64 Byte</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>1 MByte</td>
<td>1 MByte</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>16 kByte</td>
<td>64 Byte</td>
</tr>
<tr>
<td>L2 Latency</td>
<td>56 cycles (min 21)</td>
<td>13 cycles (min 11)</td>
</tr>
<tr>
<td>L2 Read Bandwidth</td>
<td>23 GB/s</td>
<td>13 GB/s</td>
</tr>
<tr>
<td>L2 Write Bandwidth</td>
<td>12 GB/s</td>
<td>12 GB/s</td>
</tr>
<tr>
<td>L1 Latency</td>
<td>4 cycles (min 1)</td>
<td>3 cycles (min 2)</td>
</tr>
<tr>
<td>L1 Read Bandwidth</td>
<td>46 GB/s</td>
<td>35 GB/s</td>
</tr>
<tr>
<td>L1 Write Bandwidth</td>
<td>12 GB/s</td>
<td>35 GB/s</td>
</tr>
<tr>
<td>Memory Read Bandwidth</td>
<td>5.8 GB/s</td>
<td>6.1 GB/s</td>
</tr>
<tr>
<td>Memory Write Bandwidth</td>
<td>4.1 GB/s</td>
<td>6.1 GB/s</td>
</tr>
</tbody>
</table>
Cacheread 16 byte loads

- Intel Pentium 4 Prescott
- AMD Athlon64 4000+
Memory Hierarchy Cache Bandwidth

Cacheread 8 byte loads

![Graph showing cache read performance for Intel Pentium 4 Prescott and AMD Athlon64 4000+ processors](image-url)
Cachewrite 16 byte stores

Memory Hierarchy Cache Bandwidth

- Graph showing cache write performance for Intel Pentium 4 and AMD Athlon 64 processors.
- Bandwidth measured in MB/s against cache size in KByte.
Memory Hierarchy Cache Bandwidth

Cachewrite 16 byte stores

- AMD Athlon 64
- Intel Pentium 4
### Peak Performance

<table>
<thead>
<tr>
<th></th>
<th><strong>Pentium 4</strong></th>
<th><strong>Athlon64</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>2920 Mflops (44.6 %)</td>
<td>2338 Mflops (48.7 %)</td>
</tr>
<tr>
<td><strong>MultAdd</strong></td>
<td>23.3 GByte/s</td>
<td>18.7 GByte/s</td>
</tr>
<tr>
<td>Add 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 2 var</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Peak Performance

<table>
<thead>
<tr>
<th></th>
<th>Pentium 4</th>
<th>Athlon64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add</strong></td>
<td>2920 Mflops (44.6%)</td>
<td>2338 Mflops (48.7%)</td>
</tr>
<tr>
<td><strong>MultAdd</strong></td>
<td>23.3 GByte/s</td>
<td>18.7 GByte/s</td>
</tr>
<tr>
<td></td>
<td>5229 MFlops (81%)</td>
<td>4153 MFlops (86%)</td>
</tr>
<tr>
<td><strong>Add 2</strong></td>
<td>20.9 GByte/s</td>
<td>16.1 GByte/s</td>
</tr>
<tr>
<td><strong>Add 2 var</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Peak Performance

<table>
<thead>
<tr>
<th></th>
<th>Pentium 4</th>
<th>Athlon64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add</strong></td>
<td>2920 Mflops (44.6 %)</td>
<td>2338 Mflops (48.7 %)</td>
</tr>
<tr>
<td><strong>MultAdd</strong></td>
<td>23.3 GByte/s</td>
<td>18.7 GByte/s</td>
</tr>
<tr>
<td></td>
<td>5229 MFlops (81 %)</td>
<td>4153 MFlops (86 %)</td>
</tr>
<tr>
<td><strong>Add 2</strong></td>
<td>2339 MFlops (36 %)</td>
<td>1187 (24 %)</td>
</tr>
<tr>
<td><strong>Add 2 var</strong></td>
<td>37.4 GByte/s</td>
<td>19.0 GByte/s</td>
</tr>
</tbody>
</table>
# Peak Performance

<table>
<thead>
<tr>
<th></th>
<th>Pentium 4</th>
<th>Athlon64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add</strong></td>
<td>2920 Mflops (44.6 %)</td>
<td>2338 Mflops (48.7 %)</td>
</tr>
<tr>
<td></td>
<td>23.3 GByte/s</td>
<td>18.7 GByte/s</td>
</tr>
<tr>
<td><strong>MultAdd</strong></td>
<td>5229 MFlops (81 %)</td>
<td>4153 MFlops (86 %)</td>
</tr>
<tr>
<td></td>
<td>20.9 GByte/s</td>
<td>16.1 GByte/s</td>
</tr>
<tr>
<td><strong>Add 2</strong></td>
<td>2339 MFlops (36 %)</td>
<td>1187 (24 %)</td>
</tr>
<tr>
<td></td>
<td>37.4 GByte/s</td>
<td>19.0 GByte/s</td>
</tr>
<tr>
<td><strong>Add 2 var</strong></td>
<td>2454 MFlops (38 %)</td>
<td>2082 MFlops (43.3 %)</td>
</tr>
<tr>
<td></td>
<td>39.2 GByte/s</td>
<td>33.3 GByte/s</td>
</tr>
</tbody>
</table>

movdqa xmm1, [x+ecx*8]
movdqa xmm3, [y+ecx*8]
addpd xmm3, xmm1

_replaced by_

movdqa xmm4, [x+ecx*8]
addpd xmm4, [y+ecx*8]
Peak Performance: The Code

Example (Peakflop Code snippet)

```
.loop:
movapd  xmm1, [x+ecx*8]
addpd  xmm6, xmm0
mulpd  xmm1, xmm7
movapd  xmm2, [x+ecx*8+16]
addpd  xmm5, xmm0
mulpd  xmm2, xmm7
movapd  xmm3, [x+ecx*8+32]
addpd  xmm1, xmm0
mulpd  xmm3, xmm7
movapd  xmm4, [x+ecx*8+48]
addpd  xmm2, xmm0
mulpd  xmm4, xmm7
add ecx, 8
cmp ecx, 1000
jb .loop
```
Intel vs. AMD

- Low latency (AMD) against high bandwidth (Intel)
- Intel is more sensitive against type of instructions
- AMD suffers from low bandwidth L2 Cache connection
- For streaming applications the Netburst Architecture is superior
- AMD has very good memory connection
- Hardware prefetcher works more efficiently on the P4
- Software prefetch instructions work more efficiently on the Athlon64
## Memcpy: Influence of instruction types

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Pentium 4 (MB/s)</th>
<th>Athlon64 (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CISC</td>
<td>2481</td>
<td>2001</td>
</tr>
<tr>
<td>RISC</td>
<td>2424</td>
<td>2834</td>
</tr>
<tr>
<td>MMX</td>
<td>2489</td>
<td>2880</td>
</tr>
<tr>
<td>MMX NT</td>
<td>3737</td>
<td>4104</td>
</tr>
<tr>
<td>MMX NT SW-Prefetch</td>
<td>3964</td>
<td>5199</td>
</tr>
<tr>
<td>SSE NT SW-Prefetch</td>
<td>4012</td>
<td>5206</td>
</tr>
<tr>
<td>SSE2 Block Prefetch</td>
<td>4644</td>
<td>6030</td>
</tr>
</tbody>
</table>
Stream Triad: In Memory

<table>
<thead>
<tr>
<th></th>
<th>Pentium 4</th>
<th>Athlon64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>4193 MB/s</td>
<td>4533 MB/s (3114 MB/s)</td>
</tr>
<tr>
<td>Optimized</td>
<td>4946 MB/s</td>
<td>5626 MB/s</td>
</tr>
</tbody>
</table>

The difference in performance is caused by effective prefetching and the separation of prefetching data into cache and doing the actual computations with storing it back.
Stream Triad: In Cache

Stream triad
in cache

- Intel Pentium 4
- AMD Athlon64
- AMD Athlon 64 opt

Dimension
MByte/s

- 0
- 10
- 100
- 1000
- 15000
- 20000
- 25000
- 30000
- 35000
Software Pipelined Loops

Itanium 2
Red-Black Gauss-Seidel in Cache

![Graph showing performance comparison]

- Black line: C Version dd
- Blue line: Fastest C Version
- Green line: ASM reuse register
- Red line: ASM melted pipelines

Length in x dimension
Mflops/s
Can you afford to waste a factor of 2-5?

Intel Pentium 4 Prescott
Red-Black Gauss-Seidel 2D

- Reference C
- Non Temporal Moves ASM
- Non Temporal Moves with Prefetching ASM
- 3 Iterations blocked ASM
- 3 Iterations blocked with Prefetching ASM
## Points for discussion

There is obviously a strong need for a tighter integration of ISA and Software. In addition to that the implementation of the ISA should be more transparent and reliable.

What can be done to solve that problem on hardware and software side?