Evaluating the Productivity of a Multicore Architecture

Jeremy Kepner and Nadya Bliss

MIT Lincoln Laboratory

HPEC 2008

This work is sponsored by the Department of Defense under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.
Outline

- Parallel Design
  - Programming Models
  - Architectures
  - Productivity Results
- Architecture Buffet
- Programming Buffet
- Productivity Assessment
- Summary
Signal Processor Devices

- Wide range of device technologies for signal processing systems
- Each has their own tradeoffs. *How do we choose?*
Multicore Processor Buffet

<table>
<thead>
<tr>
<th>Homogeneous</th>
<th>Heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Vector</td>
<td>Intel Duo/Duo</td>
</tr>
<tr>
<td></td>
<td>AMD Opteron</td>
</tr>
<tr>
<td></td>
<td>IBM PowerX</td>
</tr>
<tr>
<td></td>
<td>Sun Niagara</td>
</tr>
<tr>
<td></td>
<td>IBM Blue Gene</td>
</tr>
<tr>
<td>Long Vector</td>
<td>Cray XT</td>
</tr>
<tr>
<td></td>
<td>Cray XMT</td>
</tr>
<tr>
<td></td>
<td>Clearspeed</td>
</tr>
</tbody>
</table>

- Wide range of programmable multicore processors
- Each has their own tradeoffs. *How do we choose?*
### Multicore Programming Buffet

<table>
<thead>
<tr>
<th>Flat</th>
<th>Hierarchical</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>word</strong></td>
<td></td>
</tr>
<tr>
<td>pThreads</td>
<td>CUDA</td>
</tr>
<tr>
<td>StreamIt</td>
<td>ALPH</td>
</tr>
<tr>
<td>UPC</td>
<td>MCF</td>
</tr>
<tr>
<td>CAF</td>
<td>Sequouia</td>
</tr>
<tr>
<td>Cilk</td>
<td></td>
</tr>
</tbody>
</table>

| **object**    |                    |
| VSIPL++       | PVTOL              |
| GA++          | pMatlabXVM         |
| pMatlab       |                    |
| StarP         |                    |

- Wide range of multicore programming environments
- Each has their own tradeoffs. *How do we choose?*
Performance vs Effort

<table>
<thead>
<tr>
<th>Style</th>
<th>Example</th>
<th>Granularity</th>
<th>Training</th>
<th>Effort</th>
<th>Performance per Watt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphical</td>
<td>Spreadsheet</td>
<td>Module</td>
<td>Low</td>
<td>1/30</td>
<td>1/100</td>
</tr>
<tr>
<td>Domain Language</td>
<td>Matlab, Maple, IDL</td>
<td>Array</td>
<td>Low</td>
<td>1/10</td>
<td>1/5</td>
</tr>
<tr>
<td>Object Oriented</td>
<td>Java, C++</td>
<td>Object</td>
<td>Medium</td>
<td>1/3</td>
<td>1/1.1</td>
</tr>
<tr>
<td>Procedural Library</td>
<td>VSIPL, BLAS</td>
<td>Medium</td>
<td>Medium</td>
<td>2/3</td>
<td>1/1.05</td>
</tr>
<tr>
<td>Procedural Language</td>
<td>C, Fortran</td>
<td>High</td>
<td>Medium</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Assembly</td>
<td>x86, PowerPC</td>
<td>Register</td>
<td>High</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Gate Array</td>
<td>VHDL</td>
<td>Gate</td>
<td>High</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Standard Cell</td>
<td>Cell</td>
<td>Cell</td>
<td>High</td>
<td>30</td>
<td>100</td>
</tr>
<tr>
<td>Custom VLSI</td>
<td>Transistor</td>
<td>Transistor</td>
<td>High</td>
<td>100</td>
<td>1000</td>
</tr>
</tbody>
</table>

- Applications can be implemented with a variety of interfaces
- Clear tradeoff between effort (3000x) and performance (100,000x)
  - Translates into mission capability vs mission schedule

Programmable Multicore (this talk)
Assessment Approach

- “Write” benchmarks in many programming environments on different multicore architectures
- Compare performance/watt and relative effort to serial C

Graph:
- Relative Performance/Watt vs Relative Effort
- Goal
- Traditional Parallel Programming
- Java, Matlab, Python, etc.
- “All too often”

MIT Lincoln Laboratory
Outline

- Parallel Design
- **Programming Models**
- Architectures
- Productivity Results
- Summary

- Environment features
- Estimates
- Performance Complexity
Programming Environment Features

<table>
<thead>
<tr>
<th>Technology</th>
<th>UPC</th>
<th>F2008</th>
<th>GA++</th>
<th>PVL</th>
<th>VSIPL</th>
<th>PVTOL</th>
<th>Titanium</th>
<th>StarP</th>
<th>pMatlab</th>
<th>DCT</th>
<th>Chapel</th>
<th>X10</th>
<th>Fortress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organization</td>
<td>Std Body</td>
<td>Std Body</td>
<td>DOE PNNL</td>
<td>Std Body</td>
<td>Lincoln</td>
<td>Std Body</td>
<td>UC Berkeley</td>
<td>ISC</td>
<td>Lincoln</td>
<td>Math-works</td>
<td>Cray</td>
<td>IBM</td>
<td>Sun</td>
</tr>
<tr>
<td>Sponsor</td>
<td>DoD</td>
<td>DOE SC</td>
<td>DOE</td>
<td>DoD HPCMP</td>
<td>DOE, NSF</td>
<td>DoD</td>
<td>DARPA</td>
<td>DARPA</td>
<td>DARPA</td>
<td>DARPA</td>
<td>DARPA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Lang Ext</td>
<td>Lang Ext</td>
<td>Library</td>
<td>Library</td>
<td>Library</td>
<td>New Lang</td>
<td>Library</td>
<td>Library</td>
<td>Library</td>
<td>New Lang</td>
<td>New Lang</td>
<td>New Lang</td>
<td></td>
</tr>
<tr>
<td>Base Lang</td>
<td>C</td>
<td>Fortran</td>
<td>C++</td>
<td>C++</td>
<td>C++</td>
<td>Java</td>
<td>Matlab</td>
<td>Matlab</td>
<td>Matlab</td>
<td>ZPL</td>
<td>Java</td>
<td>HPF</td>
<td></td>
</tr>
<tr>
<td>Precursors</td>
<td>CAF</td>
<td>STAPL, POOMA</td>
<td>PVL, POOMA</td>
<td>VSIPL++, pMatlab</td>
<td>pMatlab</td>
<td>PVL, StarP</td>
<td>pMatlab, StarP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Parallel</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block-cyclic</td>
<td>1D</td>
<td>ND blk</td>
<td>2D</td>
<td>2D</td>
<td>Y</td>
<td>ND</td>
<td>2D</td>
<td>4D</td>
<td>1D</td>
<td>ND</td>
<td>ND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task Parallel</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipelines</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hier. arrays</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automap</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse</td>
<td>?</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPGA IO</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Too many environments with too many features to assess individually
- Decompose into general classes
  - Serial programming environment
  - Parallel programming model
- Assess only relevant serial environment and parallel model pairs
Dimensions of Programmability

• Performance
  – The performance of the code on the architecture
  – Measured in: flops/sec, Bytes/sec, GUPS, …

• Effort
  – Coding effort required to obtain a certain level of performance
  – Measured in: programmer-days, lines-of-code, function points,

• Expertise
  – Skill level of programmer required to obtain a certain level of performance
  – Measured in: degree, years of experience, multi-disciplinary knowledge required, …

• Portability
  – Coding effort required to port code from one architecture to the next and achieve a certain level of performance
  – Measured in: programmer-days, lines-of-code, function points, …

• Baseline
  – All quantities are relative to some baseline environment
  – Serial C on a single core x86 workstation, cluster, multi-core, …
## Serial Programming Environments

<table>
<thead>
<tr>
<th>Programming Language</th>
<th>Assembly (C+Altivec)</th>
<th>Procedural (ANSI C)</th>
<th>Objects (C++, Java)</th>
<th>High Level Languages (Matlab)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Efficiency</td>
<td>0.8</td>
<td>0.5</td>
<td>0.2</td>
<td>0.15</td>
</tr>
<tr>
<td>Relative Code Size</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>1/3</td>
</tr>
<tr>
<td>Effort/Line-of-Code</td>
<td>4 hour</td>
<td>2 hour</td>
<td>1 hour</td>
<td>20 min</td>
</tr>
<tr>
<td>Portability</td>
<td>Zero</td>
<td>Low</td>
<td>Very High</td>
<td>High</td>
</tr>
<tr>
<td>Granularity</td>
<td>Word</td>
<td>Multi-word</td>
<td>Multi-word</td>
<td>Object</td>
</tr>
</tbody>
</table>

- OO High Level Languages are the current desktop state-of-the-practice :-)
- Assembly/SIMD are the current multi-core state-of-the-practice :-(
- Single core programming environments span 10x performance and 100x relative code size
### Parallel Programming Environments

<table>
<thead>
<tr>
<th>Approach</th>
<th>Direct Memory Access (DMA)</th>
<th>Message Passing (MPI)</th>
<th>Threads (OpenMP)</th>
<th>Recursive Threads (Cilk)</th>
<th>PGAS (UPC, VSIPL++)</th>
<th>Hierarchical PGAS (PVTOL, HPCS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Efficiency</td>
<td>0.8</td>
<td>0.5</td>
<td>0.2</td>
<td>0.4</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Relative Code Size</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>1/3</td>
<td>1/10</td>
<td>1/10</td>
</tr>
<tr>
<td>Effort/Line-of-Code</td>
<td>Very High</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Portability</td>
<td>Zero</td>
<td>Very High</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
<td>TBD</td>
</tr>
<tr>
<td>Granularity</td>
<td>Word</td>
<td>Multi-word</td>
<td>Word</td>
<td>Array</td>
<td>Array</td>
<td>Array</td>
</tr>
</tbody>
</table>

- Message passing and threads are the current desktop state-of-the-practice :-|
- DMA is the current multi-core state-of-the-practice :-(
- Parallel programming environments span 4x performance and 100x relative code size
Canonical 100 CPU Cluster Estimates

- Programming environments form regions around serial environment
### Relevant Serial Environments and Parallel Models

**Partitioning Scheme**
- Serial
- Multi-Threaded
- Distributed Arrays
- Hierarchical Arrays
- Assembly + DMA

<table>
<thead>
<tr>
<th></th>
<th>Serial</th>
<th>Multi-Threaded</th>
<th>Distributed Arrays</th>
<th>Hierarchical Arrays</th>
<th>Assembly + DMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraction of Programmers</td>
<td>1</td>
<td>0.95</td>
<td>0.50</td>
<td>0.10</td>
<td>0.05</td>
</tr>
<tr>
<td>Relative Code Size</td>
<td>1</td>
<td>1.1</td>
<td>1.5</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>“Difficulty”</td>
<td>1</td>
<td>1.15</td>
<td>3</td>
<td>20</td>
<td>200</td>
</tr>
</tbody>
</table>

**Focus on a subset of relevant programming environments**
- C/C++ + serial, threads, distributed arrays, hierarchical arrays
- Assembly + DMA

**“Difficulty”** = (relative code size) / (fraction of programmers)
Performance complexity (Strohmeier/LBNL) compares performance as a function of the programming model.

- In the above graph, point “G” is ~100x easier to program than point “B”.
Outline

- Parallel Design
- Programming Models
- Architectures
- Productivity Results
- Summary

- Kuck Diagram
- Homogeneous UMA
- Heterogeneous NUMA
- Benchmarks
Single Processor Kuck Diagram

- Processors denoted by boxes
- Memory denoted by ovals
- Lines connected associated processors and memories
- Subscript denotes level in the memory hierarchy
Parallel Kuck Diagram

- Replicates serial processors
- **net** denotes network connecting memories at a level in the hierarchy (incremented by 0.5)
Multicore Architecture 1: Homogeneous

- Off-chip: 1 (all cores have UMA access to off-chip memory)
- On-chip: 1 (all cores have UMA access to on-chip 3D memory)
- Core: $N_{\text{core}}$ (each core has its own cache)
Multicore Architecture 2: Heterogeneous

- Off-chip: 1 (all supercores have UMA access to off-chip memory)
- On-chip: N (sub-cores share a bank of on-chip 3D memory and 1 control processor)
- Core: \( N_{\text{core}} \) (each core has its own local store)
HPC Challenge SAR benchmark (2D FFT)

- 2D FFT (with a full all-to-all corner turn) is a common operation in SAR and other signal processing.
- Operation is complex enough to highlight programmability issues.

\[ \text{MATLAB Code} \]

\[
A = \text{complex}(\text{rand}(N,M), \ldots \ldots \text{rand}(N,M));
\]

\[
\text{FFT along columns}
B = \text{fft}(A, [], 1);
\]

\[
\text{FFT along rows}
C = \text{fft}(B, [], 2);
\]
Projective Transform

- Canonical kernel in image processing applications
- Takes advantage of cache on single core processors
- Takes advantage of multiple cores
- Results in regular distributions of both source and destination images
Outline

- Parallel Design
- Programming Models
- Architectures
- Productivity Results

- Implementations
- Performance vs Effort
- Productivity vs Model

- Summary
Case 1: Serial Implementation

**Heterogeneous Performance**

**NOTES**

- Single threaded program
- Complexity: LOW
- Initial implementation to get the code running on a system
- No parallel programming expertise required
- Users capable of writing this program: 100%

**HOMOGENEOUS PERFORMANCE**

**Execution**
- This program will run on a single core

**Memory**
- Off chip, on chip cache, and local cache will be used

**Execution**
- This program will run on a single control processor

**Memory**
- Only off chip memory will be used

```matlab
A = complex(rand(N,M), rand(N,M));

//FFT along columns
for j=1:M
    A(:,j) = fft(A(:,j));
end

//FFT along rows
for i=1:N
    A(i,:) = fft(A(i,:));
end
```
Case 2: Multi-Threaded Implementation

**CODE**

```matlab
A = complex(rand(N,M), rand(N,M));
#pragma omp parallel ...
//FFT along columns
for j=1:M
  A(:,j) = fft(A(:,j));
end
#pragma omp parallel ...
//FFT along rows
for i=1:N
  A(i,:) = fft(A(i,:));
end
```

**NOTES**

- Multi-threaded program: each thread operated on a single column (row) of the matrix
- Complexity: LOW
- Minimal parallel programming expertise required
- Users capable of writing this program: 99%

**Execution**

- This program will run on all control processors

**Memory**

- Only off chip memory will be used
- Poor locality will cause cause a memory bottleneck
Case 3: Parallel 1D Block Implementation

CODE

```matlab
mapA = map([1 36], {}, [0:35]); //column map
mapB = map([36 1], {}, [0:35]); //row map
A = complex(rand(N,M,mapA), rand(N,M,mapA));
B = complex(zeros(N,M,mapB), rand(N,M,mapB));
//Get local indices
myJ = get_local_ind(A);
myI = get_local_ind(B);
//FFT along columns
for j=1:length(myJ)
    A.local(:,j) = fft(A.local(:,j));
end
B(:,:,1) = A; //corner turn
//FFT along rows
for i=1:length(myI)
    B.local(i,:) = fft(B.local(i,:));
end
```

NOTES

- Explicitly parallel program using 1D block distribution
- Complexity: MEDIUM
- Parallel programming expertise required, particularly for understanding data distribution
- Users capable of writing this program: 75%

Execution
- This program will run on all control processors
- Only off chip memory will be used

Memory
- Off chip memory, on chip cache, and local cache will be used
- Better locality will decrease memory bottleneck
mapHCol = map([1 8], {}, [0:7]); // col hierarchical map
mapHRow = map([8 1], {}, [0:7]); // row hierarchical map
mapH = map([0:7]); // base hierarchical map
mapA = map([1 36], {}, [0:35], mapH); // column map
mapB = map([36 1], {}, [0:35], mapH); // row map
A = complex(rand(N,M,mapA), rand(N,M,mapA));
B = complex(zeros(N,M,mapB), rand(N,M,mapB));

// Get local indices
myJ = get_local_ind(A);
myI = get_local_ind(B);

// FFT along columns
for j=1:length(myJ)
    temp = A.local(:,j); // get local col
    temp = reshape(temp); // reshape col into matrix
    alocal = zeros(size(temp_col), mapHcol);
    blocal = zeros(size(temp_col), mapHrow);
    alocal(:,:) = temp; // distribute col to fit into SPE/cache
    myHj = get_local_ind(alocal);
    for jj = length(myHj)
        alocal.local(:,jj) = fft(alocal.local(:,jj));
    end
    blocal(:,:) = alocal; // corner turn that fits into SPE/cache
    myHi = get_local_ind(blocal);
    for ii = length(myHi)
        blocal.local(ii,:) = fft(blocal.local(ii,:));
    end
    temp = reshape(blocal); // reshape matrix into column
    A.local = temp; // store result
end
B(:,:) = A; // corner turn

// FFT along rows ...

Case 4: Parallel 1D Block Hierarchical Implementation

- Complexity: HIGH
- Users capable of writing this program: <20%

Execution
- This program will run on all cores

Memory
- Off chip, on-chip, and local store memory will be used
- Hierarchical arrays allow detailed management of memory bandwidth

Heterogeneous Performance

Homogeneous Performance

Execution
- This program will run on all cores

Memory
- Off chip, on-chip cache, and local cache will be used
- Caches prevent detailed management of memory bandwidth
Trade offs exist between performance and programming difficulty.

Different architectures enable different performance and programming capabilities.

Forces system architects to understand device implications and consider programmability.

Programming Models:
- A: C single threaded
- B: C multi-threaded
- C: Parallel Arrays
- D: Hierarchical Arrays
- E: Hand Coded Assembly

Programming Difficulty = \( \frac{(\text{Code Size})}{(\text{Fraction of Programmers})} \)
Defining Productivity

- Productivity is a ratio between utility to cost
- From the programmer perspective this is proportional to performance over difficulty
Productivity vs Programming Model

- Productivity varies with architecture and application
- Homogeneous: threads or parallel arrays
- Heterogeneous: hierarchical arrays

Programming Models:
- A C single threaded
- B C multi-threaded
- C Parallel Arrays
- D Hierarchical Arrays
- E Hand Coded Assembly
Summary

• Many multicore processors are available

• Many multicore programming environments are available

• Assessing which approaches are best for which architectures is difficult

• Our approach
  – “Write” benchmarks in many programming environments on different multicore architectures
  – Compare performance/watt and relative effort to serial C

• Conclusions
  – For homogeneous architectures C/C++ using threads or parallel arrays has highest productivity
  – For heterogeneous architectures C/C++ using hierarchical arrays has highest productivity