The Future of Many Core Computing:
A tale of two processors

Tim Mattson
Intel Labs
January 2010
Disclosure

• The views expressed in this talk are those of the speaker and not his employer.
• I am in a research group and know nothing about Intel products. So anything I say about them is highly suspect.
• This was a team effort, but if I say anything really stupid, it’s all my fault ... don’t blame my collaborators.
A common view of many-core chips

What the Cores will look like:
From a few large cores to many lightweight cores

Optimized for speed
Optimized for performance/watt

An Intel Exec’s slide from IDF’2006

Pentium® processor era chips optimized for raw speed on single threads. Pipelined, out of order execution.

Today’s chips use cores which balance single threaded and multi-threaded performance.

5-10 years: 10s-100s of energy efficient, IA cores optimized for multithreading.
Challenging the sacred cows

Assumes cache coherent shared address space!

• Is that the right choice?
  – Most expert programmers do not fully understand relaxed consistency memory models required to make cache coherent architectures work.
  – The only programming models proven to scale non-trivial apps to 100’s to 1000’s of cores all based on distributed memory.
  – Coherence incurs additional architectural overhead

...IA cores optimized for multithreading
The Coherency Wall

- As you scale the number of cores on a cache coherent system (CC), “cost” in “time and memory” grows to a point beyond which the additional cores are not useful in a single parallel program. This is the coherency wall.

\[ \text{Cost (time and/or memory)} = O(N^\alpha) \quad 1 \leq \alpha \leq 2 \]

For a scalable, directory based scheme, CC incurs an N-body effect … cost scales at best linearly (Fixed memory size as cores are added) and at worst quadratically (memory grows linearly with number of cores).

HW Dist. Mem. HW cost scales at best a fixed cost for the local neighborhood and at worst as the diameter of the network.

... each directory entry will be 128 bytes long for a 1024 core processor supporting fully-mapped directory-based cache coherence. This may often be larger than the size of the cacheline that a directory entry is expected to track.*

* R. Kumar, T.G. Mattson, G. Pokam, R. van der Wijngaart, “The case for message passing on many-core chips, submitted to HotPar 2010
Isn’t shared memory programming easier? Not necessarily.

Extra work upfront, but easier optimization and debugging means overall, less time to solution

Message passing

Effort

Time

initial parallelization can be quite easy

But difficult debugging and optimization means overall project takes longer

Multi-threading

Effort

Time

Proving that a shared address space program using semaphores is race free is an NP-complete problem*

The many core design challenge

- **Scalable architecture:**
  - How should we connect the cores so we can scale as far as we need (O(100’s to 1000) should be enough)?

- **Software:**
  - Can “general purpose programmers” write software that takes advantage of the cores?
  - Will ISV’s actually write scalable software?

- **Manufacturability:**
  - Validation costs grow steeply as the number of transistors grows. Can we use tiled architectures to address this problem?
    - Validate a tile (M transistors) and the connections between tiles ...
    - Drops validation costs from K*O(N) to K’*O(M) (warning, K, K’ can be very large).

Intel’s “TeraScale” processor research program is addressing these questions with a series of Test chips ... two so far.

80 core Research processor

48 core SCC processor
Agenda

• The 80 core Research Processor
  – Max FLOPS/Watt in a tiled architecture

• The 48 core SCC processor
  – Scalable IA cores for software/platform research
Agenda

• The 80 core Research Processor
  – Max FLOPS/Watt in a tiled architecture

• The 48 core SCC processor
  – Scalable IA cores for software/platform research
Acknowledgements

• The software team
  – Tim Mattson, Rob van der Wijngaart (Intel)
  – Michael Frumkin (then at Intel, now at Google)

• Implementation
  – Circuit Research Lab Advanced Prototyping team (Hillsboro, OR and Bangalore, India)

• PLL design
  – Logic Technology Development (Hillsboro, OR)

• Package design
  – Assembly Technology Development (Chandler, AZ)

A special thanks to our “optimizing compiler” ... Yatin Hoskote, Jason Howard, and Saurabh Dighe of Intel’s Microprocessor Technology Laboratory.
• Basic statistics:
  - 65 nm CMOS process
  - 100 Million transistors in 275 mm²
  - 8x10 tiles, 3mm²/tile
  - Mesosynchronous clock
  - 1.6 SP TFLOP @ 5 Ghz and 1.2 V
  - 320 GB/s bisection bandwidth
  - Variable voltage and multiple sleep states for explicit power management
We’ve made good progress with the hardware: Intel’s 80 core test chip (2006)

- **2KB DATA MEMORY**
- **3KB INSTR. MEMORY**
- **5 PORT ROUTER**
- **COMPUTE CORE: 2 FLOATING POINT ENGINES**
The “80-core” tile

- 2KB Data memory (DMEM)
- 3KB Inst. Memory (256 96 bit instr)
- 6-read, 4-write 32 entry RF
- 2 single precision FPMAC units
- 5 port router for a 2D mesh and 3D stacking
- 3 Kbyte Instr. Memory (512 SP words)
- 40 GB/s
- 2 Kbyte Data Memory (512 SP words)
- Processing Engine (PE)
Programmer’s perspective

- **8x10 mesh of 80 cores**
- **All memory on-chip**
  - 256 instructions operating
  - 512 floating point numbers.
  - 32 SP registers, two loads per cycle per tile
- **Compute engine**
  - 2 SP FMAC units per tile → 4 FLOP/cycle/tile
  - 9-stage pipeline
- **Communication**
  - One sided anonymous message passing into instruction or data memory
- **Limitations:**
  - No division
  - No general branch, single branch-on-zero (single loop)
  - No wimps allowed! ... i.e. No compiler, Debugger, OS, I/O ...

---

14 SP = single precision, FMAC = floating point multiply accumulate, FLOP = floating point operations
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULT</td>
<td>Multiply operands</td>
</tr>
<tr>
<td>ACCUM</td>
<td>Accumulate with previous result</td>
</tr>
<tr>
<td>LOAD, STORE</td>
<td>Move a pair of floats between register file &amp; data memory.</td>
</tr>
<tr>
<td>LOADO, STOREO, OFFSET</td>
<td>Move a pair of floats between the register file and data memory at address plus OFFSET.</td>
</tr>
<tr>
<td>SENDI[H</td>
<td>A</td>
</tr>
<tr>
<td>SENDD[H</td>
<td>A</td>
</tr>
<tr>
<td>WFD</td>
<td>Stall while waiting for data from any tile.</td>
</tr>
<tr>
<td>STALL</td>
<td>Stall program counter (PC), waiting for a new PC.</td>
</tr>
<tr>
<td>BRNE, INDEX</td>
<td>INDEX sets a register for loop count. BRNE branches while the index register is greater than zero</td>
</tr>
<tr>
<td>JUMP</td>
<td>Jump to the specified program counter address</td>
</tr>
<tr>
<td>NAP</td>
<td>Put FPUs to sleep</td>
</tr>
<tr>
<td>WAKE</td>
<td>Wake FPUs from sleep</td>
</tr>
</tbody>
</table>
### Instruction word and latencies

- 96-bit instruction word, up to 8 operations/cycle

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPU</td>
<td>9</td>
</tr>
<tr>
<td>LOAD/STORE</td>
<td>2</td>
</tr>
<tr>
<td>SEND/RECEIVE</td>
<td>2</td>
</tr>
<tr>
<td>JUMP/BRANCH</td>
<td>1</td>
</tr>
<tr>
<td>NAP/WAKE</td>
<td>1</td>
</tr>
</tbody>
</table>
What did we do with the chip?

- 4 applications kernels
  - **Stencil**
    - 2D PDE solver (heat diffusion equation) using a Gauss Seidel algorithm
  - **SGEMM (Matrix Multiply)**
    - $C = A \times B$ with rectangular matrices
  - **Spreadsheet**
    - Synthetic benchmark... sum dense array of rows and columns (local sums in one D, reduction in the other D)
  - **2D FFT**
    - 2D FFT of dense array on an 8 by 8 subgrid.

These kernels were hand coded in assembly code and manually optimized. Data sets sized to fill data memory.
Programming Results

Theoretical numbers from operation/communication counts and from rate limiting bandwidths.

1.07V, 4.27GHz operation 80 C
Why this is so exciting!

First TeraScale* computer: 1997

Intel’s ASCI Red Supercomputer

9000 CPUs

one megawatt of electricity.

1600 square feet of floor space.

*Double Precision TFLOPS running MP-Linpack

First TeraScale% chip: 2007

10 years later

Intel’s 80 core teraScale Chip

1 CPU

97 watt

275 mm2

%Single Precision TFLOPS running stencil

Source: Intel
Lessons: Part 1

• What should we do with our huge transistor counts
  – A fraction of the transistor budget should be used for on-die memory.
  – The 80-core Terascale Processor with its on-die memory has a 2 cycle latency for load/store operations ... this compares to ~100 nsec access to DRAM.
  – As core counts increase, the need for on-chip memory will grow!
  – For Power/Performance, specialized cores rule!

• What role should Caches play?
  – This NoC design lacked caches.
  – Cache coherence limits scalability:
    – Coherence traffic may collide with useful communication.
    – Increases overhead ... Due to Amdahl’s law, A chip with on the order of 100 cores would be severely impacted by even a small overhead ~1%
Lessons: Part 2

• Minimize message passing overhead.
  – Routers wrote directly into memory without interrupting computing ... i.e. any core could write directly into the memory of any other core. This led to extremely small comm. latency on the order of 2 cycles.

• Programmers can assist in keeping power low if sleep/wake instructions are exposed and if switching latency is low (~ a couple cycles).

• Application programmers should help design chips
  – This chip was presented to us a completed package.
  – Small changes to the instruction set could have had a large impact on the programmability of the chip.
    – A simple computed jump statement would have allowed us to add nested loops.
    – A second offset parameter would have allowed us to program general 2D array computations.
Agenda

• The 80 core Research Processor
  – Max FLOPS/Watt in a tiled architecture

• The 48 core SCC processor
  – Scalable IA cores for software/platform research
Acknowledgements

• SCC Application software:
  RCCE library and apps and HW/SW co-design
  Developer tools (icc and MKL)

  Rob Van der Wijngaart
  Tim Mattson
  Patrick Kennedy

• SCC System software:
  Management Console software
  BareMetalC workflow
  Linux for SCC
  System Interface FPGA development
  TCP/IP network driver

  Michael Riepen
  Michael Riepen
  Thomas Lehnig
  Matthias Steidl
  Werner Haas

• And the HW-team that worked closely with the SW group:
  Jason Howard, Yatin Hoskote, Sriram Vangal, Nitin Borkar, Greg Ruhl
SCC full chip

- 24 tiles in 6x4 mesh with 2 cores per tile (48 cores total).

<table>
<thead>
<tr>
<th>Technology</th>
<th>45nm Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interconnect</td>
<td>1 Poly, 9 Metal (Cu)</td>
</tr>
<tr>
<td>Transistors</td>
<td>Die: 1.3B, Tile: 48M</td>
</tr>
<tr>
<td>Tile Area</td>
<td>18.7mm²</td>
</tr>
<tr>
<td>Die Area</td>
<td>567.1mm²</td>
</tr>
</tbody>
</table>
SCC Dual-core Tile

- 2 P54C cores (16K L1$/core)
- 256K L2$ per core
- 8K Message passing buffer
- Clock Crossing FIFOs b/w Mesh interface unit and Router

- Tile area 18.7mm²
- Core are 3.9mm²
- Cores and uncore units @1GHz
- Router @2GHz
Hardware view of SCC

- 48 P54C cores in 6x4 mesh with 2 cores per tile
- 45 nm, 1.3 B transistors, 25 to 125 W
- 16 to 64 GB total main memory using 4 DDR3 MCs

R = router, MC = Memory Controller, P54C = second generation Pentium core, CC = cache cntrl.
Router Architecture

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>2GHz @ 1.1V</td>
</tr>
<tr>
<td>Latency</td>
<td>4 cycles</td>
</tr>
<tr>
<td>Link Width</td>
<td>16 Bytes</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>64GB/s per link</td>
</tr>
<tr>
<td>Architecture</td>
<td>8 VCs over 2 MCs</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>500mW @ 50°C</td>
</tr>
</tbody>
</table>
On-Die 2D Mesh

- 16B wide data links + 2B sideband
  - Target frequency: 2GHz
  - Bisection bandwidth: 1.5Tb/s to 2Tb/s, avg. power 6W to 12W
  - Latency: 4 cycles (2ns)
- 2 message classes and 8 VCs
- Low power circuit techniques
  - Sleep, clock gating, voltage control, low power RF
  - Low power 5 port crossbar design
- Speculative VC allocation
- Route pre-computation
- Single cycle switch allocation
Core Memory Management

- Each core has an address look up Table (LUT) extension
  - Provides address translation and routing information.

- Table manages Memory space as 16MB pages marked as private or shared
  - Shared space seen by all cores ... but NO Cache coherency
  - Private memory ... coherent with a cores L1 and L2 cache (P54C memory model).

- User is responsible for setting up pages to fit within the core and memory controller constraints

- LUT boundaries are dynamically programmed

256MB
- Maps to LUT
- Maps to VRCs
- Maps to MC3
- Maps to MC1
- Maps to MC2
- Maps to MPBs

512MB
- Private

FPGA registers
- APIC/boot
- PCI hierarchy

MC# = one of the 4 memory controllers, MPB = message passing buffer, VRC’s = Voltage Regulator control
Package and Test Board

<table>
<thead>
<tr>
<th>Technology</th>
<th>45nm Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>1567 pin LGA package</td>
</tr>
<tr>
<td></td>
<td>14 layers (5-4-5)</td>
</tr>
<tr>
<td>Signals</td>
<td>970 pins</td>
</tr>
</tbody>
</table>
SCC system overview
Core & Router Fmax

![Graph showing the relationship between Vcc (V) and Freq (GHz) for Core and Router at 50°C. The graph includes points for 0.55V, 0.73V, and 0.94V, with corresponding frequencies of 60MHz, 300MHz, 1.4GHz, and 1.3GHz.]
Measured full chip power

50°C

Active Power

Leakage Power

125W @ 1GHz core, 2GHz mesh

Vcc (V)

Power (W)

25W

51W

110W

201W

0.7 0.8 0.91 1 1.05 1.1 1.14 1.21 1.28
Power breakdown

**Full Power Breakdown**
Total -125.3W

- MC & DDR3-800 19%
- Routers & 2D-mesh 10%
- Cores 69%
- Global Clocking 2%

Clocking: 1.9W  
Cores: 87.7W  
MCs: 23.6W

**Low Power Breakdown**
Total - 24.7W

- MC & DDR3-800 69%
- Routers & 2D-mesh 5%
- Cores 21%
- Global Clocking 5%

Clocking: 1.2W  
Cores: 5.1W  
MCs: 17.2W

Cores-1GHz, Mesh-2GHz, 1.14V, 50°C  
Cores-125MHz, Mesh-250MHz, 0.7V, 50°C
Programmer’s view of SCC

- 48 x86 cores with the familiar x86 memory model for Private DRAM
- 3 memory spaces, with fast message passing between cores

(on/off-chip)

Shared off-chip DRAM (variable size)

Shared on-chip Message Passing Buffer (8KB/core)

Shared test and set register
SCC Software research goals

• Understand programmability and application scalability of many-core chips.

• Answer question “what can you do with a many-core chip that has (some) shared non-cache-coherent memory?”

• Study usage models and techniques for software controlled power management

• Sample software for other programming model and applications researchers (industry partners, Flame group at UT Austin, UPCRC, YOU ... i.e. the MARC program)

Our research resulted in a light weight, compact, low latency communication library called RCCE (pronounced “Rocky”)
**SCC Platforms**

- Three platforms for SCC and RCCE
  - Functional emulator (on top of OpenMP)
  - SCC board with two “OS Flavors” ... Linux or Baremetal (i.e. no OS)

Functional emulator, based on OpenMP.

RCCE supports greatest common denominator between the three platforms

Third party names are the property of their owners.
High level view of RCCE

• RCCE is a compact, lightweight communication environment.
  – SCC and RCCE were designed together side by side:
    – ... a true HW/SW co-design project.

• RCCE is a research vehicle to understand how message passing APIs map onto many core chips.

• RCCE is for experienced parallel programmers willing to work close to the hardware.

• RCCE Execution Model:
  – Static SPMD:
    – identical UEs created together when a program starts (this is a standard approach familiar to message passing programmers)

UE: Unit of Execution ... a software entity that advances a program counter (e.g. process of thread).
How does RCCE work? Part 1

Consequences of MPBT properties:

- If data changed by another core and image still in L1, read returns stale data.
  - **Solution:** Invalidate before read.
- L1 has write-combining buffer; write incomplete line? expect trouble!
  - **Solution:** don’t. Always push whole cache lines
- If image of line to be written already in L1, write will not go to memory.
  - **Solution:** invalidate before write.

Message passing buffer memory is special … of type MPBT

Cached in L1, L2 bypassed. Not coherent between cores

Data cached on read, not write. Single cycle op to invalidate all MPBT in L1

... Note this is not a flush

Discourage user operations on data in MPB. Use only as a data movement area managed by RCCE ... Invalidate early, invalidate often
How does RCCE work? Part 2

- Treat Msg Pass Buf (MPB) as 48 smaller buffers ... one per core.

- Symmetric name space ... Allocate memory as a collective op. Each core gets a variable with the given name at a fixed offset from the beginning of a core’s MPB.

A = (double *) RCCE_malloc(size)
Called on all cores so any core can put/get(A at Core_ID) without error-prone explicit offsets
How does RCCE work? Part 3

- The foundation of RCCE is a one-sided put/get interface.

- Symmetric name space ... Allocate memory as a collective and put a variable with a given name into each core’s MPB.

... and use flags to make the put’s and get’s “safe”
The RCCE library

- RCCE API provides the basic message passing functionality expected in a tiny communication library:
  - One + two sided interface (put/get + send/recv) with synchronization flags and MPB management exposed.
    - The “gory” interface for programmers who need the most detailed control over SCC
  - Two sided interface (send/recv) with most detail (flags and MPB management) hidden.
    - The “basic” interface for typical application programmers.
Linpack and NAS Parallel benchmarks

1. Linpack (HPL): solve dense system of linear equations
   - Synchronous comm. with “MPI wrappers” to simplify porting

2. BT: Multipartition decomposition
   - Each core owns multiple blocks (3 in this case)
   - update all blocks in plane of 3x3 blocks
   - send data to neighbor blocks in next plane
   - update next plane of 3x3 blocks

3. LU: Pencil decomposition
   - Define 2D-pipeline process.
     - await data (bottom+left)
     - compute new tile
     - send data (top+right)

Third party names are the property of their owners.
RCCE functional emulator vs. MPI
HPL implementation of the LINPACK benchmark

Low overhead synchronous message passing pays off even in emulator mode (compared to MPI)

These results provide a comparison of RCCE and MPI on an older 4 processor Intel® Xeon® MP SMP platform* with tiny 4x4 block sizes. These are not official MP-LINPACK results.

*3 GHz Intel® Xeon® MP processor in a 4 socket SMP platform (4 cores total), L2=1MB, L3=8MB, Intel® icc 10.1 compiler, Intel® MPI 2.0

Third party names are the property of their owners.
Linpack, on the Linux SCC platform

- Linpack (HPL)* strong scaling results:
  - GFLOPS vs. # of cores for a fixed size problem (1000).
  - This is a tough test ... scaling is easier for large problems.

### Calculation Details:
- Un-optimized C-BLAS
- Un-optimized block size (4x4)
- Used latency-optimized whole cache line flags
- Performance dropped ~10% with memory optimized 1-bit flags

Matrix order 1000

SCC processor 500MHz core, 1GHz routers, 25MHz system interface, and DDR3 memory at 800 MHz.

* These are not official LINPACK benchmark results.

Third party names are the property of their owners.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference [http://www.intel.com/performance](http://www.intel.com/performance) or call (U.S.) 1-800-628-8686 or 1-916-356-3104.
LU/BT NAS Parallel Benchmarks, SCC

Problem size: Class A, 64 x 64 x 64 grid*

SCC processor 500MHz core, 1GHz routers, 25MHz system interface, and DDR3 memory at 800 MHz.

- Using latency optimized, whole cache line flags

* These are not official NAS Parallel benchmark results.

Third party names are the property of their owners.
Power and memory-controller domains

Power \sim F V^2

- Power Control domains (RPC):
  - 7 voltage domains ... 6 4-tile blocks and one for on-die network.
  - 1 clock divider register per tile (i.e. 24 frequency domains)
  - One RPC register so can process only one voltage request at a time; other requestors block
RCCE Power Management API

• RCCE power management emphasizes safe control: V/GHz changed together within each 4-tile (8-core) power domain.
  – A Master core sets V + GHz for all cores in domain.
    – RCCE_istep_power():
      – steps up or down V + GHz, where GHz is max for selected voltage.
    – RCCE_wait_power():
      – returns when power change is done
    – RCCE_step_frequency():
      – steps up or down only GHz

• Power management latencies
  – V changes: Very high latency, \(O(\text{Million})\) cycles.
  – GHz changes: Low latency, \(O(\text{few})\) cycles.
Power management test

- A three-tier master-worker hierarchy,
  - one overall master, one team-lead per power domain, Team-members (cores) to do the work.
- Workload: A stencil computation to solve a PDE.

Dependent, synchronized subtasks; exchange interface data each iteration

Overall data space

Independent tasks (all different sizes)

Team member  xch  Team member  xch  Team member  xch  Team lead
SCC Power Management

Advanced Workload Aware Power Management Technology

Fine-grained dynamic frequency and voltage Control

Power Management

OFF   ON

121 Watts
SCC Power Management

Advanced Workload Aware Power Management Technology
Fine-grained dynamic frequency and voltage control

Power Management
OFF  ON

38% reduction

75 watts
SCC Demo Showcase

Financial Analytics
w/ shared virtual memory

Microsoft Visual Studio

Advanced Power Management

JavaScript Physics Modeling

HPC Parallel Workloads

Hadoop Web Search
Conclusions

• RCCE software works
  – RCCE’s restrictions (Symmetric MPB memory model and blocking communications) have not been a fundamental obstacle
  – Functional emulator is a useful development/debug device

• SCC architecture
  – The on-chip MPB was effective for scalable message passing applications
  – Software controlled power management works ... but it’s challenging to use because (1) granularity of 8 cores and (2) high latencies for voltage changes
  – The Test&set registers (only one per core) will be a bottleneck.
    – Sure wish we had asked for more!

• Future work
  – Add shmalloc() to expose shared off-chip DRAMM (in progress).
  – Move resource management into OS/drivers so multiple apps can work together safely.
  – We have only just begun to explore power management capabilities ... we need to explore additional usage models.
Backup Slides

• Details on 80 core processor application kernels
• More on RCCE
**Stencil**

- Five point stencil for Gauss Seidel relaxation to solve a heat diffusion equation with Dirichlet/periodic boundary conditions.
- Flattened 2D array dimensions and unrolled fused inner and outer loops to meet the single-loop constraint.
- Periodic Boundary conditions relaxed so updates at iteration \( q \) might use values from iteration \( q-1 \) off by one mesh width. This reduces method to \( O(h) \) ... answer’s correct but convergence slows.

- Parallelization:
  - Solve over a long narrow strip. Copy fringes between cores so fringes are contiguous (1D communication loop) if split vertically.

![Stencil over NxM grid](image)

- \( M=2240, N=16 \)
SGEMM

• Only one level of loops so we used a dot product algorithm ... unrolled loop for dot product
• Stored A and C by rows and B by column in diagonal wrapped order

On core number i
Loop over \( j = 1, M \)
\[
\{ \\
\quad C_{ij} = \text{dot\_product (row } A_i \ast \text{ column } B_j) \\
\quad \text{Circular shift column } B_j \text{ to neighbor} \\
\}
\]

• Treat cores as a ring and circular shift columns of B around the ring.
• After they complete once cycle through the full ring, the computation is done

\[
C(N,N) = A(N,M) \ast B(M,N) \\
N = 80, \ M = 206
\]
Spreadsheet

- Consider a table of data $v$ and weights $w$, stored by columns.
- Compute weighted row and column sums (dot products):
  - Column sum: $v_i = \sum_k v_{i,k}w_{i,k} = \sum_k v_{i+kN}w_{i+kN}$
  - Row sum: $v_k = \sum_i v_{i,k}w_{i,k} = \sum_i v_{i+kN}w_{i+kN}$
- Data size on each tile small enough to unroll loop over rows.

- Column sums local to a tile.
- Row sums required a vector reduction across all rows.
- We processed many spreadsheets at once so we could pipeline reductions to manage latencies.
- 76 cores did local csum and passed results to one of four accumulator nodes.
- The four nodes combined results to get final answer.
2D FFT

- 64 Point 2D FFT on an 8 by 8 Grid.
- Pease Algorithm
  - “Peers” in each phase are constant … a constant communication pattern throughout the computation.
- Parallelization:
  - Basic operation FFT of 64 long vector along a column of 8 tiles
    - FFT of 8-long vector in each tile
    - Communication:
      - Each cell communicates with each cell in the column.
      - When the column computations are done, each cell communicates with each cell in the row.
  - Unrolled inner loops … this filled instruction memory and limited overall problem size
Power Performance Results

Peak Performance

80°C

- (0.32 TFLOP) 1GHz
- (1 TFLOP) 3.16GHz
- (1.81 TFLOP) 5.67GHz
- (1.63 TFLOP) 5.1GHz

N=80

Average Power Efficiency

N=80
80°C

394 GFLOPS
10.5
5.8

Stencil: 1TFLOP @ 97W, 1.07V;

All tiles awake/asleep
Backup Slides

- Details on 80 core processor application kernels.
- More on RCCE
Rapidly Communicating Cores Env.
Reduced Compact Communication Environment
Research Cores Communication Environment
Rabble-of Communicating Cores Experiments
Rock Creek Communication Environment
Rationally Cool Coordination E-science
Richly Communicating Cores Ecosystem
Restricted Capability Communication Environment
Rorschach Core Communication Express

A small library for many-core communication

Rob van der Wijngaart (Software and Services Group)
Tim Mattson (Intel Labs)
RCCE: Supporting Details

• Using RCCE and example RCCE code
• Additional RCCE implementation details
• RCCE and the MPI programmer
RCCE API: Writing and running RCCE programs

• We provide two interfaces for the RCCE programmer:
  • **Basic Interface** (general purpose programmers):
    • FLAGS and Message Passing Buffer memory management hidden from the programmer.
  • **Gory interface** (hard core performance programmers):
    • One sided and two sided
    • Message Passing Buffer management is explicit
    • Flags allocated and managed by programmer.
• Build you job linking to the appropriate RCCE library, then run with rccerun

**rccerun -nue N [optional params] program[params]**

- **program** executes on N UEAs as if it were invoked as:
  “program params” (no parameters allowed for Baremetal)
- Optional parameters
  ➢ -f hostfile: lists physical core IDs available to execute code
  ➢ -emulator: run on functional emulator
RCCE API: Circular Shift one sided

```c
#include "RCCE.h"
int RCCE_APP() {
    RCCE_init(&argc, &argv);
    NUES = RCCE_num_ues();
    ID = RCCE_ue();

    ID_right = (ID+1)%NUES;
    ID_left = (ID-1+NUES)%NUES;
    size = BUFSIZE*sizeof(double);
    buffer  = (double *) malloc(size);
    cbuffer = (double *) RCCE_malloc(size);

    /* create and initialize flag variables */
    RCCE_flag_alloc(&flag_sent);
    RCCE_flag_alloc(&flag_ack);
    RCCE_flag_write(&flag_sent,
                    RCCE_FLAG_UNSET, ID));
    RCCE_flag_write(&flag_ack,
                    RCCE_FLAG_SET, ID_left));

    for (int round=0; round<nrounds; round++) {
        RCCE_wait_until(flag_ack, RCCE_FLAG_SET);
        RCCE_flag_write(&flag_ack,
                        RCCE_FLAG_UNSET, ID);
        RCCE_put(cbuffer, buffer, size, ID_right);
        RCCE_flag_write(&flag_sent,
                        RCCE_FLAG_SET, ID_left);

        RCCE_wait_until(flag_sent,
                        RCCE_FLAG_SET);
        RCCE_flag_write(&flag_sent,
                        RCCE_FLAG_UNSET, ID);
        RCCE_get(buffer, cbuffer, size, ID);
        RCCE_flag_write(&flag_ack,
                        RCCE_FLAG_SET, ID_left);
    }
```

BUFSIZE must be divisible by 4
Message must fit inside Msg Buff
#include "RCCE.h"
int RCCE_APP() {
    RCCE_init(&argc, &argv);

    for (int round=0; round<nrounds; round++) {
        RCCE_wait_until(flag_ack, RCCE_FLAG_SET);
        RCCE_flag_write(&flag_ack,
            RCCE_FLAG_UNSET, ID);
        RCCE_put(cbuffer, buffer, size, ID_right);
        RCCE_flag_write(&flag_sent,
            RCCE_FLAG_SET, ID_left);
        RCCE_wait_until(flag_sent,
            RCCE_FLAG_SET);
        RCCE_flag_write(&flag_sent,
            RCCE_FLAG_UNSET, ID);
        RCCE_get(buffer, cbuffer, size, ID);
        RCCE_flag_write(&flag_ack,
            RCCE_FLAG_SET, ID_left);
    }
}

RCCE_API: Circular Shift one-sided

RCCE_FLAG flg;
RCCE_flag_alloc(&flg);
RCCE_flag_set(flg, RCCE_FLAG_SET, ID); or RCCE_FLAG_UNSET
RCCE_wait_until(flg, RCCE_FLAG_SET, ID); or RCCE_FLAG_UNSET
RCCE_put(cbuffer, buffer, size, ID);
   *Put my private memory (buffer) into the msg buffer (cbuffer) of core ID*

RCCE_get(buffer, cbuffer, size, ID));
   *Get cbuffer from core ID and move it into my private memory (buffer)*

RCCE_flag_write(&flag_sent,
    RCCE_FLAG_UNSET, ID))
RCCE_flag_write(&flag_ack,
    RCCE_FLAG_SET, ID_left))

BUFSIZE must be divisible by 4
Message must fit inside Msg Buff
RCCE API: “Basic” interface, two sided

- flags needed to make transfers safe.
- Large messages must be broken up to fit into the Msg Buff.

We can hide these details by letting library manage flags +MPB:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCCE_wait_until</td>
<td>Sets the flag to RCCE_FLAG_SET if the specified flag is set.</td>
</tr>
<tr>
<td>RCCE_flag_write</td>
<td>Writes the flag to the specified state for the specified ID.</td>
</tr>
<tr>
<td>RCCE_put</td>
<td>Puts the data from the cbuffer to the buffer for the specified ID.</td>
</tr>
<tr>
<td>RCCE_flag_write</td>
<td>Writes the flag to the specified state for the specified ID.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCCE_send</td>
<td>Sends private memory (buffer) to core ID.</td>
</tr>
<tr>
<td>RCCE_recv</td>
<td>Receives private memory (buffer) from core ID.</td>
</tr>
</tbody>
</table>

This is Synchronous message passing ... the send and receive do not return until the communication is complete on both sides.
```c
#include <string.h>
#include "RCCE.h"

int RCCE_APP() {

    RCCE_init(&argc, &argv);
    NUES = RCCE_num_ues();

    ID = RCCE_ue();

    ID_right = (ID+1)%NUES;
    ID_left = (ID-1+NUES)%NUES;

    int size  = BUFSIZE*sizeof(double);
    buffer   = (double *) malloc (size);
    buffer2 = (double *) malloc (size);

    for (int round=0; round<nrounds; round++) {
        for (int c = 0; c<2; c++) {
            if ((ID+c)%2)
                RCCE_send(buffer, size, ID_right);
            else
                RCCE_recv(buffer2, size, ID_left);
        }
        memcpy(buffer, buffer2, size);
    }
}
```

RCCE API: Circular Shift with 2-sided Basic interface

BUFSIZE may be anything
Message need not fit inside Msg Buf

Hides buffer and flag allocation, messages “packetizing”, and flag synchronization.

Anticipate most programmers will use this RCCE version
RCCE: Supporting Details

- Using RCCE and example RCCE code
- Additional RCCE implementation details
- RCCE and the MPI programmer
RCCE Implementation details:
One-sided message passing; safely but blindly transport data between private memories

RCCE_put(char *target, char *source, size_t size, int ID)
{
    target = target + (RCCE_MPB[ID] - RCCE_MPB[RCCE_IAM]);
    RCCE_cache_invalidate();
    memcpy(target, source, size);
}
RCCE_get(char *target, char *source, size_t size, int ID)
{
    source = source + (RCCE_MPB[ID] - RCCE_MPB[RCCE_IAM]);
    RCCE_cache_invalidate();
    memcpy(target, source, size);
}

RCCE_MPB[ID] = start of MPB for UE “ID”
RCCE_IAM = library shorthand for calling UE
target/source cache line aligned, size%32=0, data fits inside MPB
RCCE Implementation details:
Two-sided message passing; safely transport data between private memories, with handshake.

RCCE_send(char *privbuf, char *combuf, RCCE_FLAG *ready, 
        RCCE_FLAG *sent, size_t size, int dest) {
    RCCE_put(combuf, privbuf, size, RCCE_IAM);
    RCCE_flag_write(sent, SET, dest);
    RCCE_wait_until(*ready, SET);
    RCCE_flag_write(ready, UNSET, RCCE_IAM);
}

RCCE_recv(char *privbuf, char *combuf, RCCE_FLAG *ready, 
        RCCE_FLAG *sent, size_t size, int source) {
    RCCE_wait_until(*sent, SET);
    RCCE_flag_write(sent, UNSET, RCCE_IAM);
    RCCE_get(privbuf, combuf, size, source);
    RCCE_flag_write(ready, SET, source); }

- Body gets called in a loop (+ padding if necessary) for large messages
- send and recv asymmetric: needed to avoid deadlock
- No size or alignment restrictions
- We get rid of these parameters in our “basic” interface (≈MPI)
RCCE Implementation Details: Flags

- Flags implemented two ways
  1. whole MPB memory line (96 flags, 30% of MPB)
  2. single bit (1 MPB memory line for all flags)
     - Control write access through atomic test&set register, implementing lock.
     - No need to protect read access.

- Implications of the two types of flags:
  - Single bit saves MPB memory but you pay with a higher latency.
  - Whole cache line wastes memory but lowers latency.
RCCE Implementation Details:
RCCE flag write scenario (single bit)

```c
void RCCE_flag_write(RCCE_FLAG *flag, RCCE_FLAG_STATUS val, int ID) {
    volatile unsigned char val_array[RCCE_LINE_SIZE];

    /* acquire lock so nobody else fiddles with the flags on the target core */
    RCCE_acquire_lock(ID);
    /* copy line containing flag to private memory */
    RCCE_get(val_array, flag->line_address, RCCE_LINE_SIZE, ID);
    /* write “val” into single bit corresponding to flag */
    RCCE_write_bit_value(val_array, flag->location, val);
    /* copy line back to MPB */
    RCCE_put(flag->line_address, val_array, RCCE_LINE_SIZE, ID);
    /* release write lock for the flags on the target core */
    RCCE_release_lock(ID);
}
```

```c
void RCCE_acquire_lock(int ID) {
    while (!(*(physical_lockaddress[ID])) & 0x01));
}
```

```c
void RCCE_release_lock(int ID) {
    *(physical_lockaddress[ID]) = 0x0;
}
```

physical_lockaddress[ID]: address of test&set register on core with rank ID.  
RCCE_flag_read does not need lock protection.
RCCE: Supporting Details

- Using RCCE and example RCCE code
- Additional RCCE implementation details
- RCCE and the MPI programmer
RCCE vs MPI

• No opaque data types in RCCE, so no MPI-style handles, only pointers

• No RCCE_datatype, except for reductions

• No communicators, except in collective communications

• Only synchronous communications
  + No message bookkeeping
  – No overlap of computations/communications
  – Deadlock?

• RCCE has low overhead due short communication stack:
  – RCCE_send→RCCE_put→memcpy
RCCE vs MPI: Avoiding deadlock

- If sending and receiving UE sets overlap, deadlock is possible. Cause: cycles in communication graph (cyclic dependence).
- If no cycles, communication may serialize
- Solution:
  - Divide communication pattern into disjoint send-receive UE sets (bipartite graphs), execute in phases.
  - Number of phases depends on pattern.
  - For permutation pattern, two phases min, three max:
    1. Each permutation can be divided into cycles (length L)
    2. If L even, red/black coloring suffices.
    3. If L odd (2n+1), apply 2. to 2n UEs, then finish communications for last UE. Each cycle takes $O(1)$ time.
  - Note: coloring is wrt position in cycle, not UE rank; may need different phase colorings for different patterns.
RCCE vs MPI: Avoiding deadlock

Programmer just posts (i)sends and (i)receives as needed.

Programmer must pair all sends and receives.
RCCE vs MPI: Avoiding deadlock

- pseudo-code example from HPC application:

```c
MPI:   if (!IAM_LEFTMOST) {
    MPI_Irecv(from_left);
    MPI_wait(on_isend);
    MPI_wait(on_irecv);
}
    compute;
    if (!IAM_RIGHTMOST) MPI_isend(to_right);
```

```c
RCCE:  if (!IAM_LEFTMOST)
    for (phase = 0; phase < 3; phase++) {
        if (send_color==phase) RCCE_send(to_right);
        if (recv_color==phase) RCCE_recv(from_left);
    }
    compute;
```

- Notes:
  - MPI version cell based; RCCE version interface based
  - RCCE fairly easy to grok, but requires restructuring to interleave sends/recvs