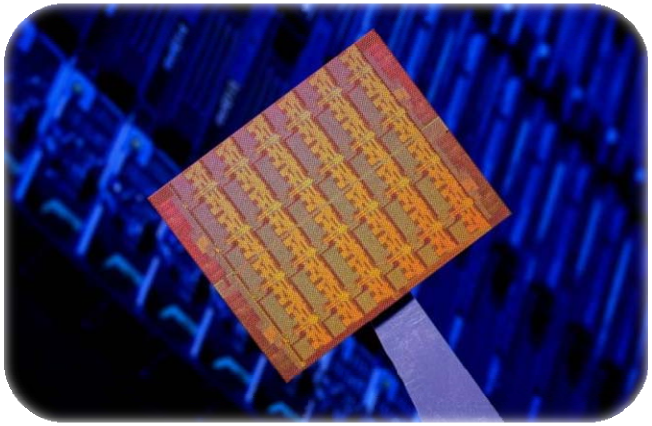
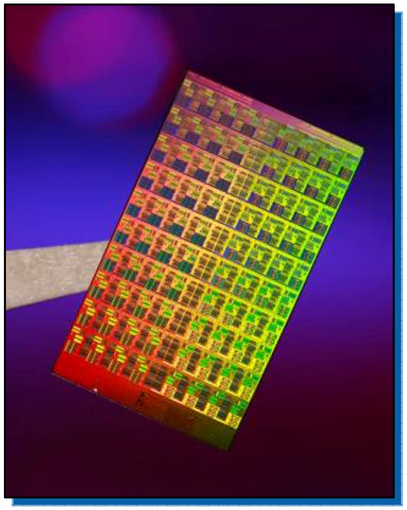


The Future of Many Core Computing:

A tale of two processors



Tim Mattson

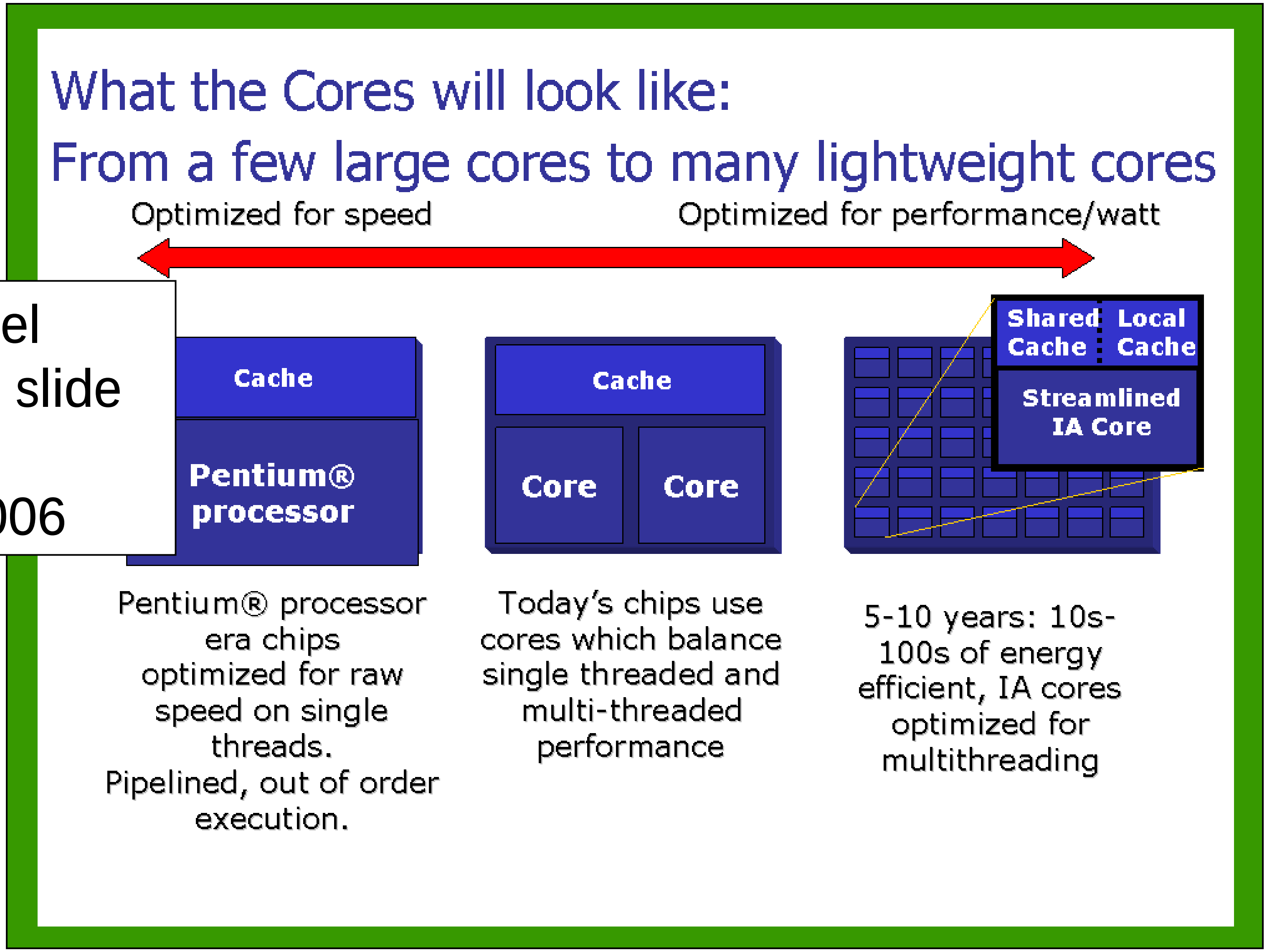
Intel Labs

January 2010

Disclosure

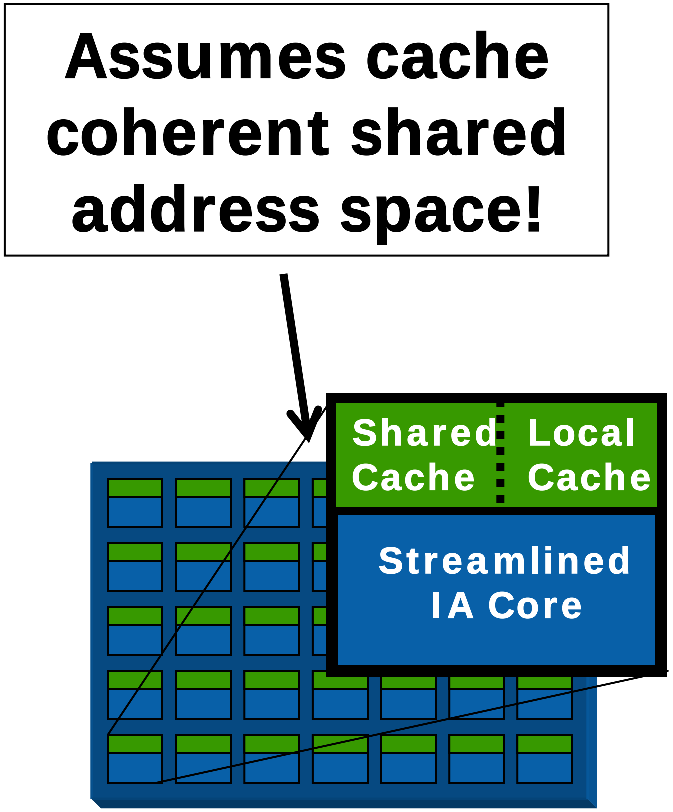
- The views expressed in this talk are those of the speaker and not his employer.
- I am in a research group and know nothing about Intel products. So anything I say about them is highly suspect.
- This was a team effort, but if I say anything really stupid, it's all my fault … don't blame my collaborators.

A common view of many-core chips



An Intel Exec's slide from IDF'2006

Challenging the sacred cows



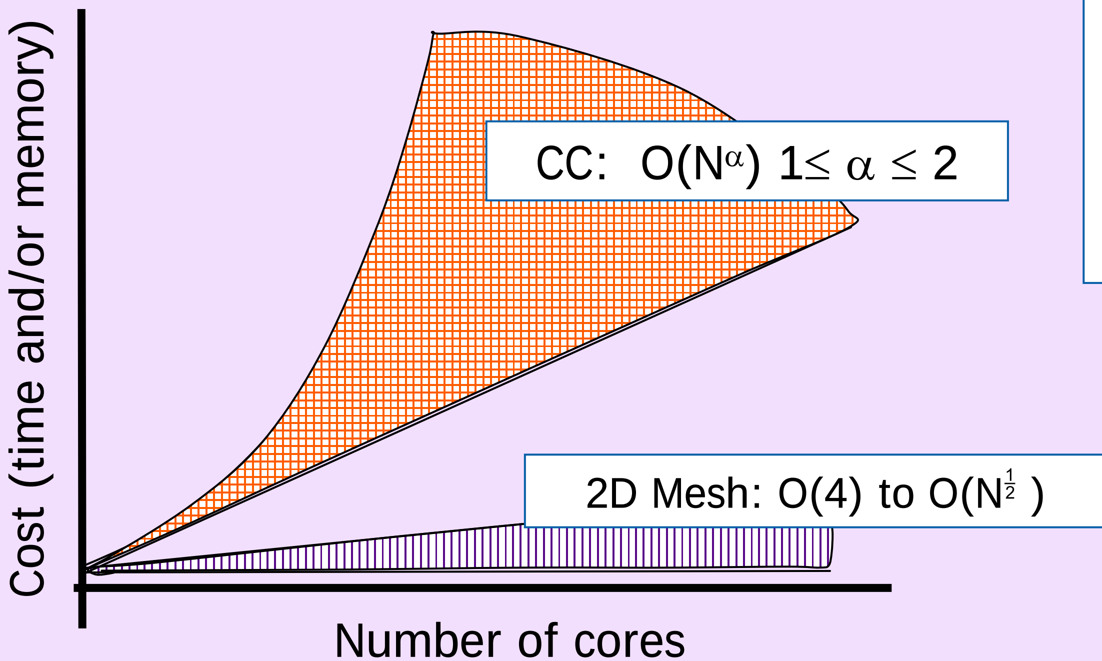
... IA cores optimized for multithreading

- 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

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.

Assume an app whose performance is not bound by memory bandwidth



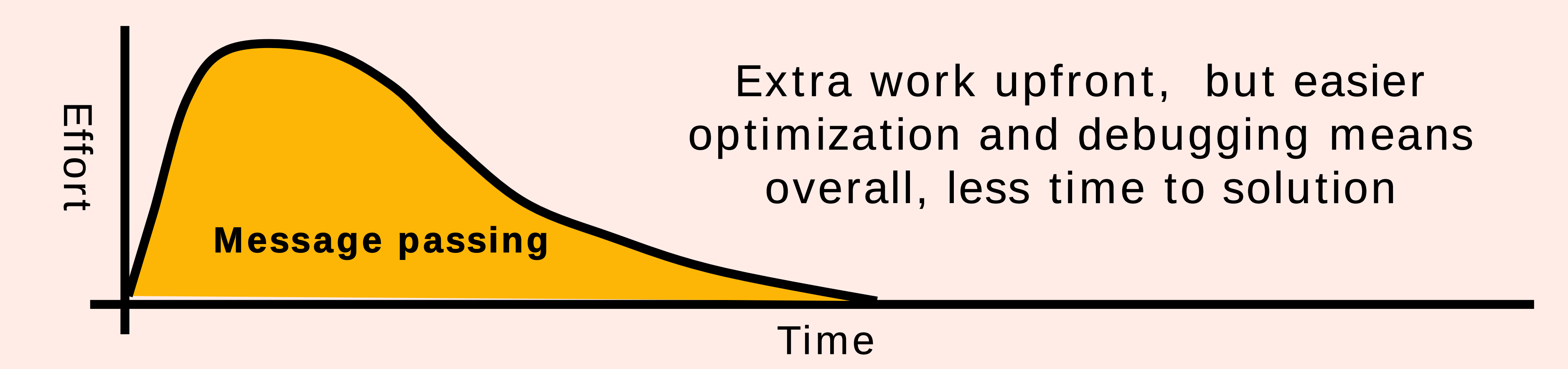
For a scalable, directory based scheme, CC incurs an N-body effect ... cost scales at best linearly (Fixed memory size as cores are added) and at worst quadratically (memory grows linearly with number of cores).

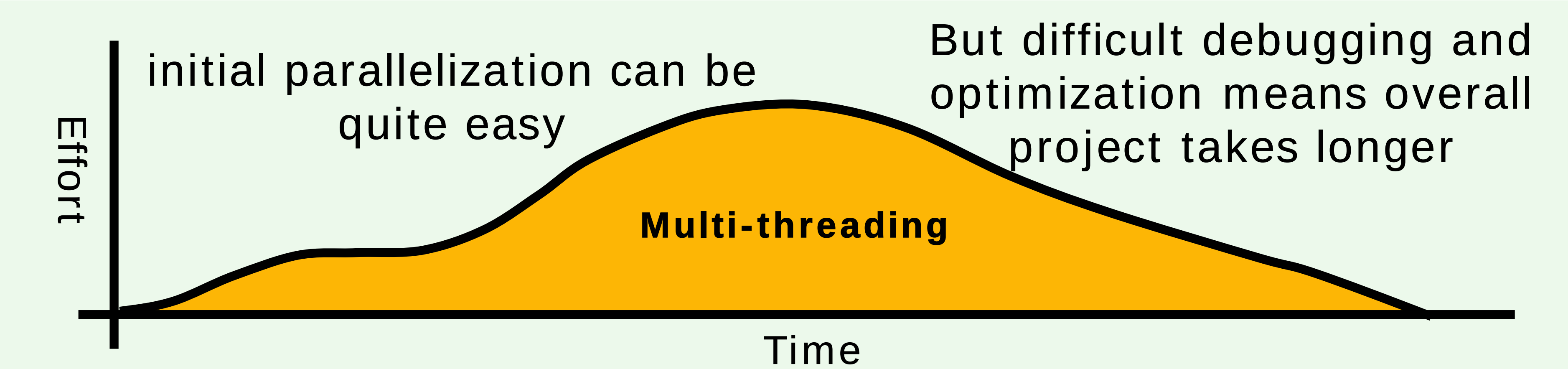
HW Dist. Mem. HW cost scales at best a fixed cost for the local neighborhood and at worst as the diameter of the network.

... 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.



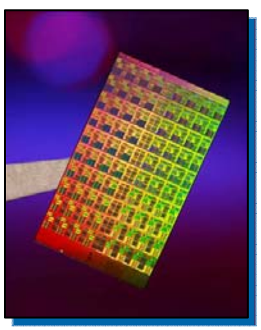


Proving that a shared address space program using semaphores is race free is an NP-complete problem*

*P. N. Klein, H. Lu, and R. H. B. Netzer, Detecting Race Conditions in Parallel Programs that Use Semaphores, Algorithmica, vol. 35 pp. 321–345, 2003

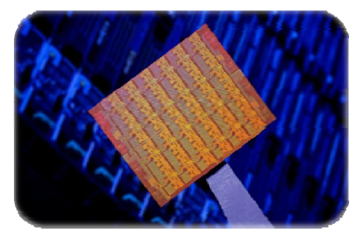
The many core design challenge

- **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).



80 core Research processor

Intel's "TeraScale" processor research program is addressing these questions with a series of Test chips … two so far.

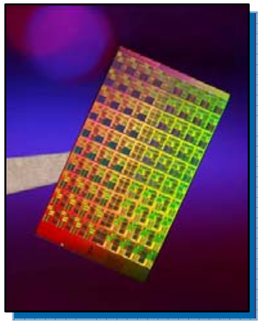


48 core SCC processor

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

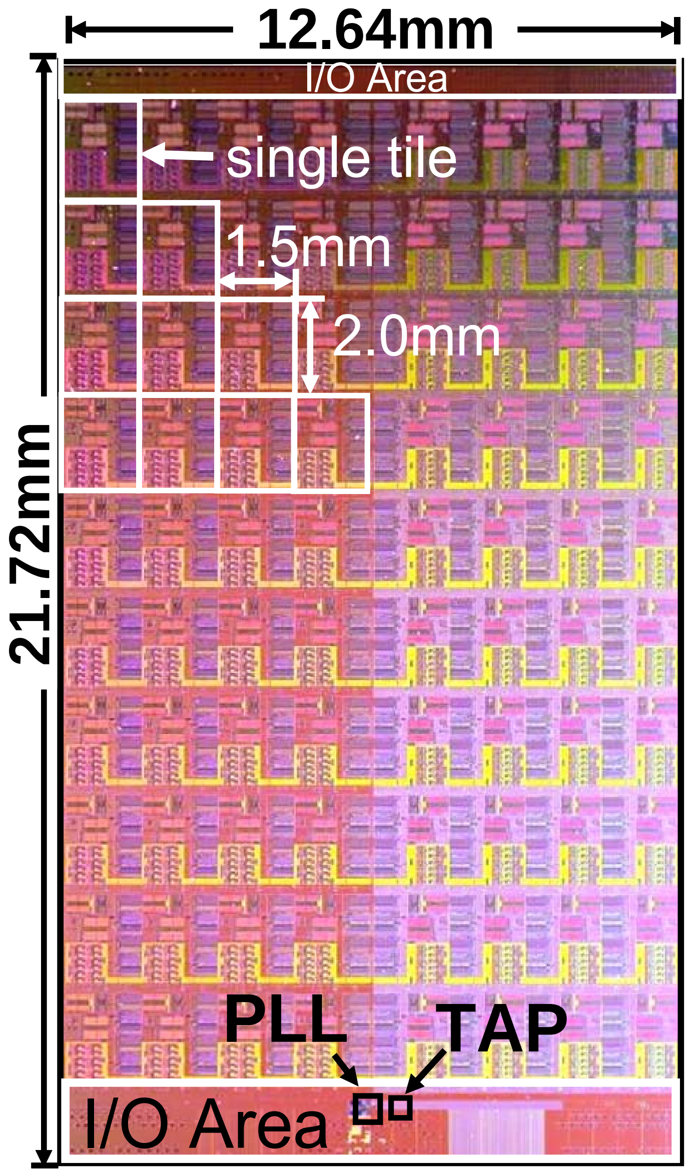
Acknowledgements

- 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.

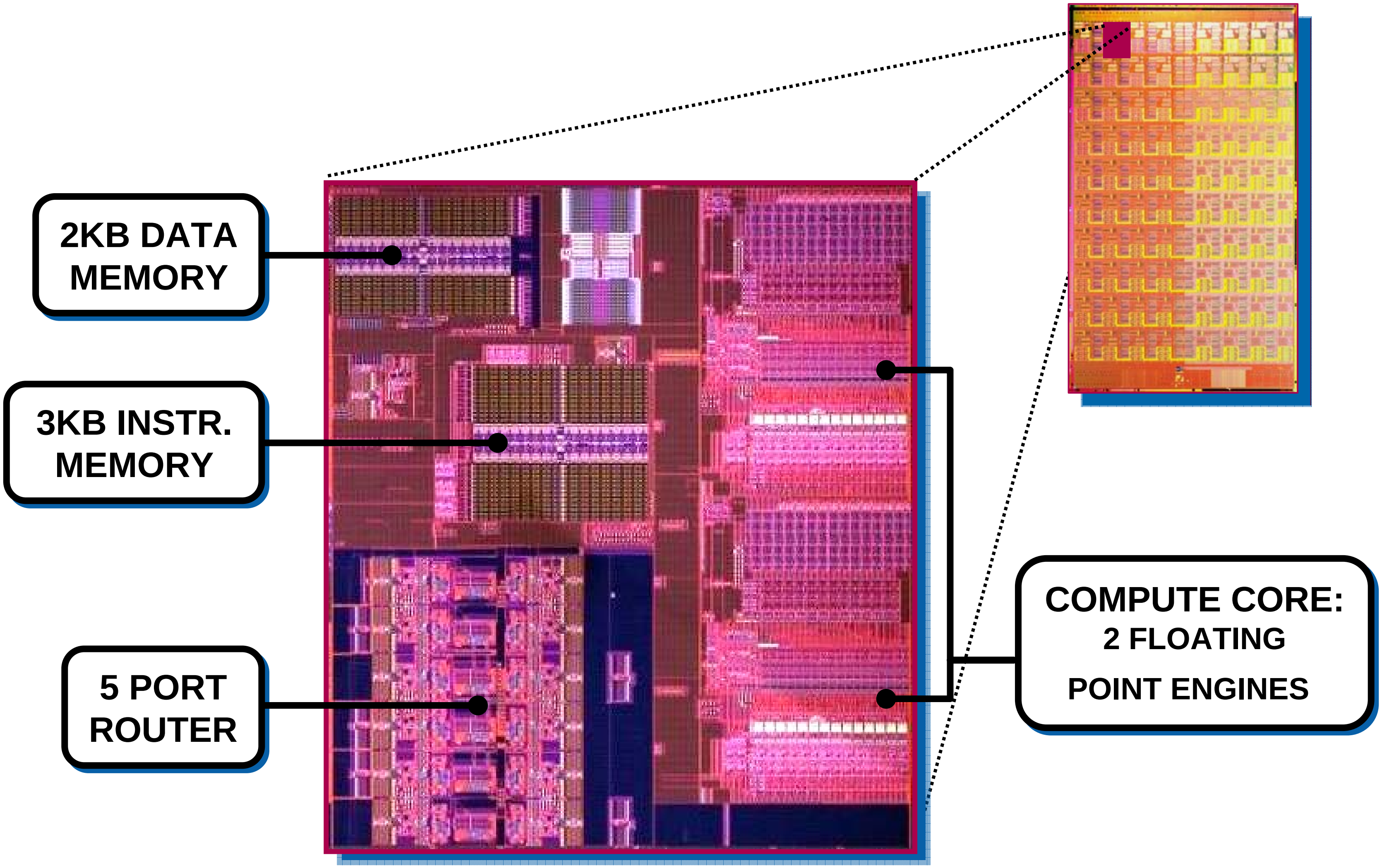
Intel's 80 core terascale processor

Die Photo and Chip Details

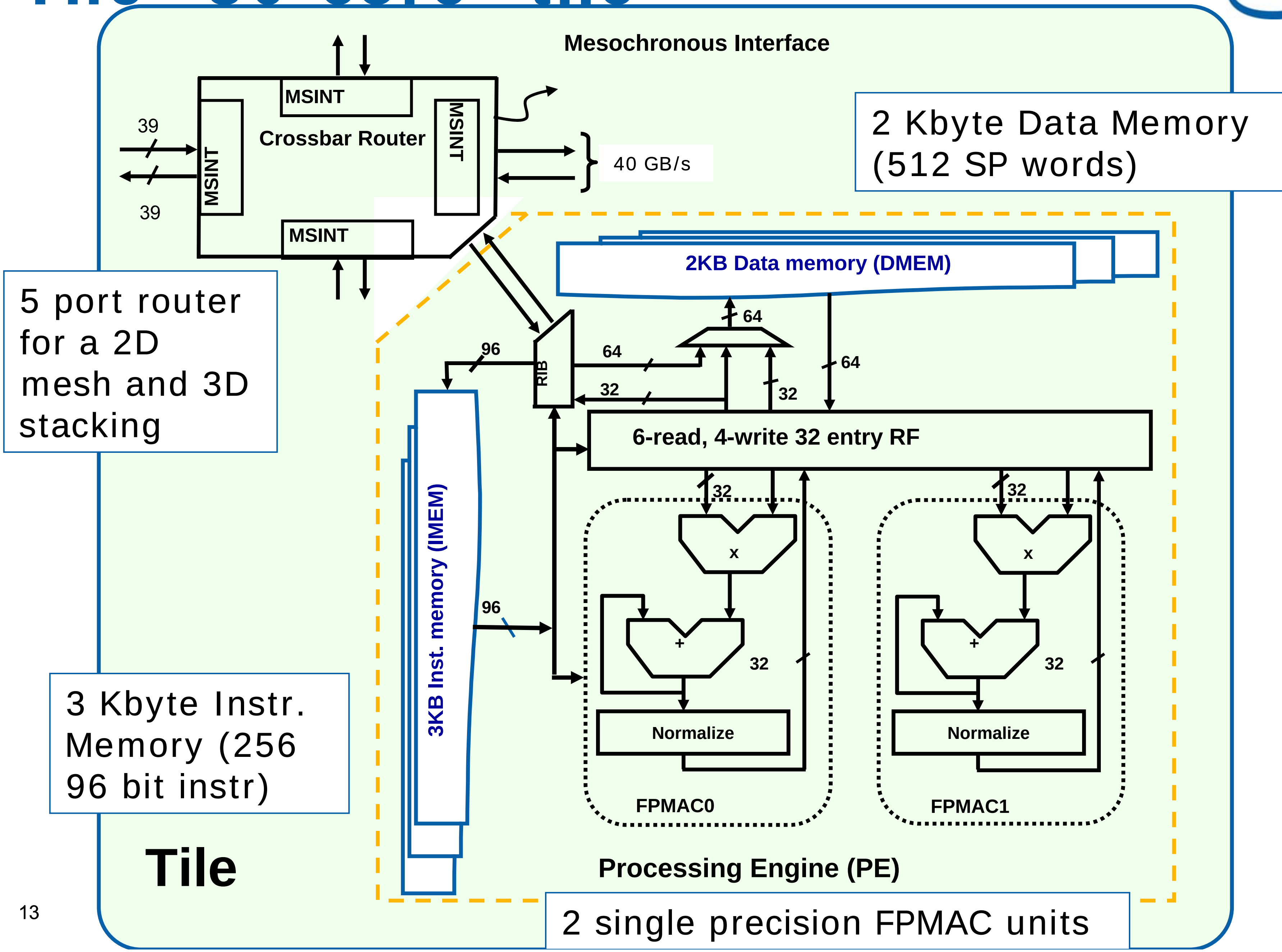


- Basic statistics:
  - 65 nm CMOS process
  - 100 Million transistors in 275 mm^2
  - 8x10 tiles, $3mm^2$/tile
  - Mesosynchronous clock
  - 1.6 SP TFLOP @ 5 Ghz and 1.2 V
  - 320 GB/s bisection bandwidth
  - Variable voltage and multiple sleep states for explicit power management

We've made good progress with the hardware: Intel's 80 core test chip (2006)



The "80-core" tile



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 ...

SP = single precision, FMAC = floating point multiply accumulate, FLOP = floating point operations

Full Instruction Set

MULT	Multiply operands	FPU
ACCUM	Accumulate with previous result	FPU
LOAD, STORE	Move a pair of floats between register file & data memory.	Load / Store
LOADO, STOREO, OFFSET	Move a pair of floats between the register file and data memory at address plus OFFSET.	Load / Store
SENDI[H|A|D|T]	Send instr. header, address, data, and tail	SND / RCV
SENDD[H|A|D|T]	Send Data header, address, data, and tail	SND / RCV
WFD	Stall while waiting for data from any tile.	SND / RCV
STALL	Stall program counter (PC), waiting for a new PC.	SND / RCV
BRNE, INDEX	INDEX sets a register for loop count. BRNE branches while the index register is greater than zero	Program flow
JUMP	Jump to the specified program counter address	Program flow
NAP	Put FPUs to sleep	Sleep
WAKE	Wake FPUs from sleep	Sleep

Instruction word and latencies

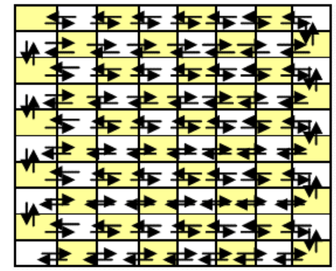
FPU (2)	LOAD/STORE	SND/RCV	PGM FLOW	SLEEP

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

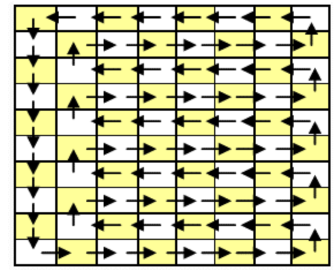
Instruction Type	Latency (cycles)
FPU	9
LOAD/STORE	2
SEND/RECEIVE	2
JUMP/BRANCH	1
NAP/WAKE	1

What did we do with the chip?

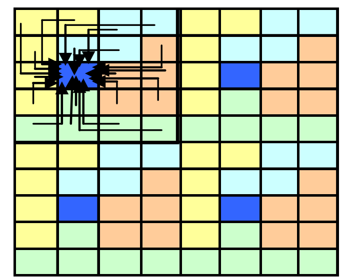
- 4 applications kernels
  - **Stencil**
    - 2D PDE solver (heat diffusion equation) using a Gauss Seidel algorithm



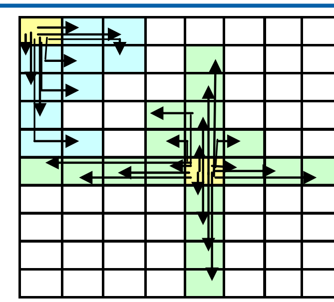
  - **SGEMM (Matrix Multiply)**
    - C = A*B with rectangular matrices



  - **Spreadsheet**
    - Synthetic benchmark … sum dense array of rows and columns (local sums in one D, reduction in the other D)



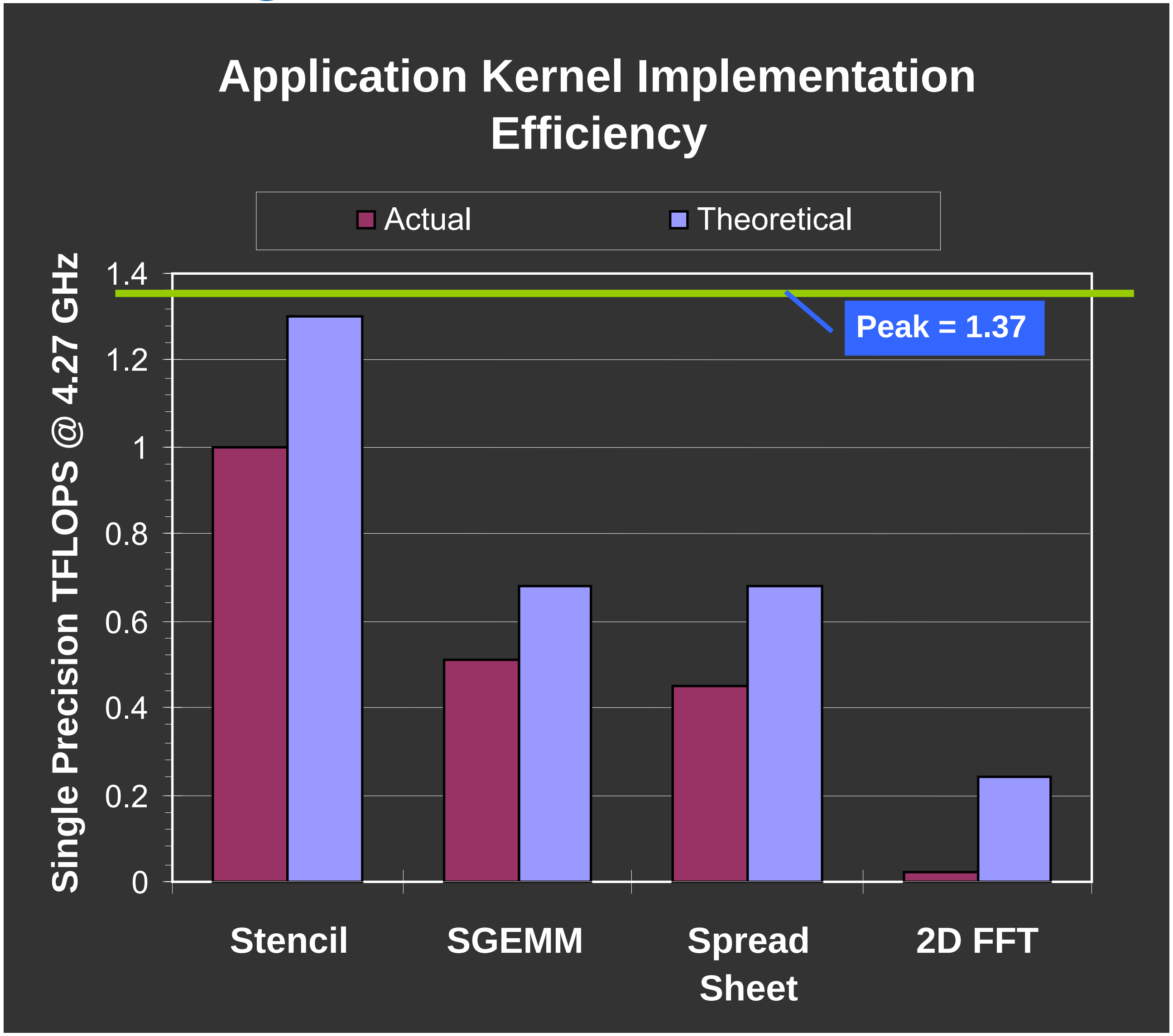
  - **2D FFT**
    - 2D FFT of dense array on an 8 by 8 subgrid.



Communication Patterns

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

Programming Results

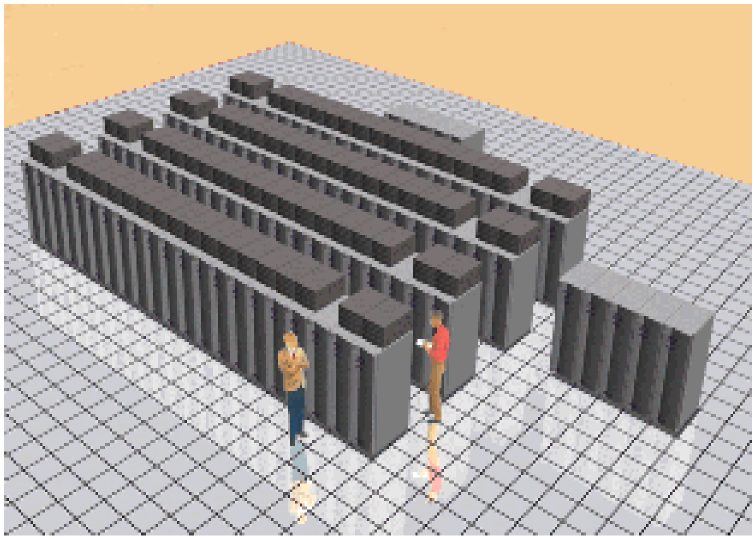


Theoretical numbers from operation/communication counts and from rate limiting bandwidths.

1.07V, 4.27GHz operation 80 C

Why this is so exciting!

First TeraScale* computer: 1997



Intel's ASCI Option Red

Intel's ASCI Red Supercomputer

9000 CPUs

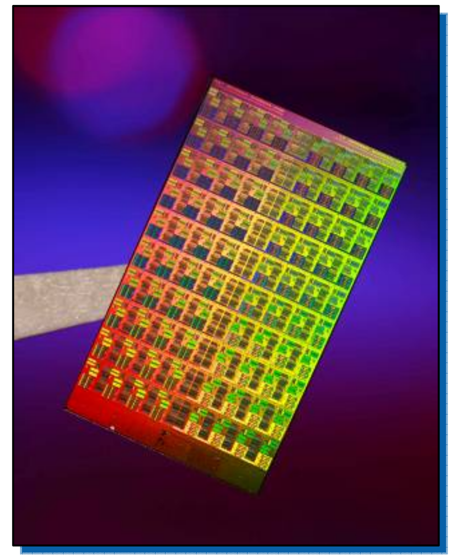
one megawatt of electricity.

1600 square feet of floor space.

*Double Precision TFLOPS running MP-Linpack

10 years later

First TeraScale% chip: 2007



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%

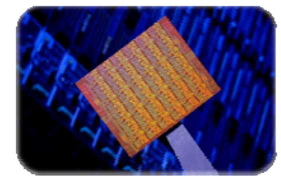
Lessons: Part 2

- 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	Rob Van der Wijngaart Tim Mattson
Developer tools (icc and MKL)	Patrick Kennedy

- SCC System software:

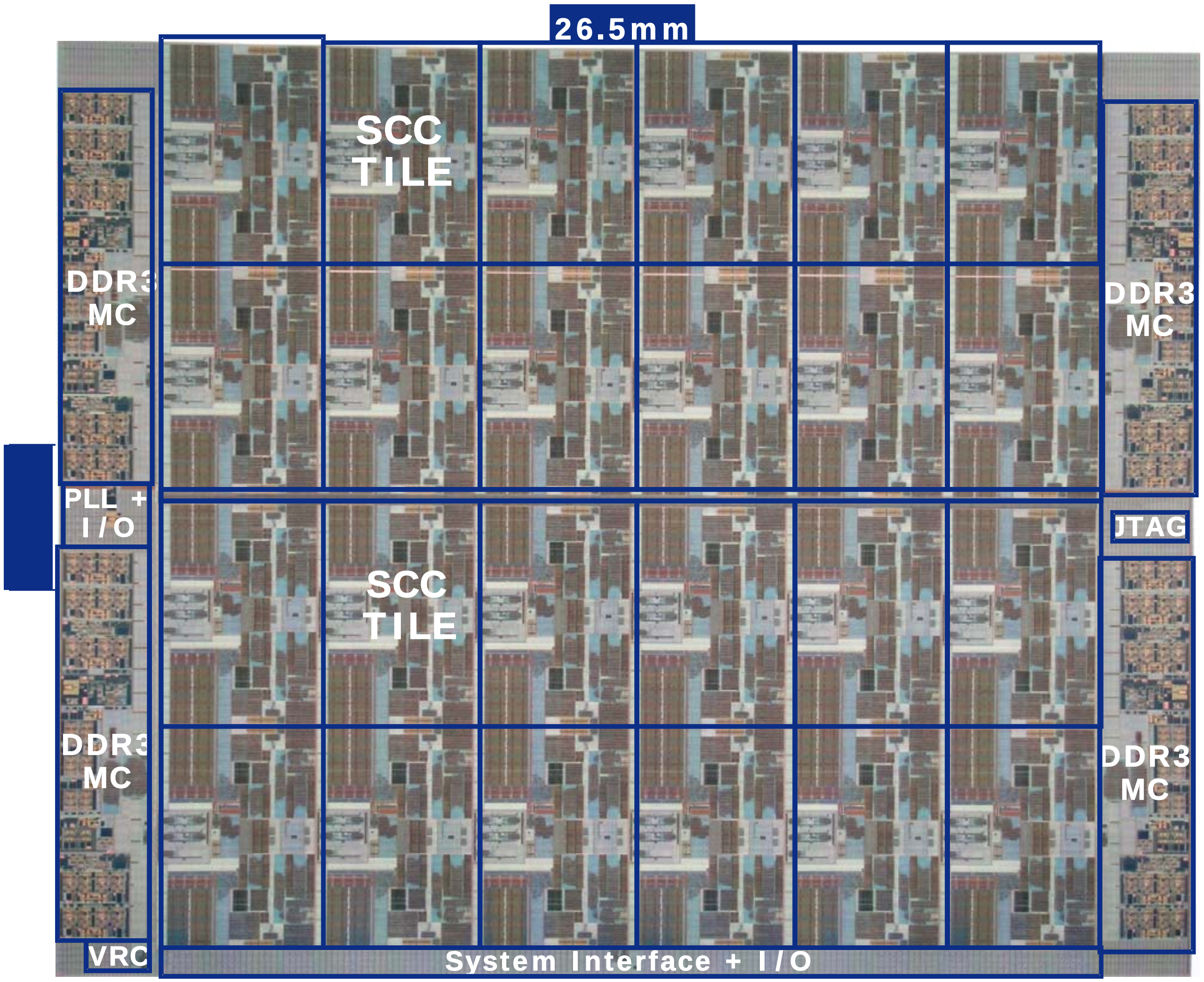
Management Console software	Michael Riepen
BareMetalC workflow	Michael Riepen
Linux for SCC	Thomas Lehnig Paul Brett
System Interface FPGA development	Matthias Steidl
TCP/IP network driver	Werner Haas

- And the HW-team that worked closely with the SW group:

Jason Howard, Yatin Hoskote, Sriram Vangal, Nitin Borkar, Greg Ruhl

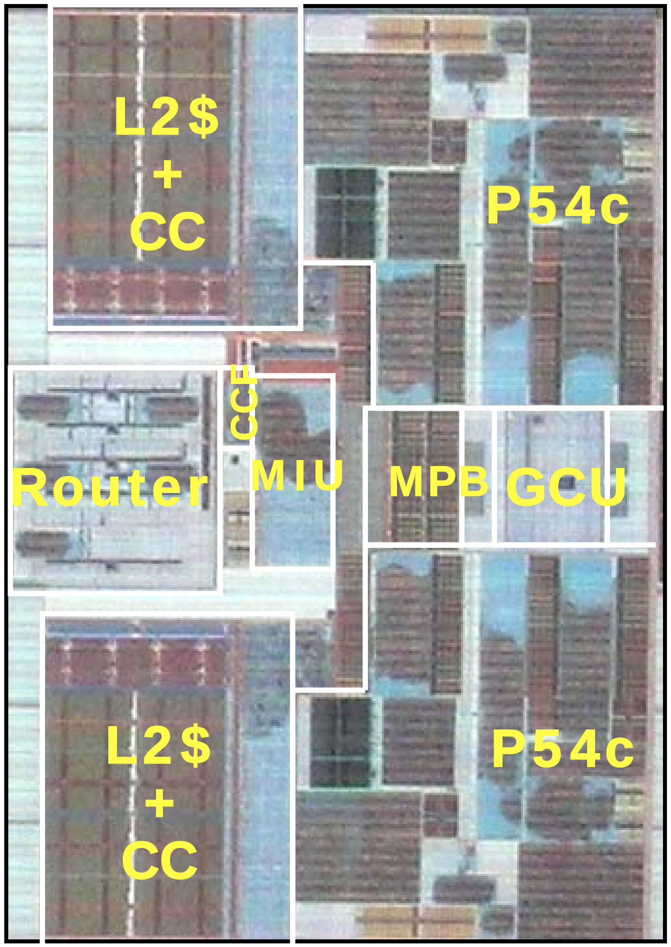
SCC full chip

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



Technology	45nm Process
Interconnect	1 Poly, 9 Metal (Cu)
Transistors	Die: 1.3B, Tile: 48M
Tile Area	18.7mm^2
Die Area	567.1mm^2

SCC Dual-core Tile

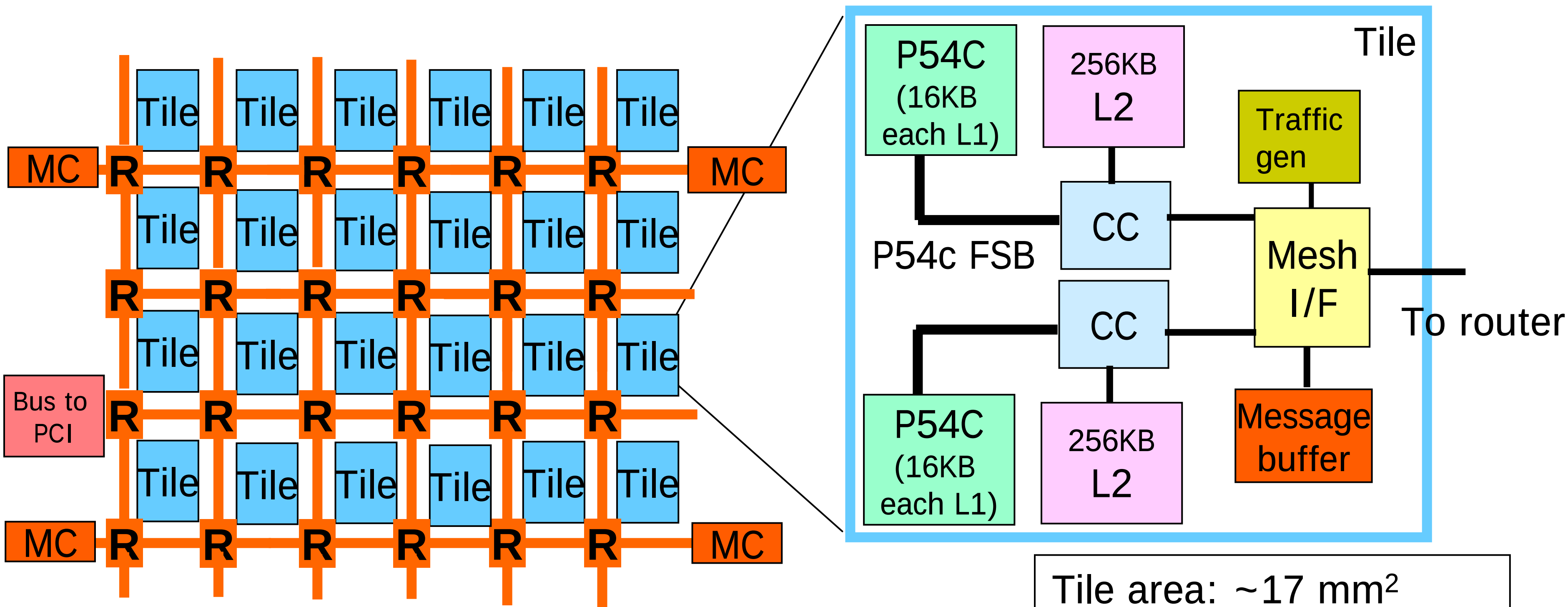


- 2 P54C cores (16K L1$/core)
- 256K L2$ per core
- 8K Message passing buffer
- Clock Crossing FIFOs b/w Mesh interface unit and Router

- Tile area 18.7mm^2
- Core are 3.9mm^2
- Cores and uncore units @1GHz
- Router @2GHz

Hardware view of SCC

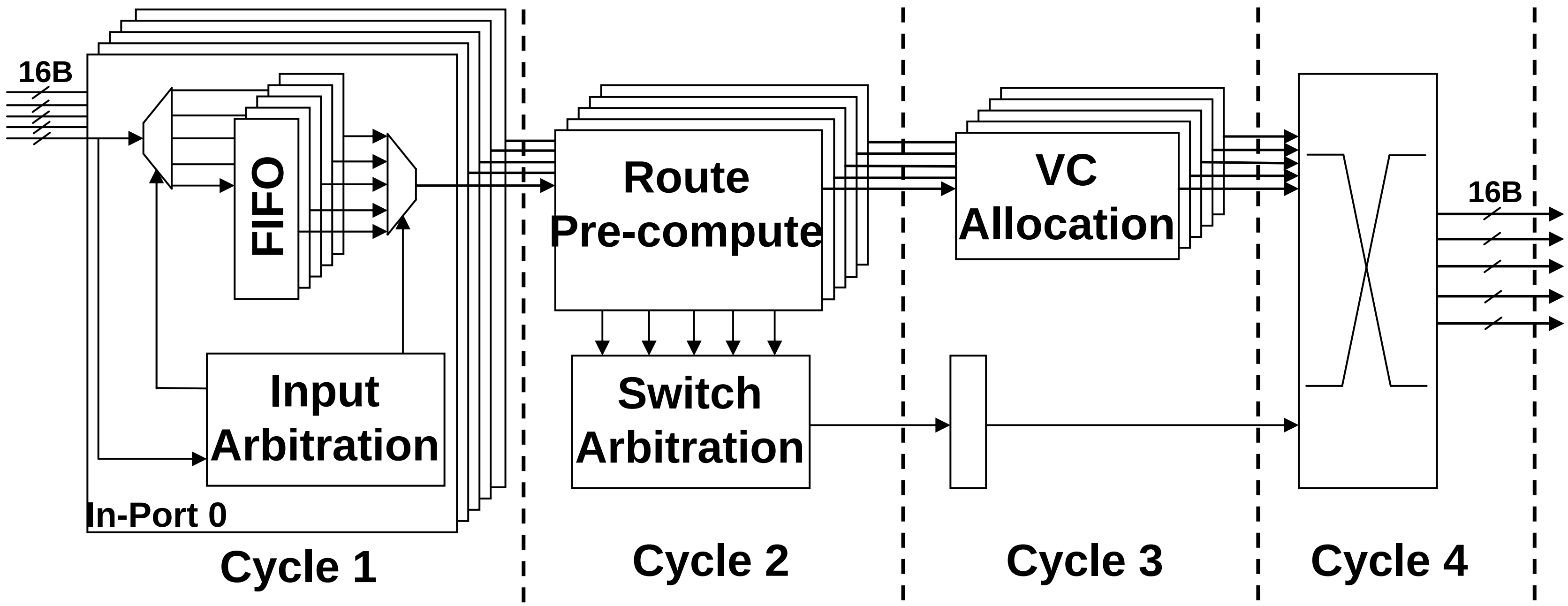
- 48 P54C cores in 6x4 mesh with 2 cores per tile
- 45 nm, 1.3 B transistors, 25 to 125 W
- 16 to 64 GB total main memory using 4 DDR3 MCs



Tile area: ~17 mm²
SCC die area: ~567 mm²

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

Router Architecture



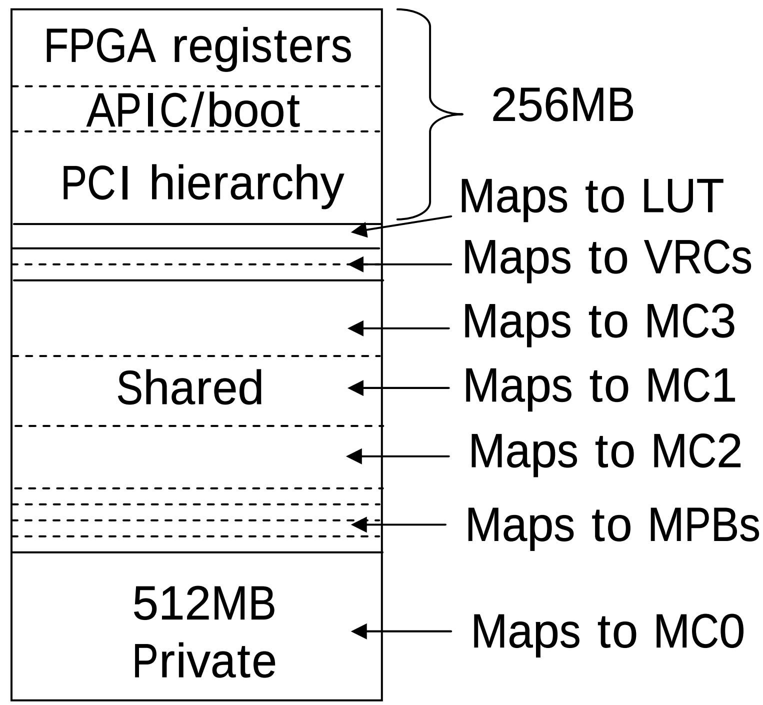
Frequency	2GHz @ 1.1V
Latency	4 cycles
Link Width	16 Bytes
Bandwidth	64GB/s per link
Architecture	8 VCs over 2 MCs
Power Consumption	500mW @ 50°C

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

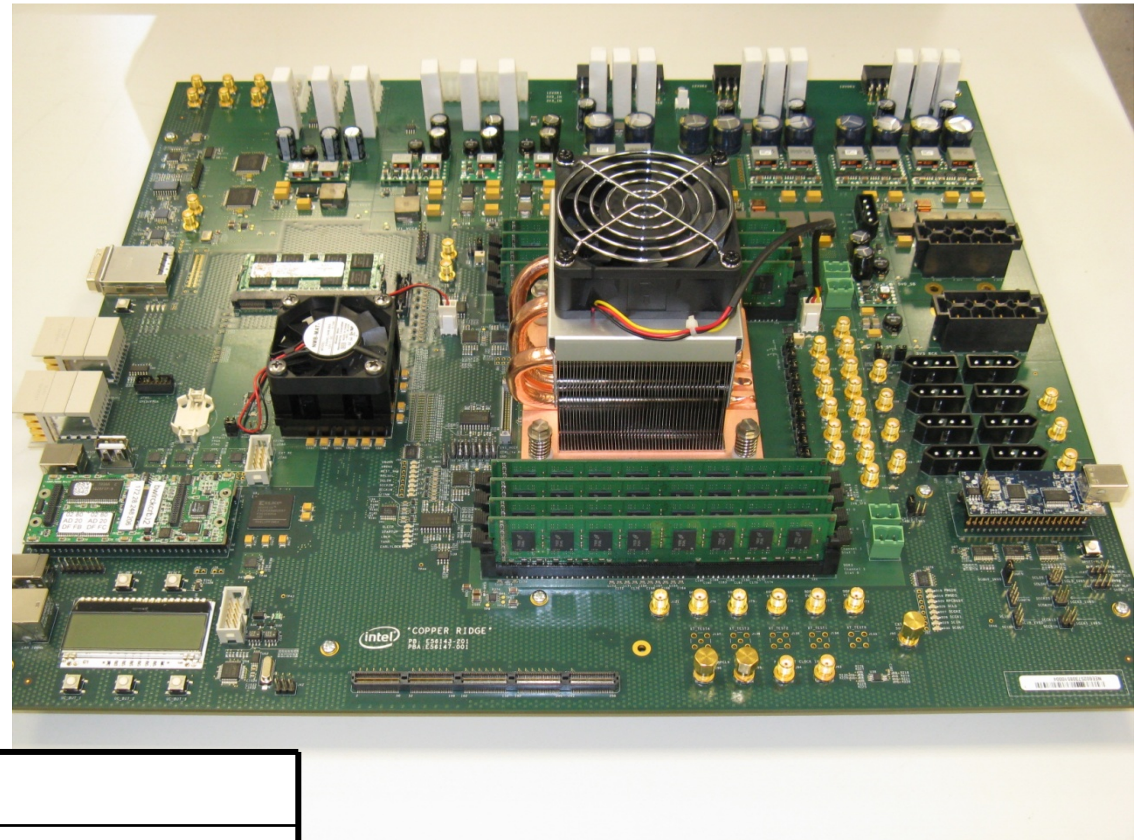
Core Memory Management

- 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



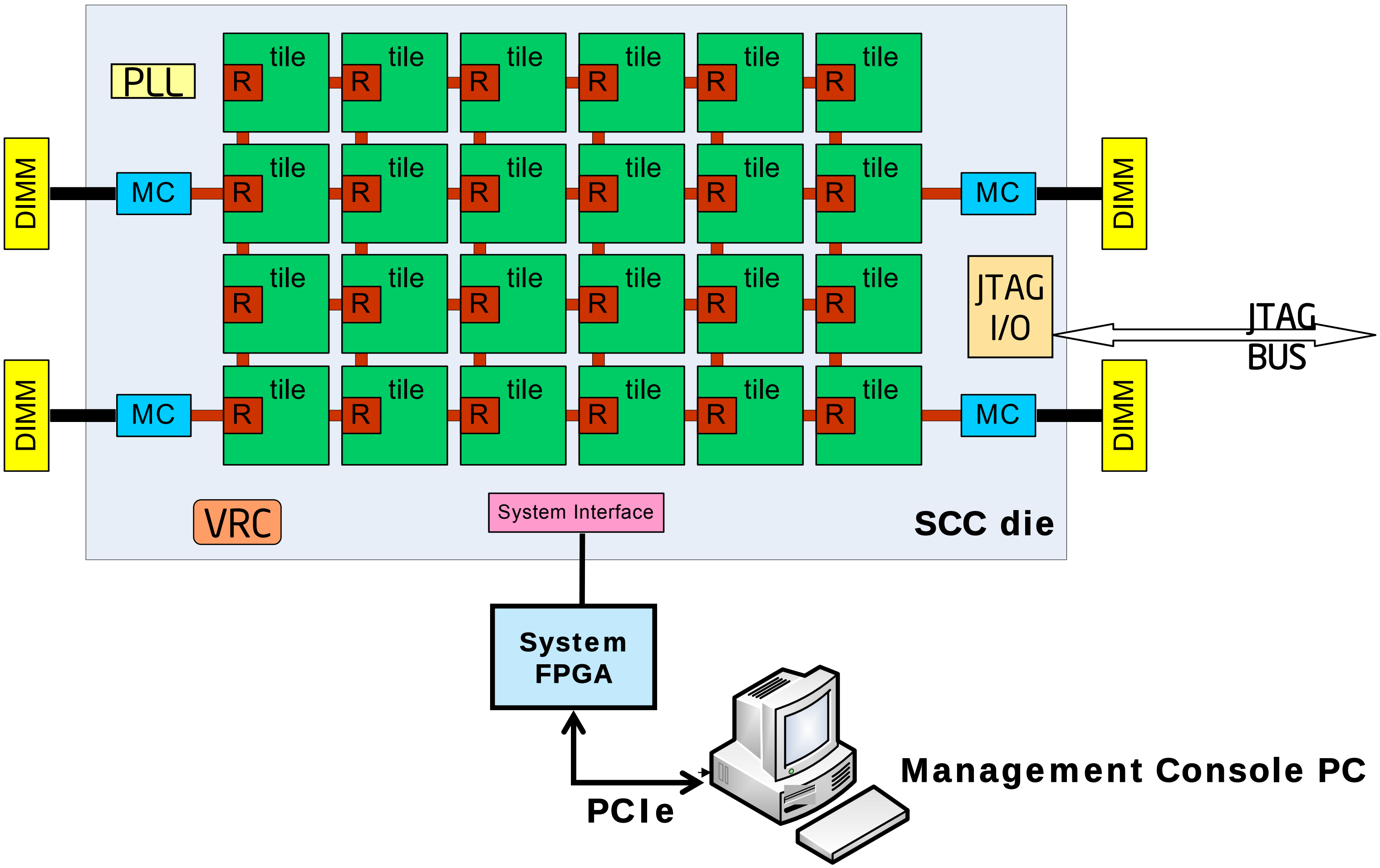
 MC# = one of the 4 memory controllers, MPB = message passing buffer, VRC's = Voltage Regulator control

Package and Test Board

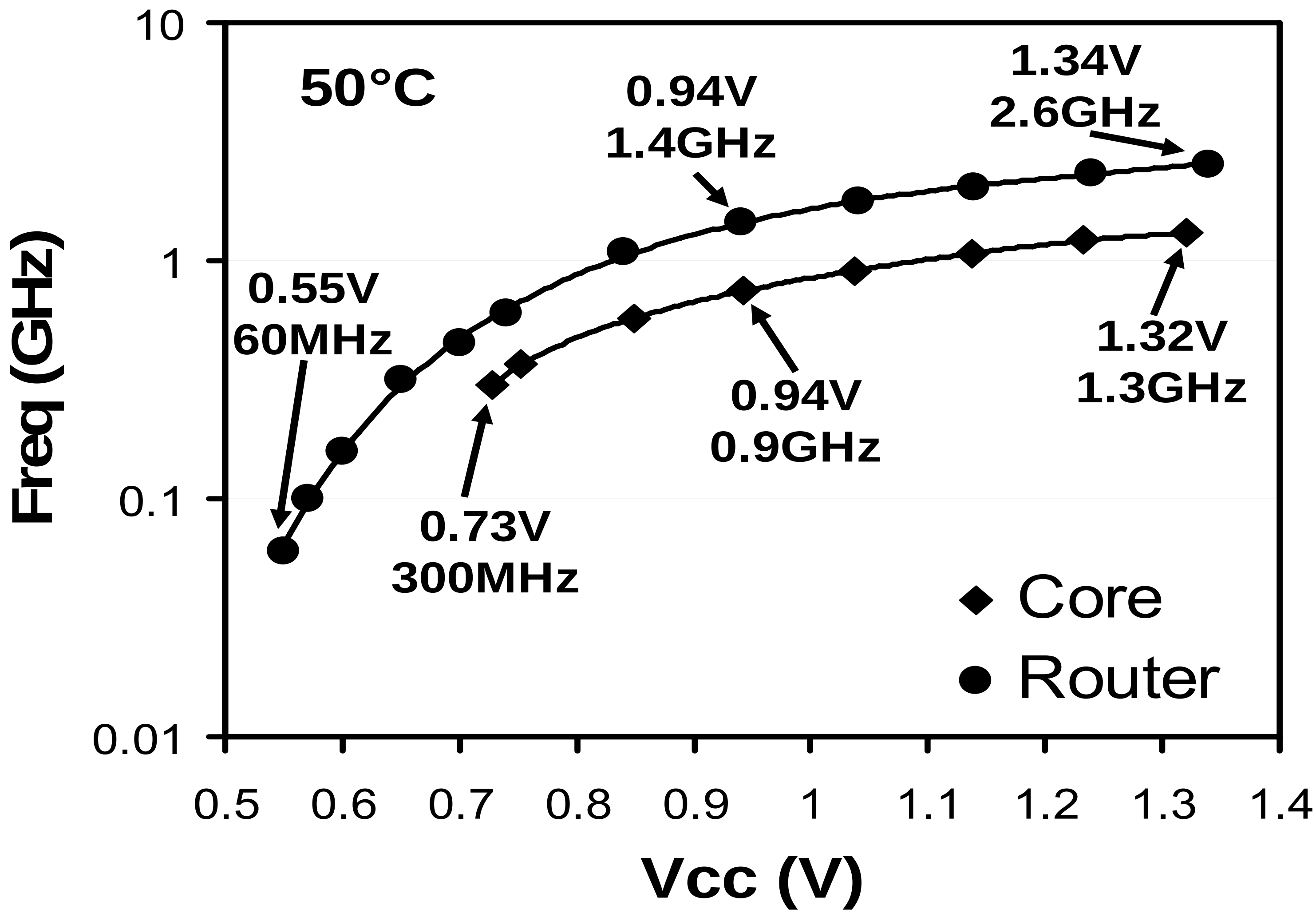


Technology	45nm Process
Package	1567 pin LGA package
	14 layers (5-4-5)
Signals	970 pins

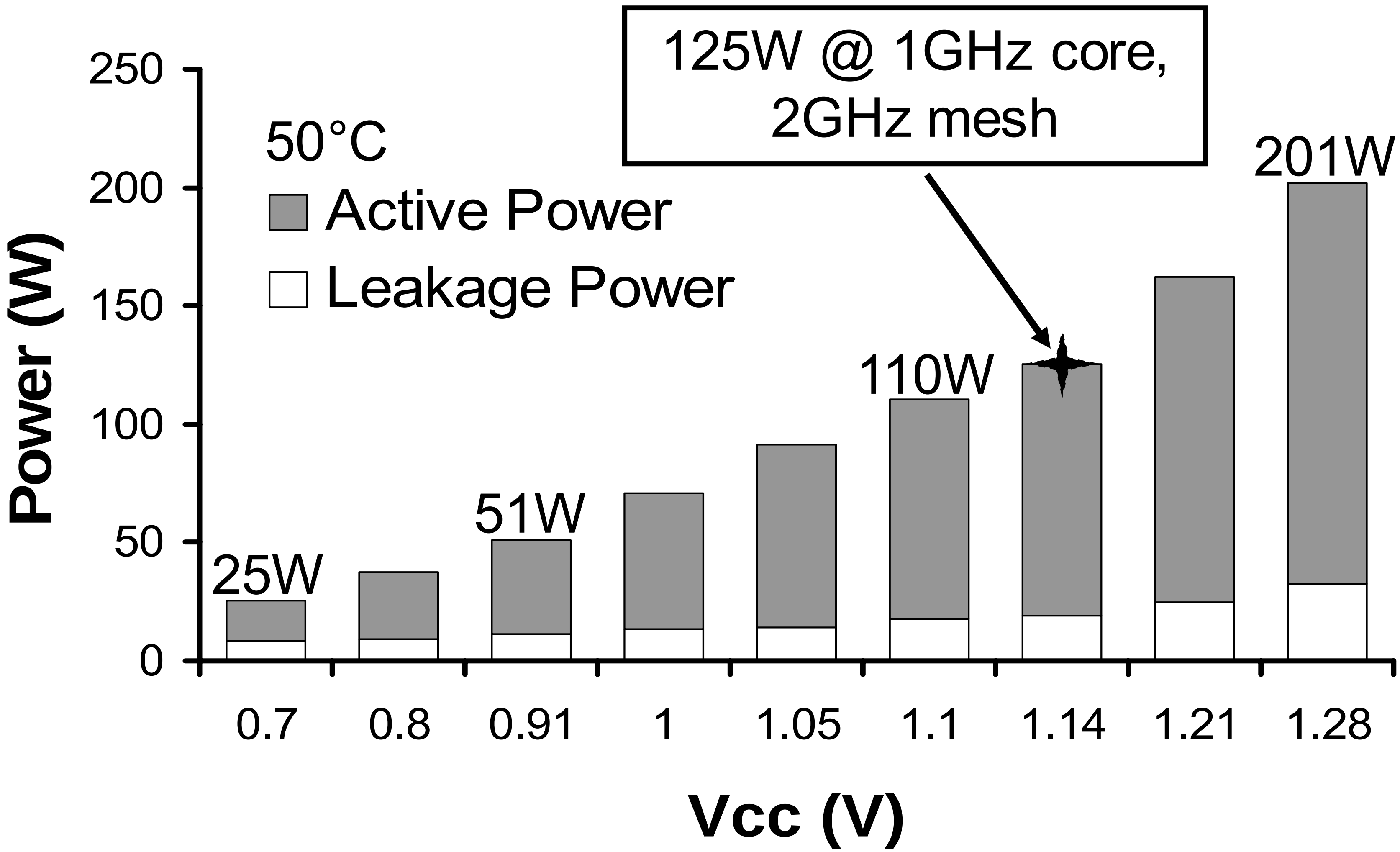
SCC system overview



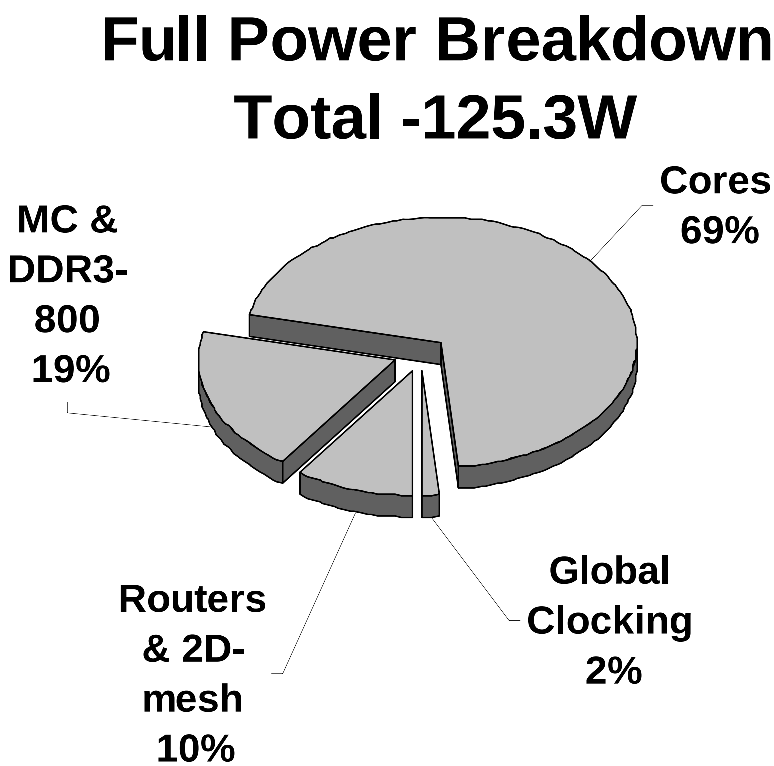
Core & Router Fmax



Measured full chip power

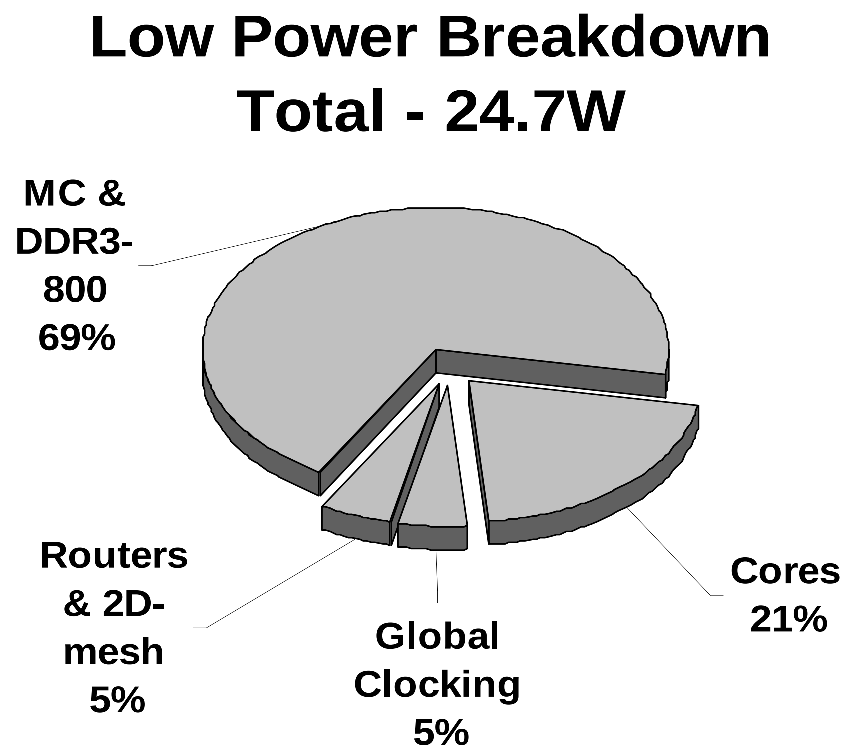


Power breakdown



Clocking: 1.9W Routers: 12.1W
Cores: 87.7W MCs: 23.6W

Cores-1GHz, Mesh-2GHz, 1.14V, 50°C

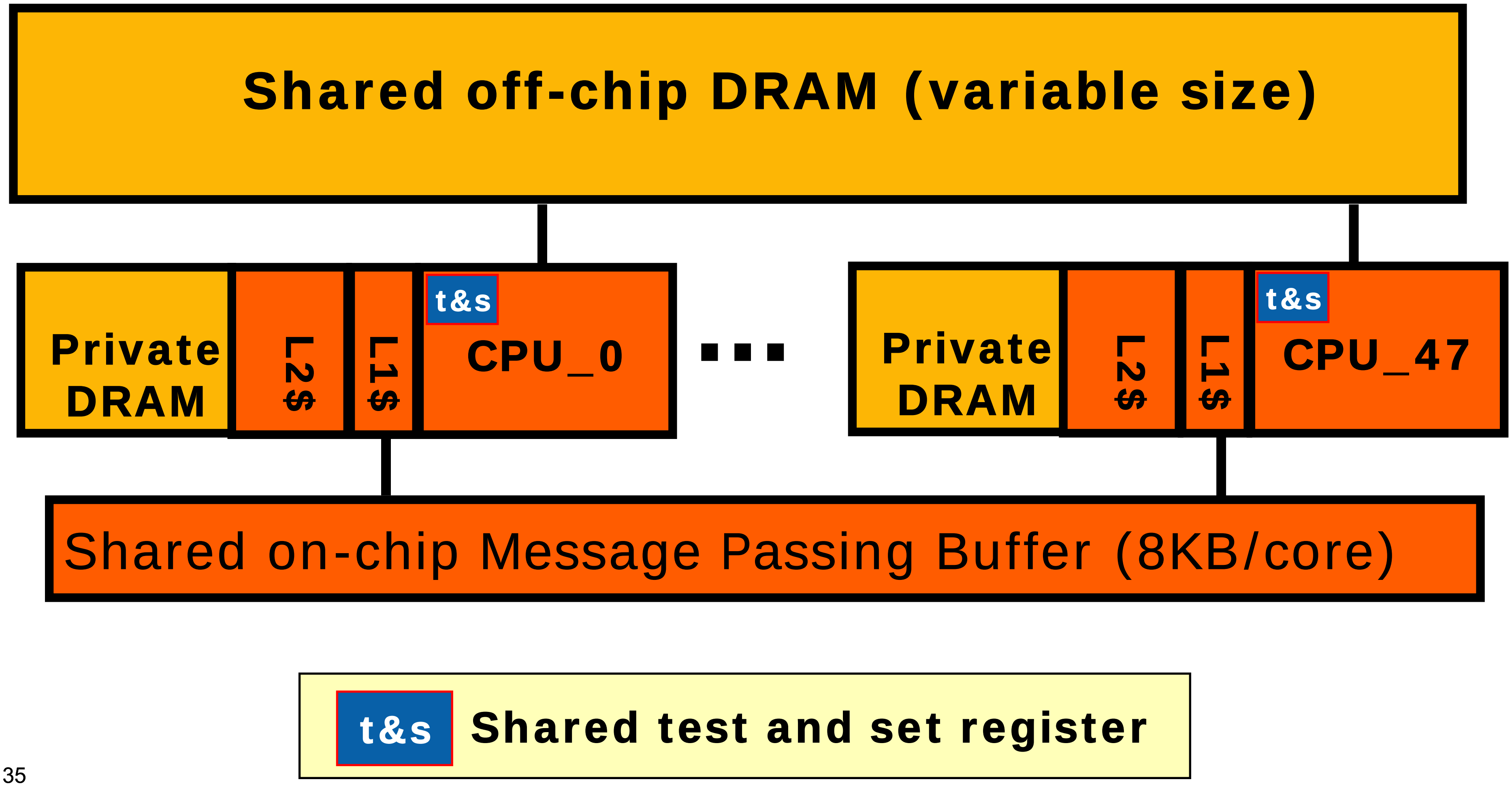


Clocking: 1.2W Routers: 1.2W
Cores: 5.1W MCs: 17.2W

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
  (/ means on/off-chip)



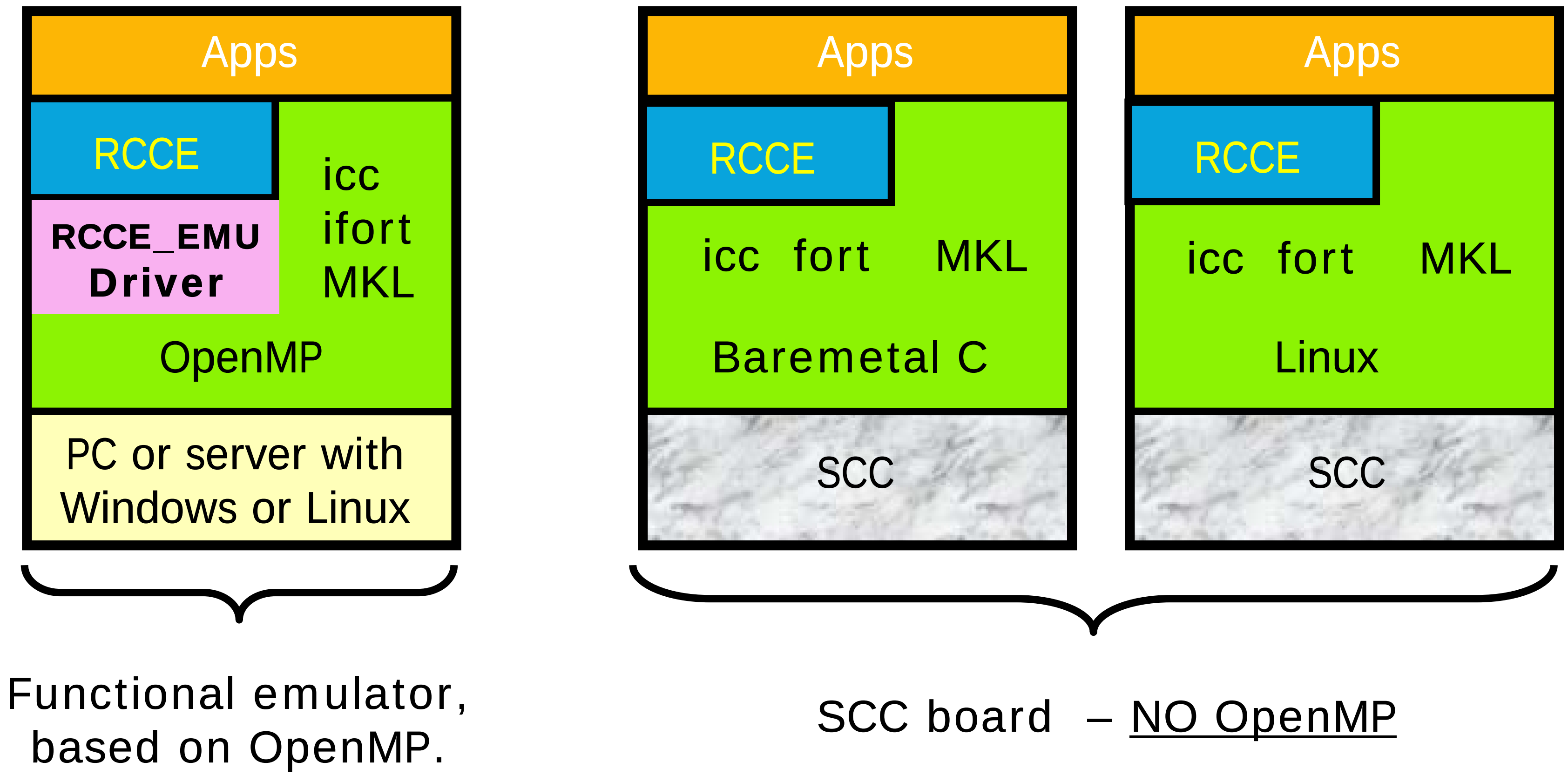
SCC Software research goals

- 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)



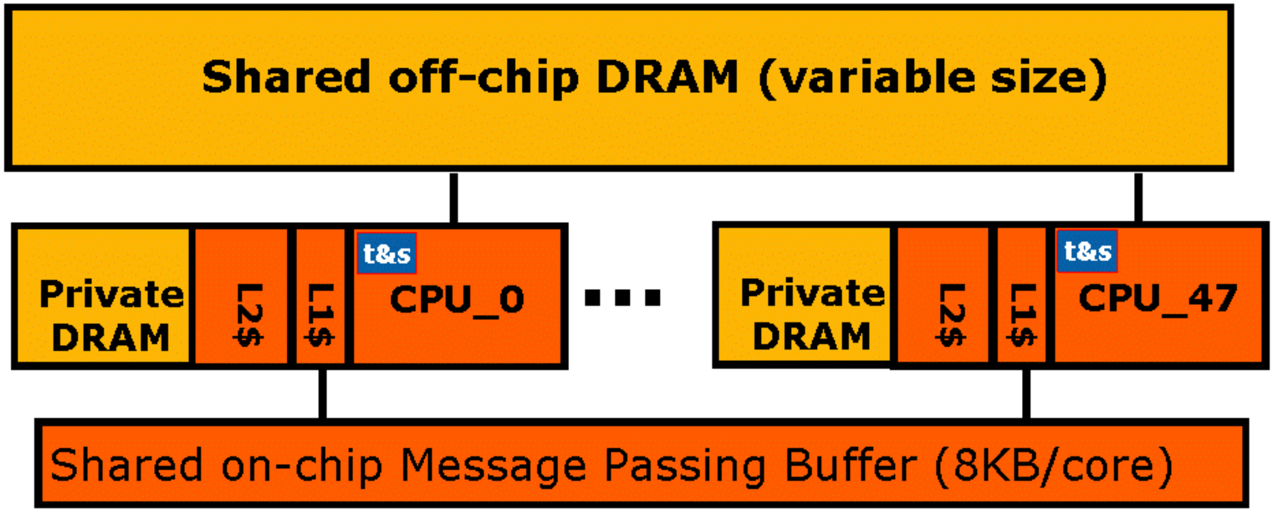
RCCE supports greatest common denominator between the three platforms

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



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

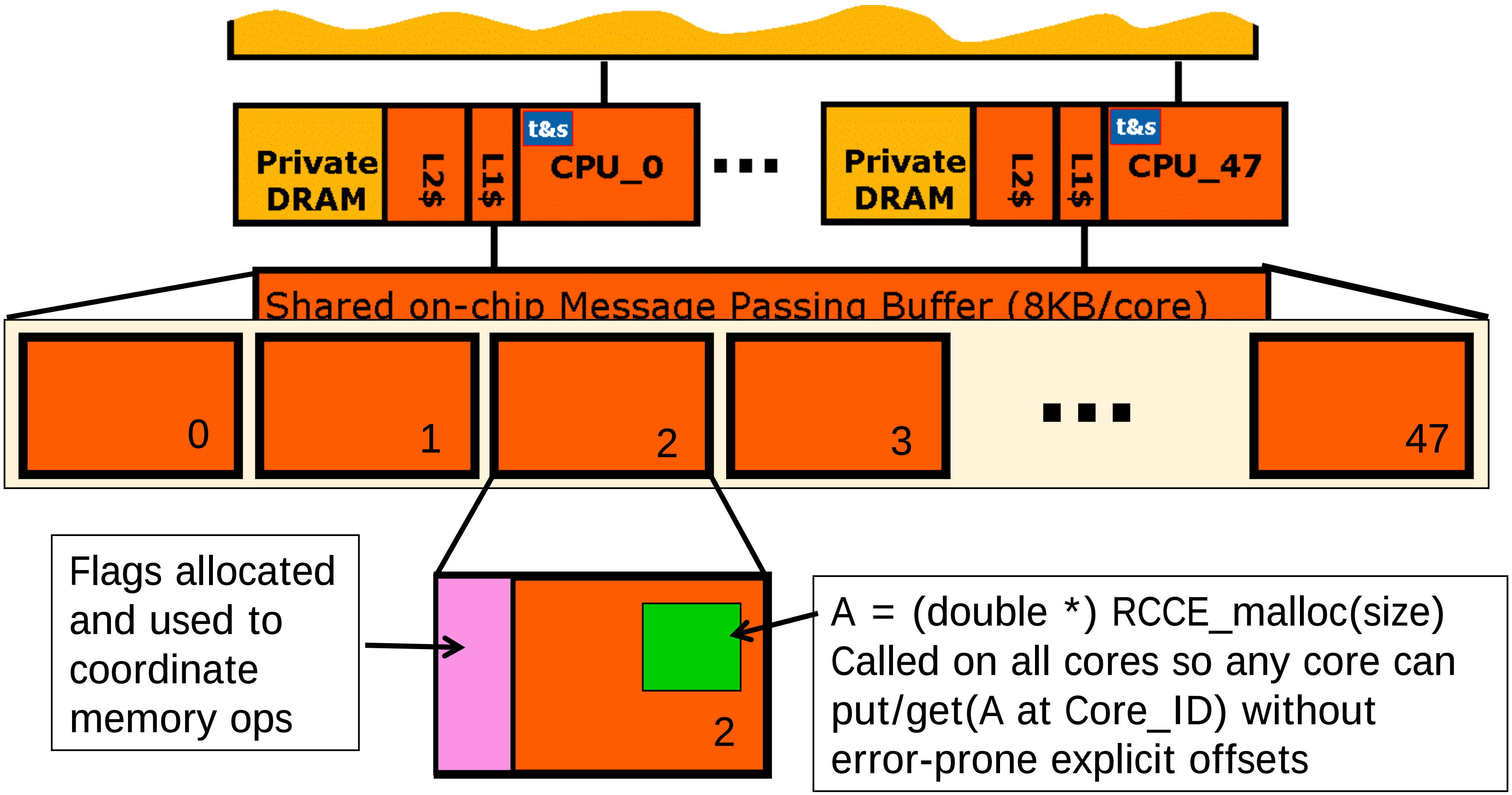
Consequences of MPBT properties:

- 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.**

Discourage user operations on data in MPB. Use only as a data movement area managed by RCCE … Invalidate early, invalidate often

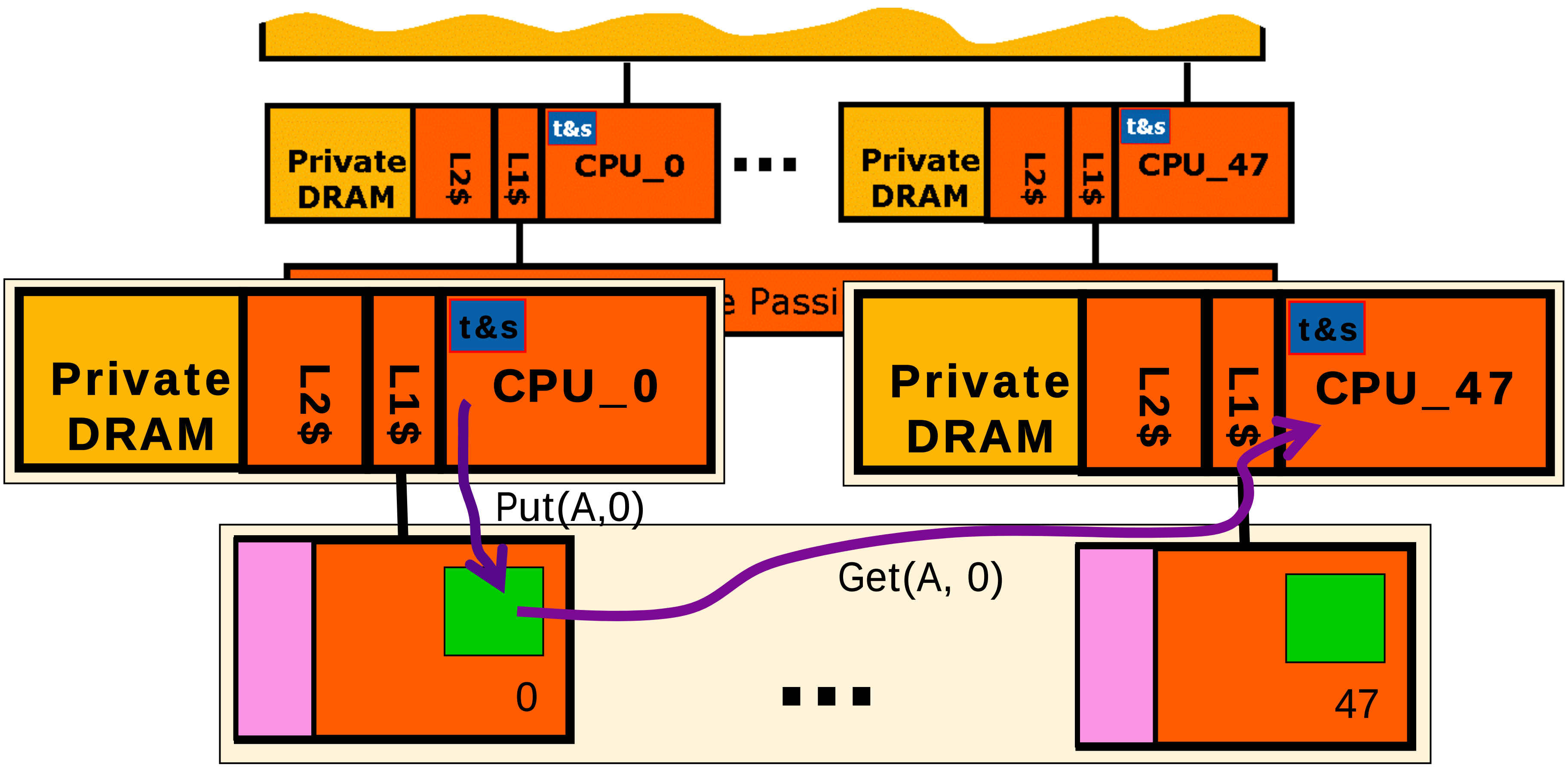
How does RCCE work? Part 2

- 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.



How does RCCE work? Part 3

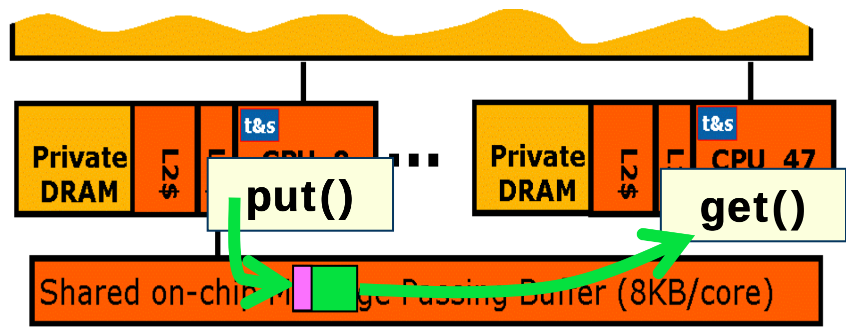
- The foundation of RCCE is a one-sided put/get interface.
- Symmetric name space … Allocate memory as a collective and put a variable with a given name into each core's MPB.



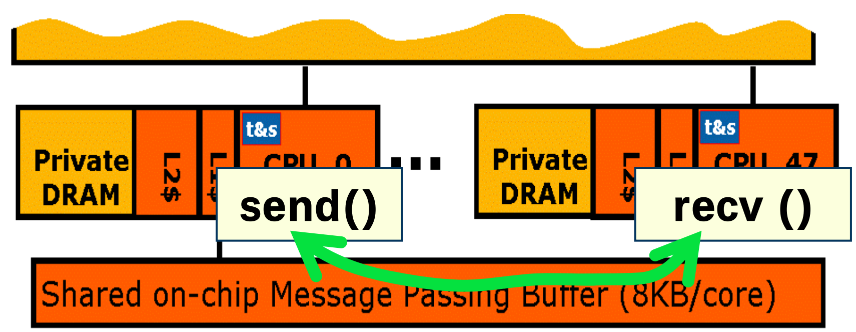
... and use flags to make the put's and get's "safe"

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

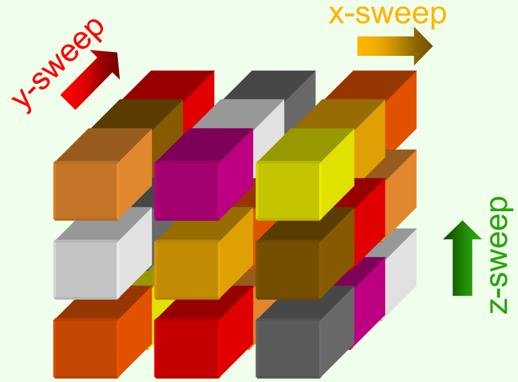


  - Two sided interface (send/recv) with most detail (flags and MPB management) hidden.
    - The "basic" interface for typical application programmers.



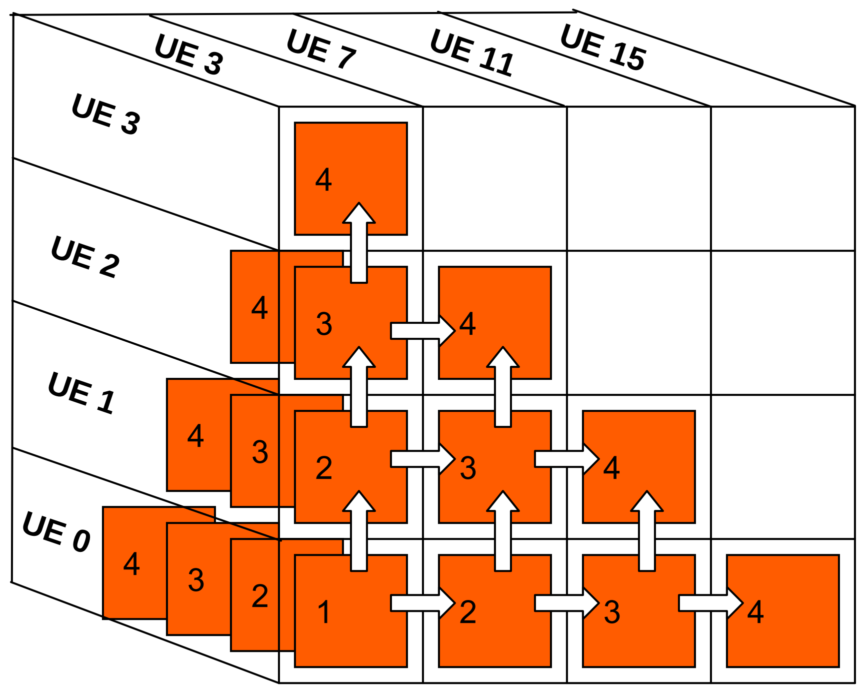
Linpack and NAS Parallel benchmarks

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

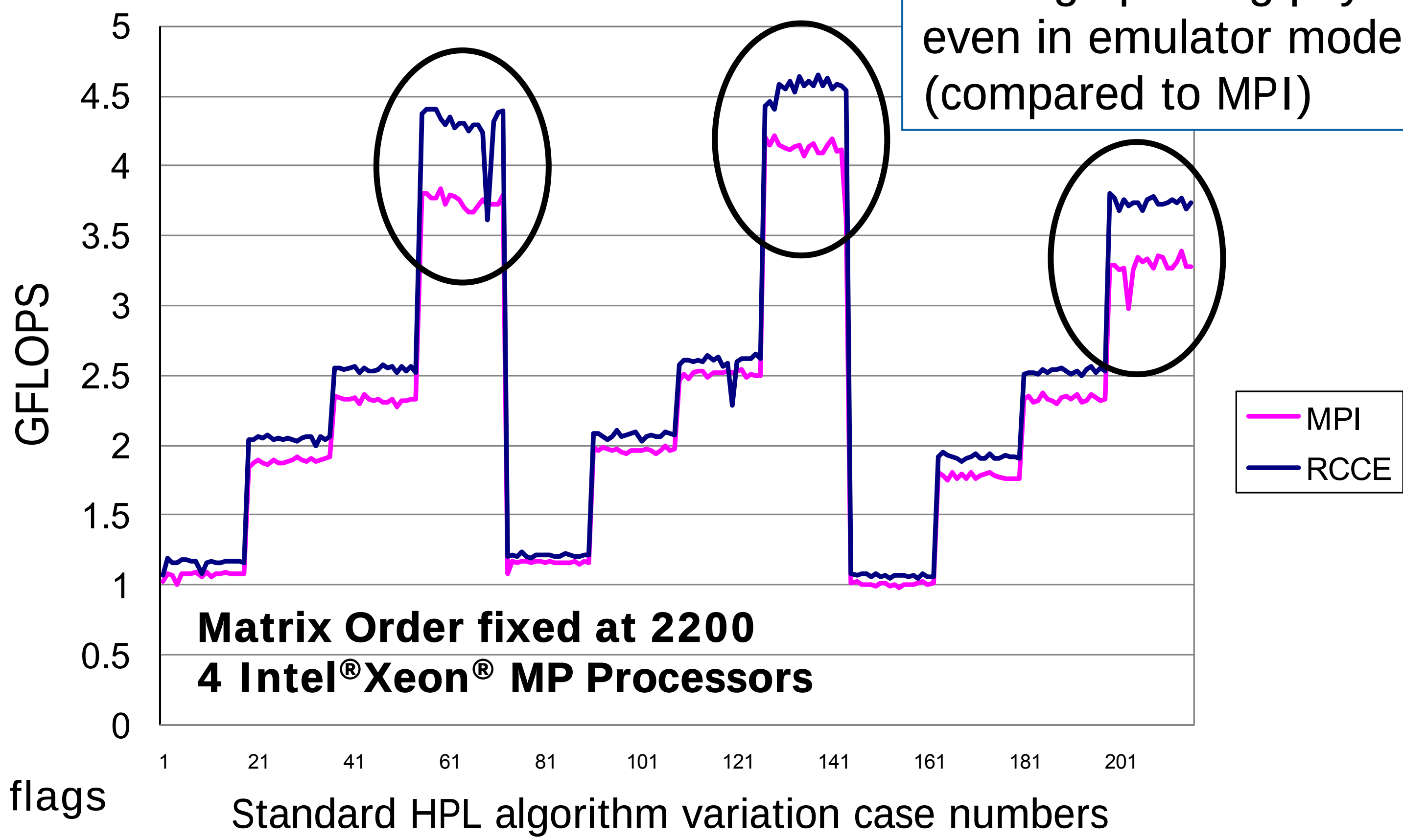
3. LU: Pencil decomposition
   – Define 2D-pipeline process.
     – await data (bottom+left)
     – compute new tile
     – send data (top+right)



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)



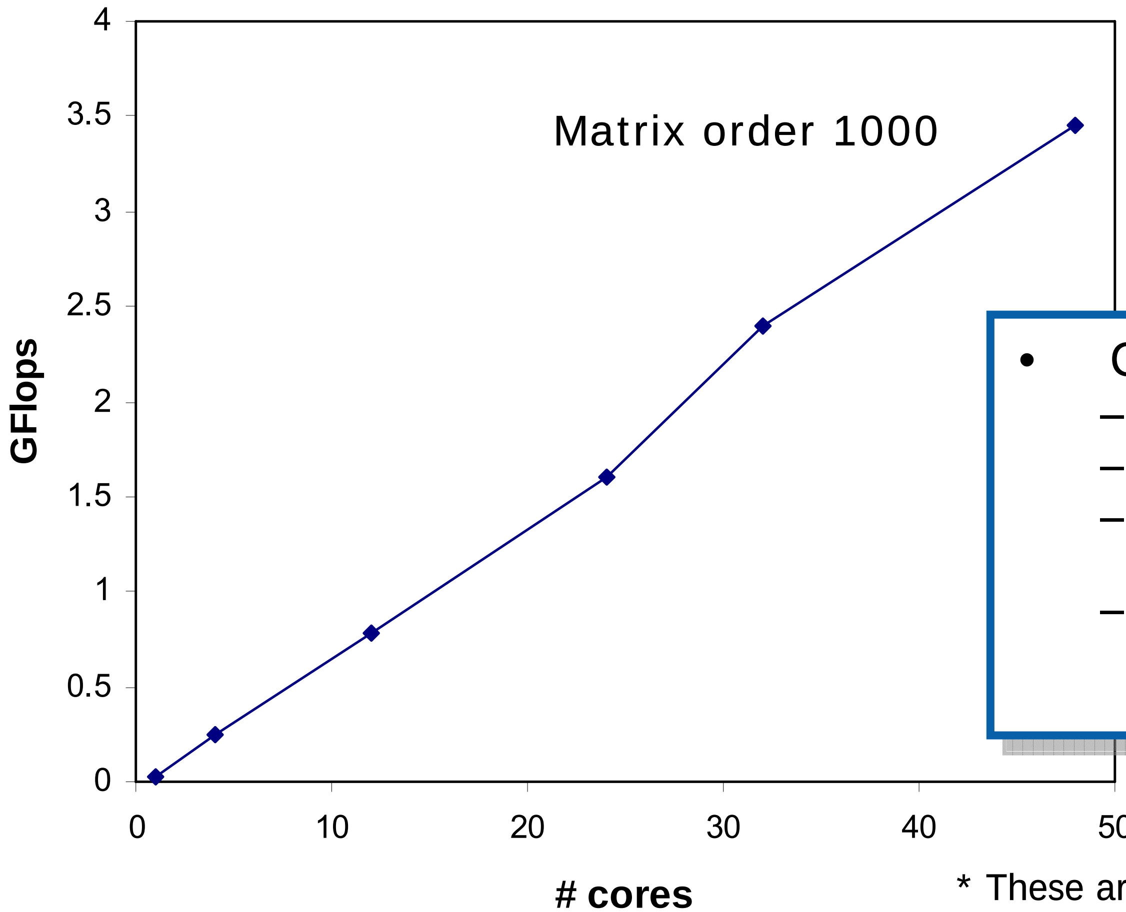
RCCE 1-bit flags

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

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.



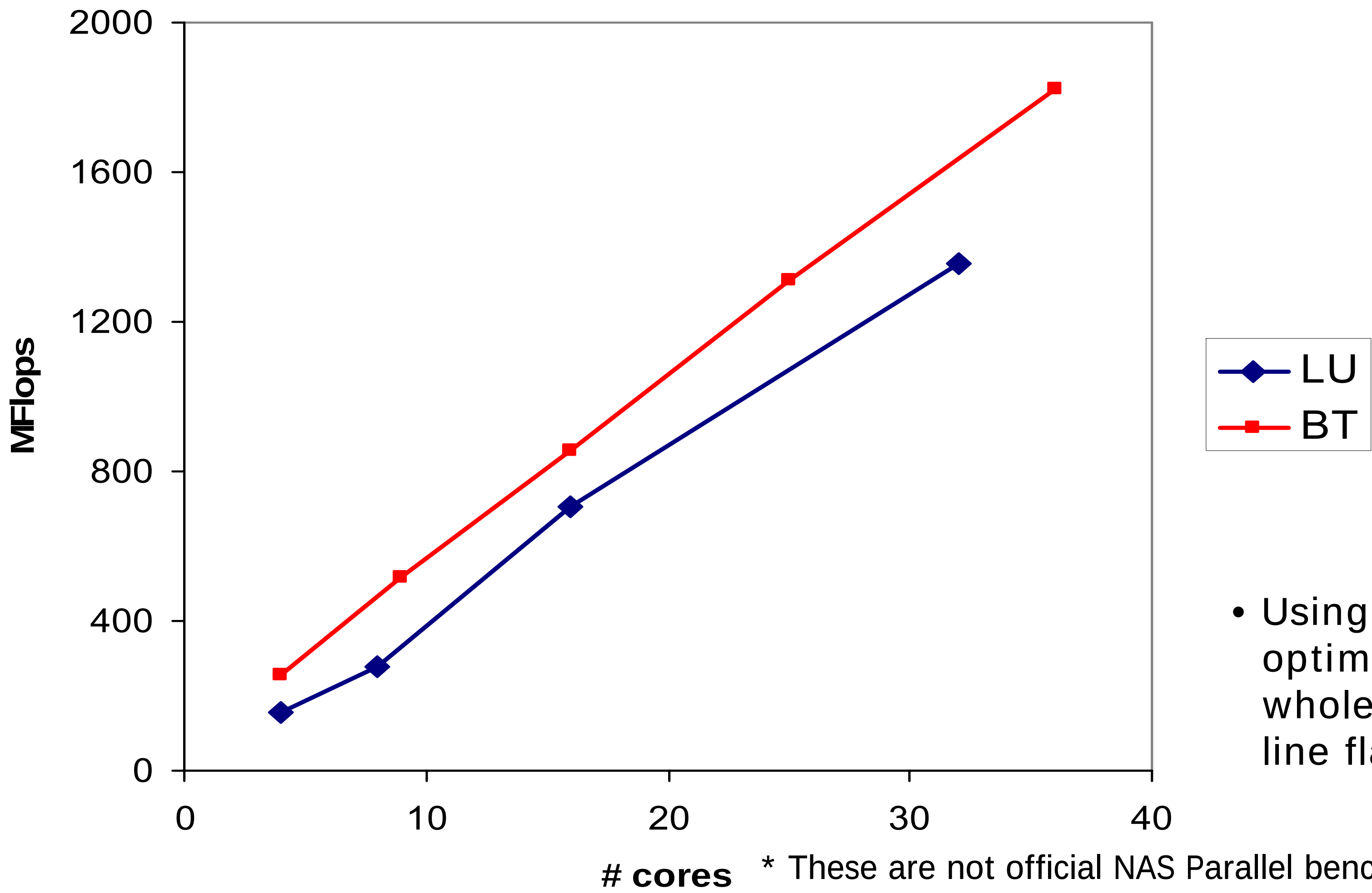
- Calculation Details:
  - Un-optimized C-BLAS
  - Un-optimized block size (4x4)
  - Used latency-optimized whole cache line flags
  - Performance dropped ~10% with memory optimized 1-bit flags

* These are not official LINPACK benchmark results.

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

LU/BT NAS Parallel Benchmarks, SCC

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

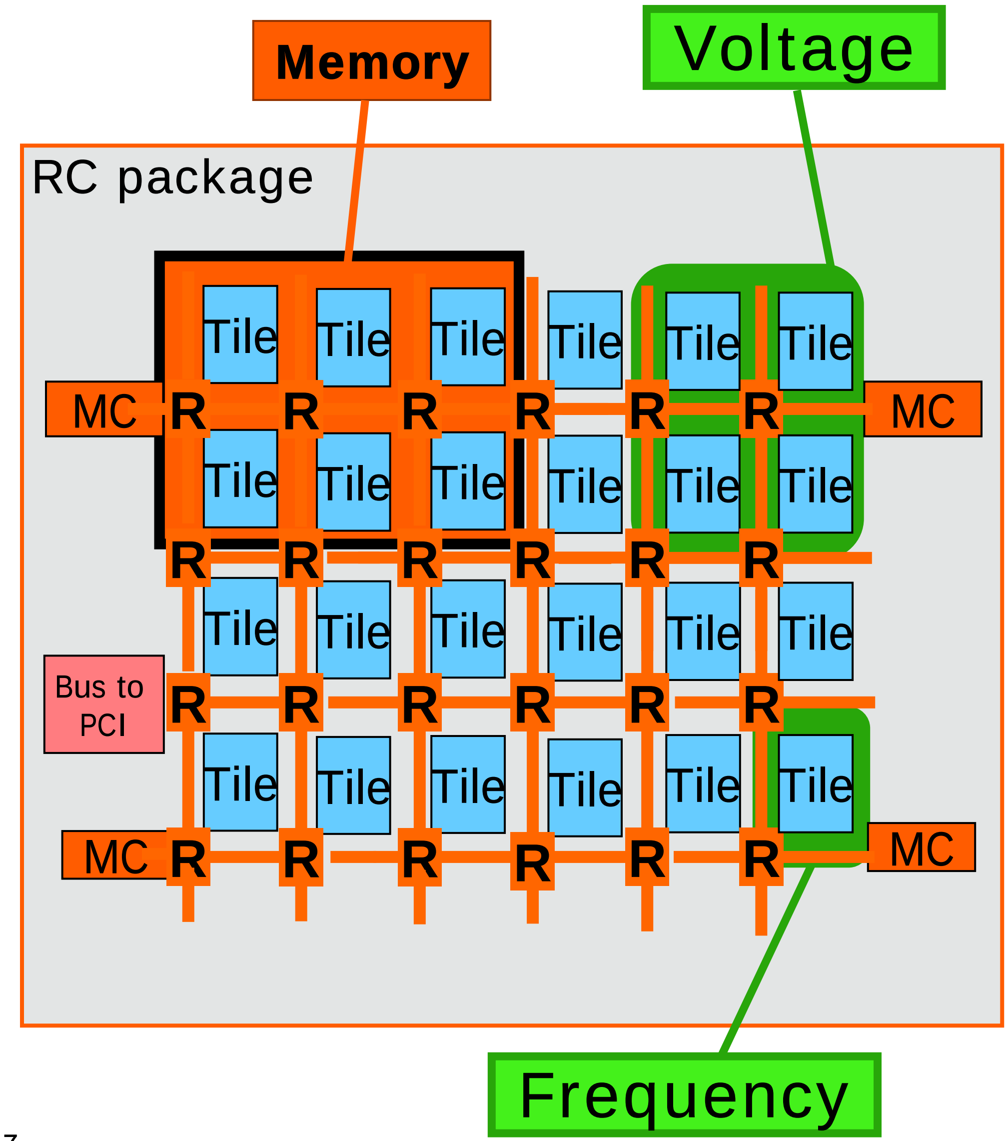


- Using latency optimized, whole cache line flags

* These are not official NAS Parallel benchmark results.

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

Power and memory-controller domains



Power ~ F V^2

–Power Control domains (RPC):

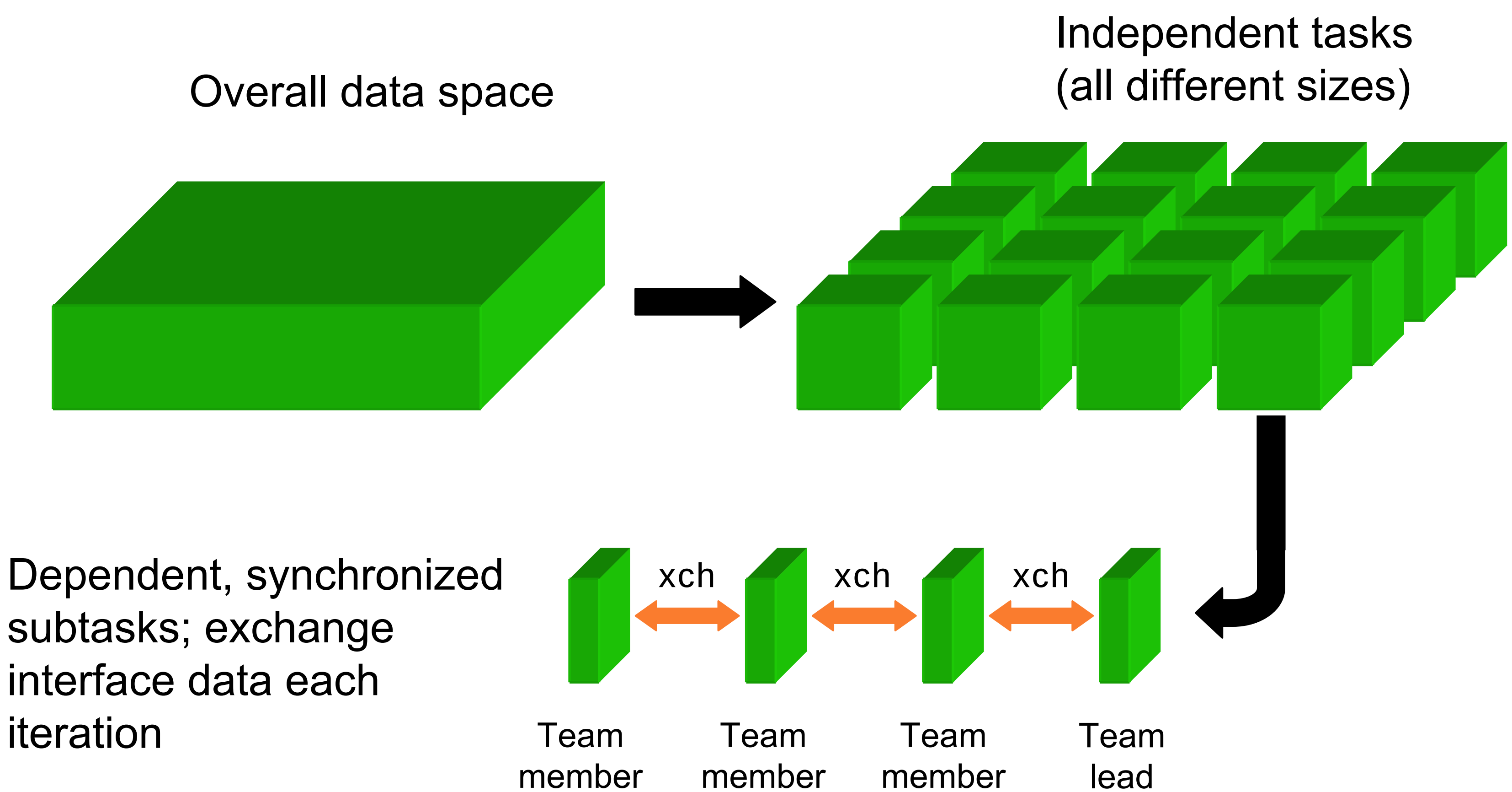
- –7 voltage domains … 6 4-tile blocks and one for on-die network.
- –1 clock divider register per tile (i.e. 24 frequency domains)
- –One RPC register so can process only one voltage request at a time; other requestors block

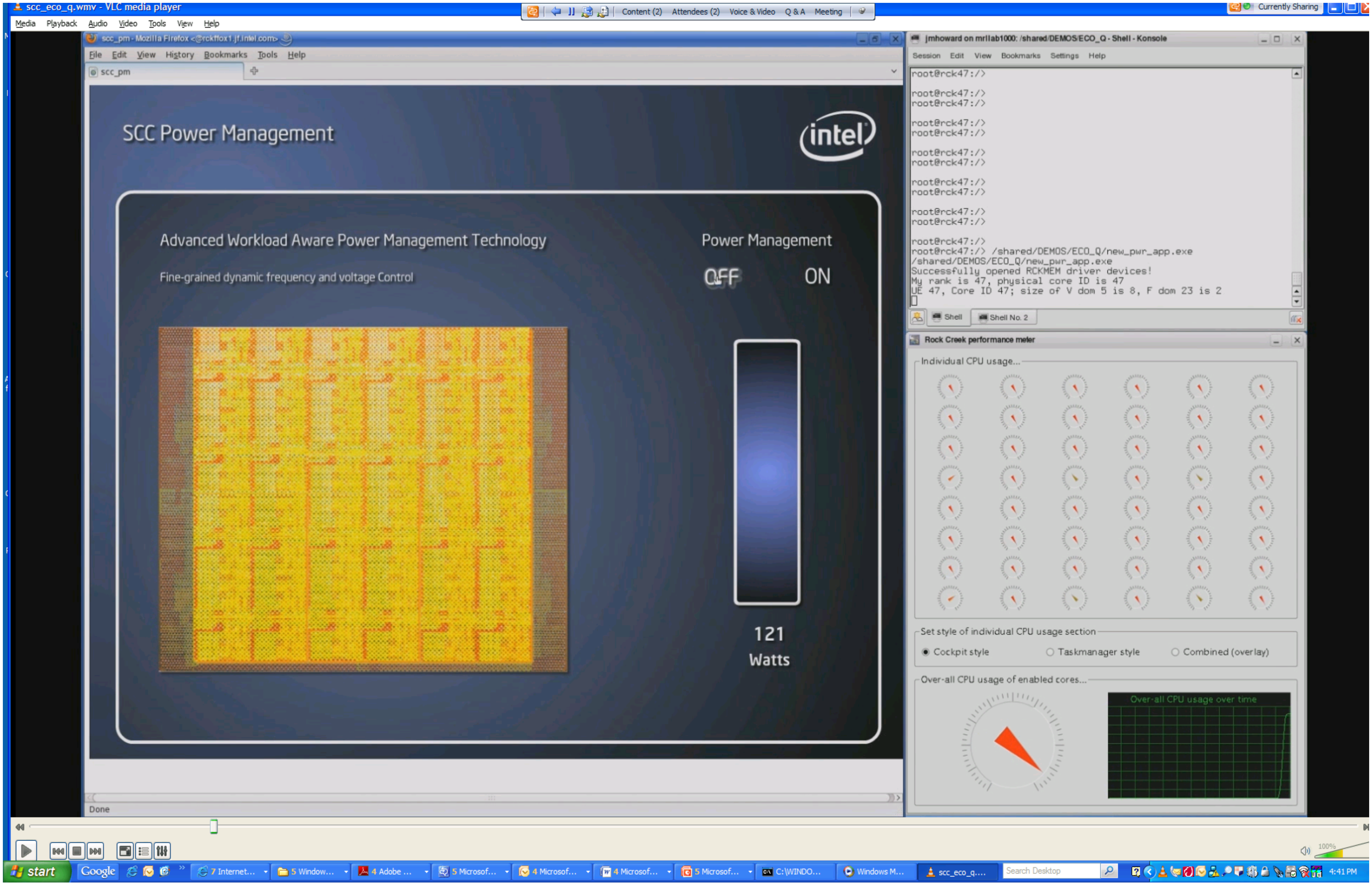
RCCE Power Management API

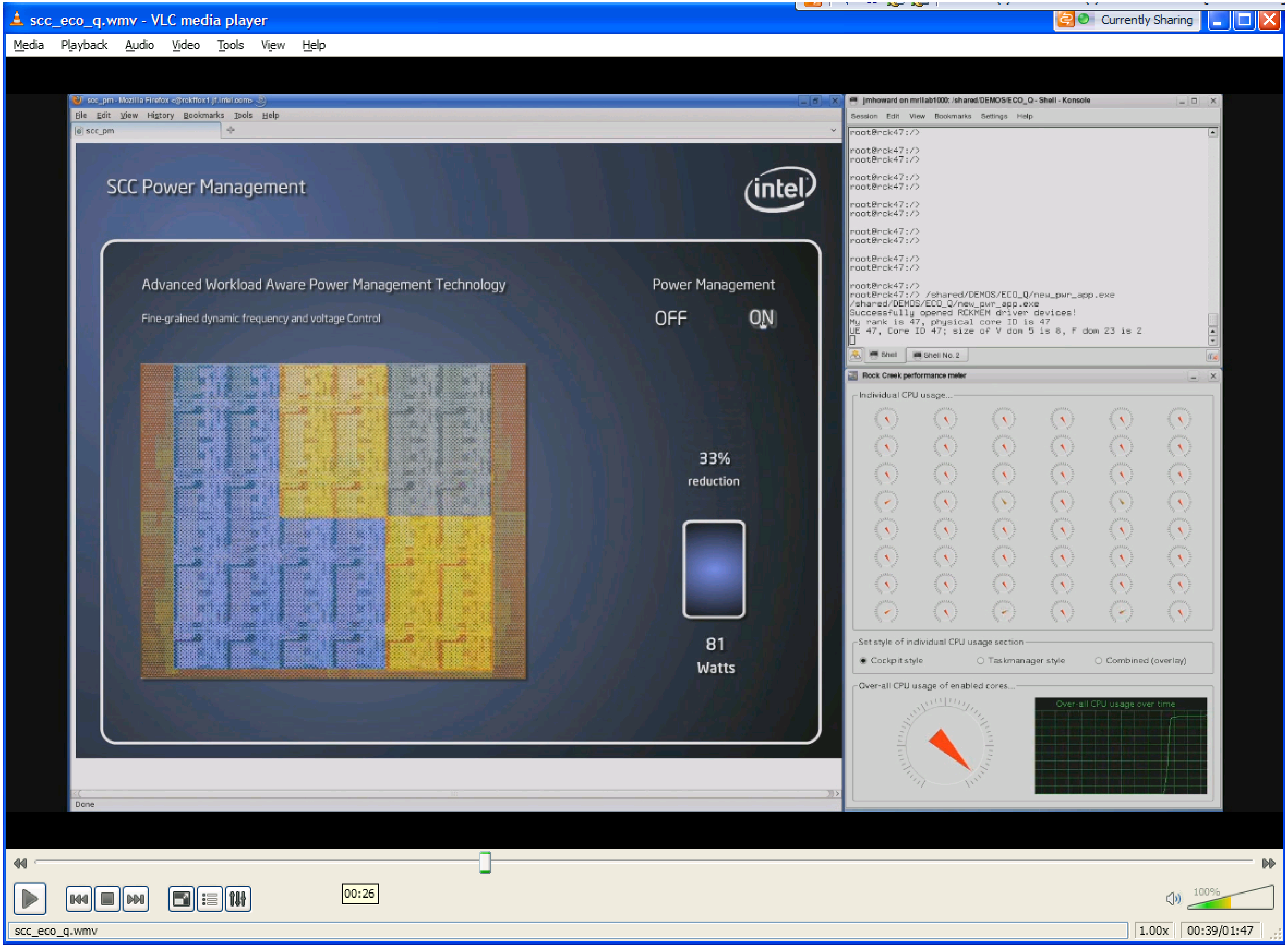
- RCCE power management emphasizes safe control: V/GHz changed together within each 4-tile (8-core) power domain.
  - A Master core sets V + GHz for all cores in domain.
    - RCCE_istep_power():
      - steps up or down V + GHz, where GHz is max for selected voltage.
    - RCCE_wait_power():
      - returns when power change is done
    - RCCE_step_frequency():
      - steps up or down only GHz
- Power management latencies
  - V changes: Very high latency, *O*(Million) cycles.
  - GHz changes: Low latency, *O*(few) cycles.

Power management test

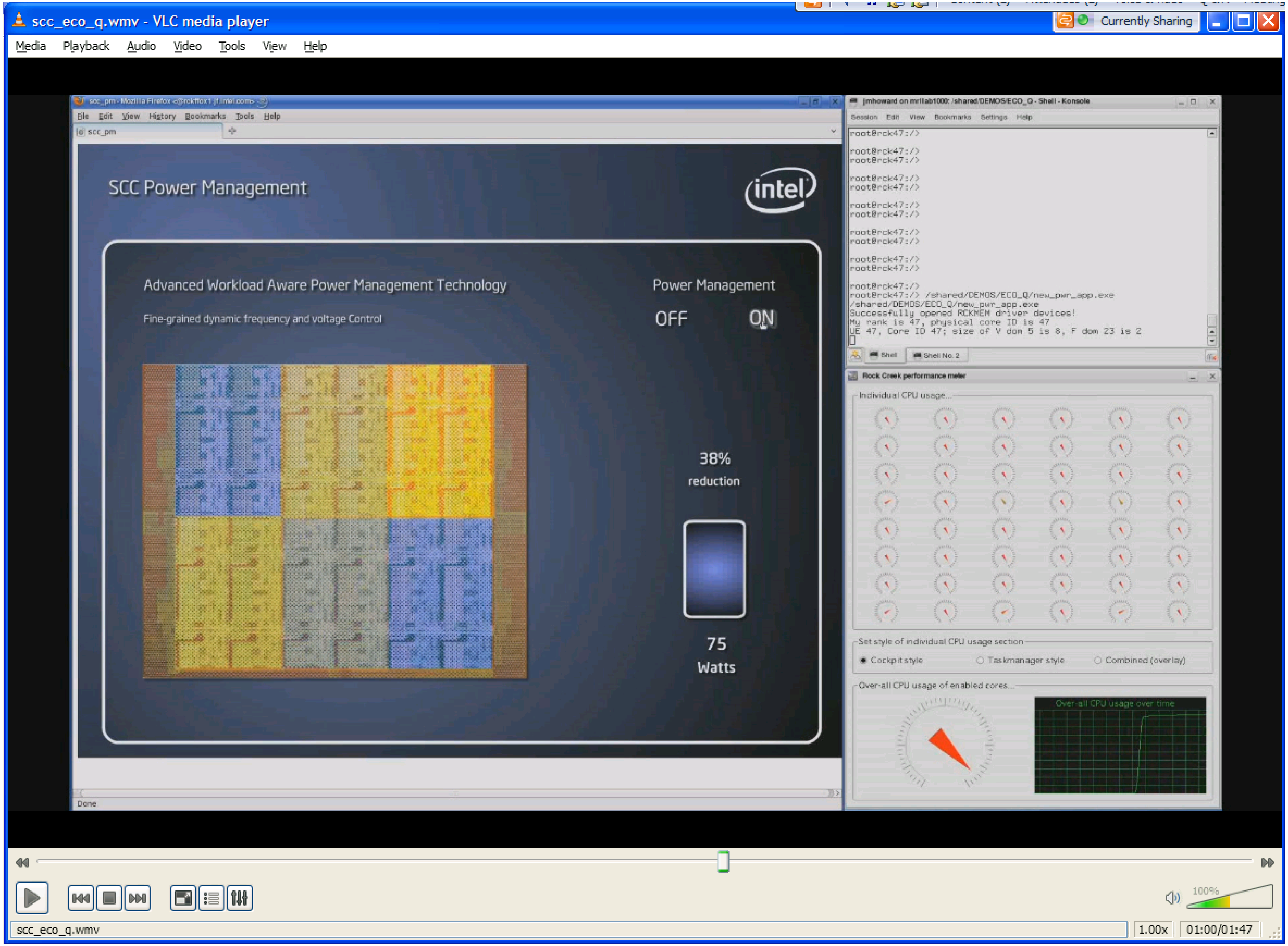
- A three-tier master-worker hierarchy,
  - one overall master, one team-lead per power domain, Team-members (cores) to do the work.
- Workload: A stencil computation to solve a PDE.



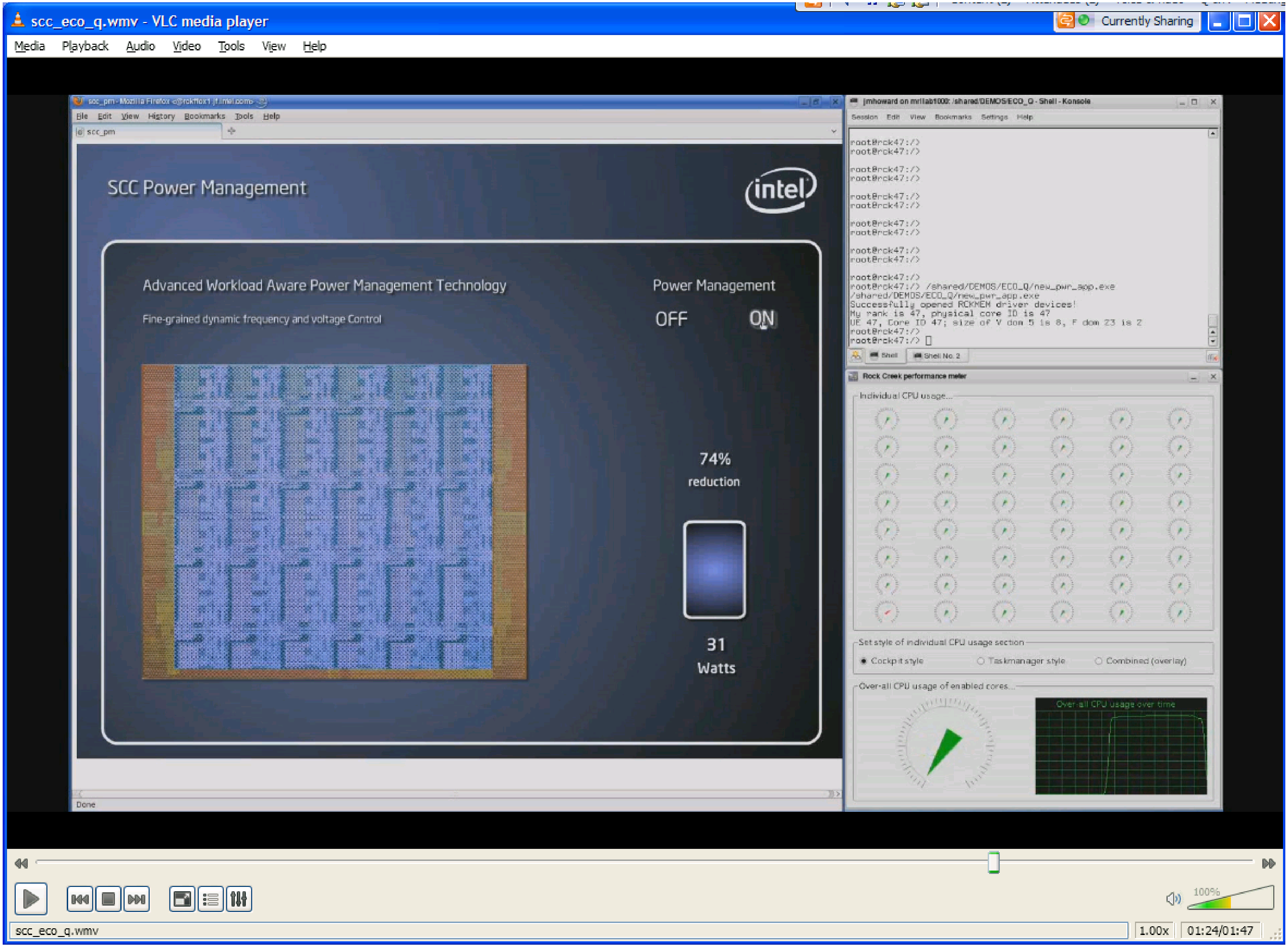

scc_eco_q.wmv - VLC media player
scc_pm - Mozilla Firefox
SCC Power Management
Advanced Workload Aware Power Management Technology
Fine-grained dynamic frequency and voltage Control
Power Management
OFF
ON
121
Watts
jmhoward on mrllab1000: /shared/DEMOS/ECO_Q - Shell - Konsole
root@rck47:/> /shared/DEMOS/ECO_Q/new_pwr_app.exe
/shared/DEMOS/ECO_Q/new_pwr_app.exe
Successfully opened RCKMEM driver devices!
My rank is 47, physical core ID is 47
UE 47, Core ID 47; size of V dom 5 is 8, F dom 23 is 2
Rock Creek performance meter
Individual CPU usage...
Set style of individual CPU usage section
Cockpit style
Taskmanager style
Combined (overlay)
Over-all CPU usage of enabled cores...
Over-all CPU usage over time



scc_eco_q.wmv - VLC media player
Currently Sharing
Media Playback Audio Video Tools View Help
SCC Power Management
intel
Advanced Workload Aware Power Management Technology
Fine-grained dynamic frequency and voltage Control
Power Management
OFF
ON
33%
reduction
81
Watts
Done
root@rck47:/> /shared/DEMOS/ECO_Q/new_pwr_app.exe
Successfully opened RCKMEM driver devices!
Rock Creek performance meter
Individual CPU usage...
Set style of individual CPU usage section
Cockpit style
Taskmanager style
Combined (overlay)
Over-all CPU usage of enabled cores...
Over-all CPU usage over time
00:26
scc_eco_q.wmv
1.00x
00:39/01:47
100%



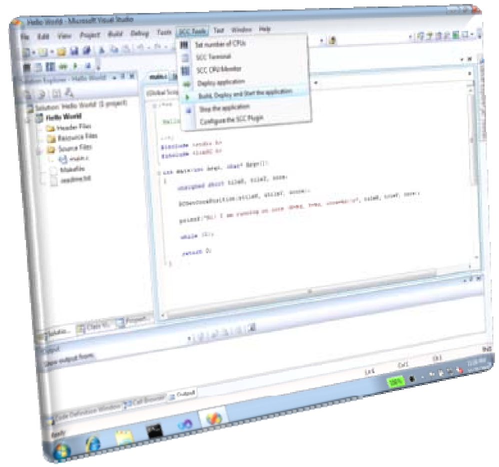
scc_eco_q.wmv - VLC media player
Currently Sharing
Media Playback Audio Video Tools View Help
SCC Power Management
intel
Advanced Workload Aware Power Management Technology
Fine-grained dynamic frequency and voltage Control
Power Management
OFF ON
38%
reduction
75
Watts
root@rck47:/> /shared/DEMOS/ECO_Q/new_pwr_app.exe
/shared/DEMOS/ECO_Q/new_pwr_app.exe
Successfully opened RCKMEM driver devices!
My rank is 47, physical core ID is 47
UE 47, Core ID 47; size of V dom 5 is 8, F dom 23 is 2
Rock Creek performance meter
Individual CPU usage
Set style of individual CPU usage section
Cockpit style
Taskmanager style
Combined (overlay)
Over-all CPU usage of enabled cores
Over-all CPU usage over time
scc_eco_q.wmv
1.00x
01:00/01:47


scc_eco_q.wmv - VLC media player
Currently Sharing
Media
Playback
Audio
Video
Tools
View
Help
SCC Power Management
intel
Advanced Workload Aware Power Management Technology
Fine-grained dynamic frequency and voltage Control
Power Management
OFF
ON
74%
reduction
31
Watts
Done
Rock Creek performance meter
Individual CPU usage
Set style of individual CPU usage section
Cockpit style
Taskmanager style
Combined (overlay)
Over-all CPU usage of enabled cores
Over-all CPU usage over time
root@rck47:/> /shared/DEMOS/ECO_Q/new_pwr_app.exe
/shared/DEMOS/ECO_Q/new_pwr_app.exe
Successfully opened RCKMEM driver devices!
My rank is 47, physical core ID is 47
UE 47, Core ID 47; size of V dom 5 is 8, F dom 23 is 2
root@rck47:/>
100%
scc_eco_q.wmv
1.00x
01:24/01:47

SCC Demo Showcase

Financial Analytics
w/ shared virtual memory

Microsoft Visual Studio

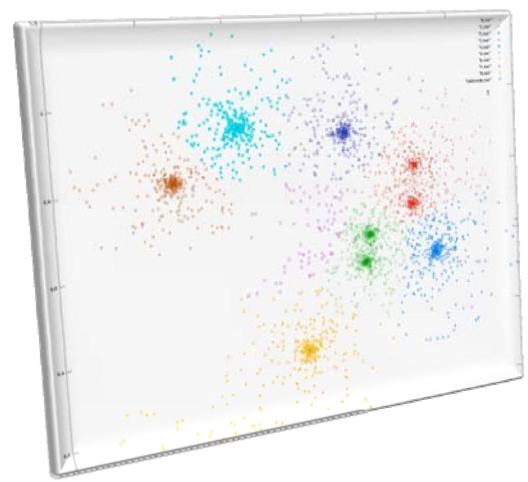


Advanced Power Management

JavaScript Physics Modeling

HPC Parallel Workloads

Hadoop Web Search

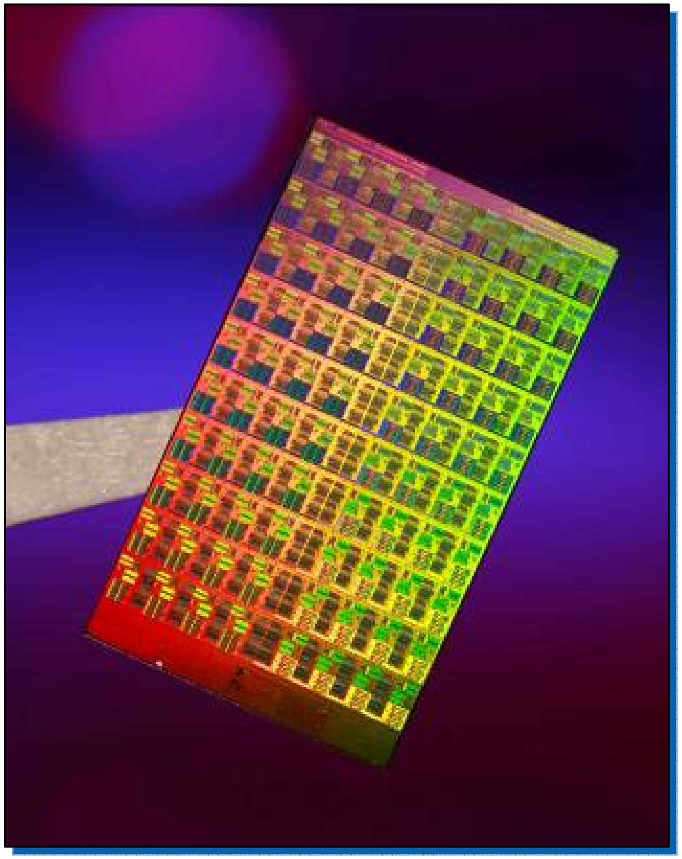


Conclusions

- 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.

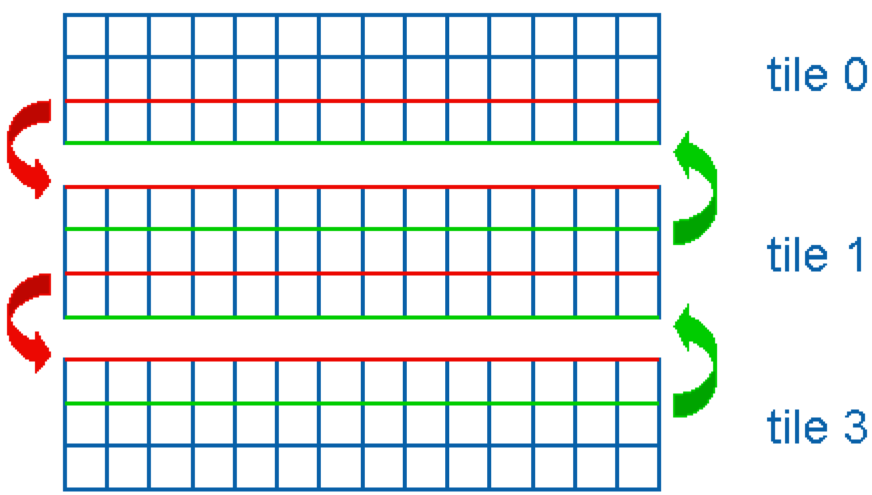
Backup Slides

- Details on 80 core processor application kernels
- More on RCCE



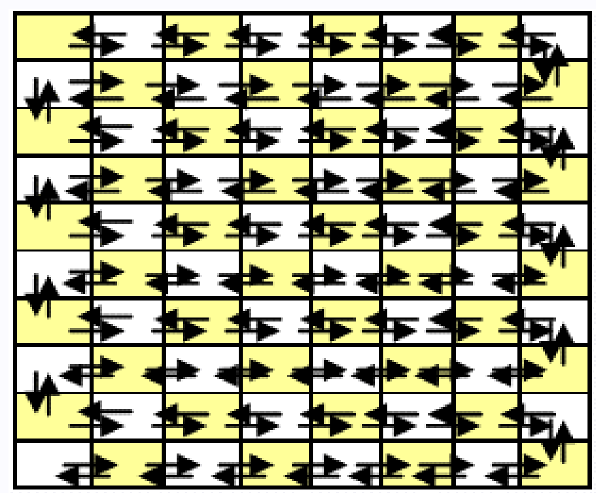
Stencil

- 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

M=2240, N= 16



Communication Pattern

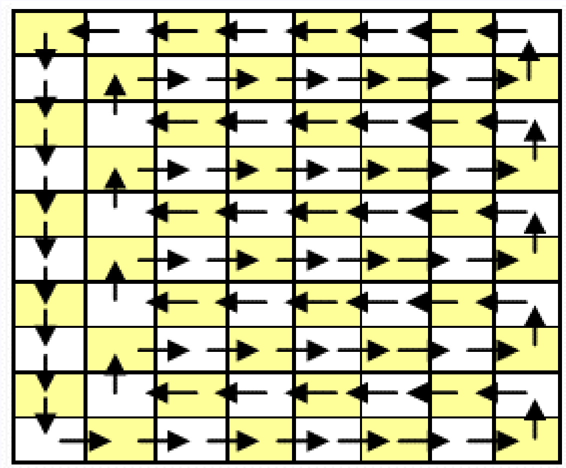
SGEMM

- 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
{
  C_ij = dot_product (row A_i * column B_j)
  Circular shift column B_j to neighbor
}
```

$C(N,N) = A(N,M)*B(M,N)$

$N = 80,\ M = 206$



Communication Pattern

- 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

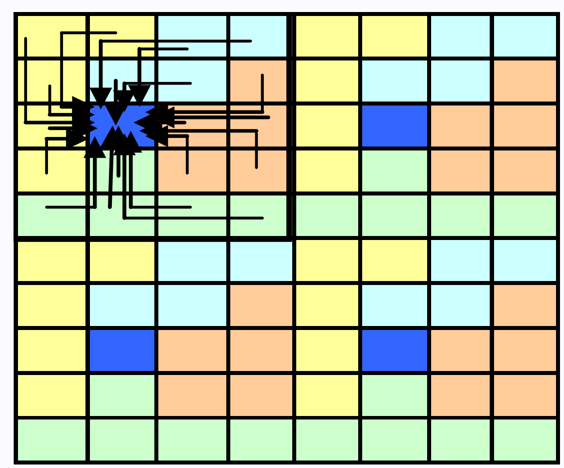
Spreadsheet

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

Linearize array indices

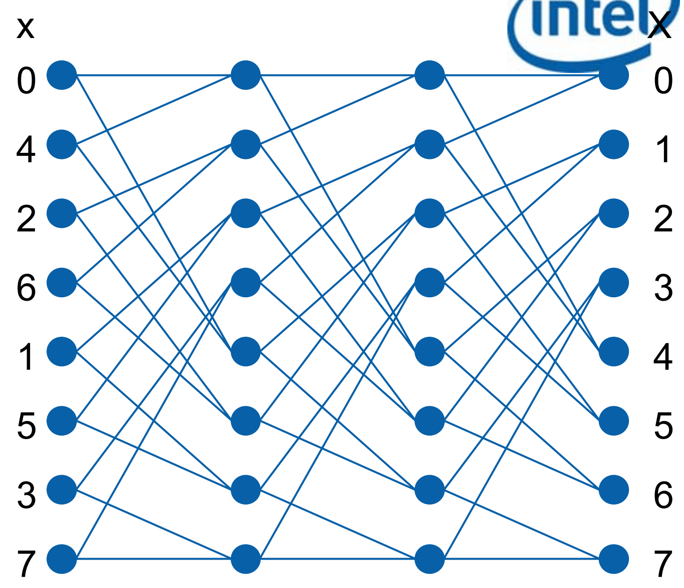
- Column sums local to a tile.
- Row sums required a vector reduction across all rows.
- We processed many spread sheets at once so we could pipeline reductions to manage latencies.
- 76 cores did local csum and passed results to one of four accumulator nodes.
- The four nodes combined results to get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

LxN table of value/weight pairs. N = 10, L = 1600

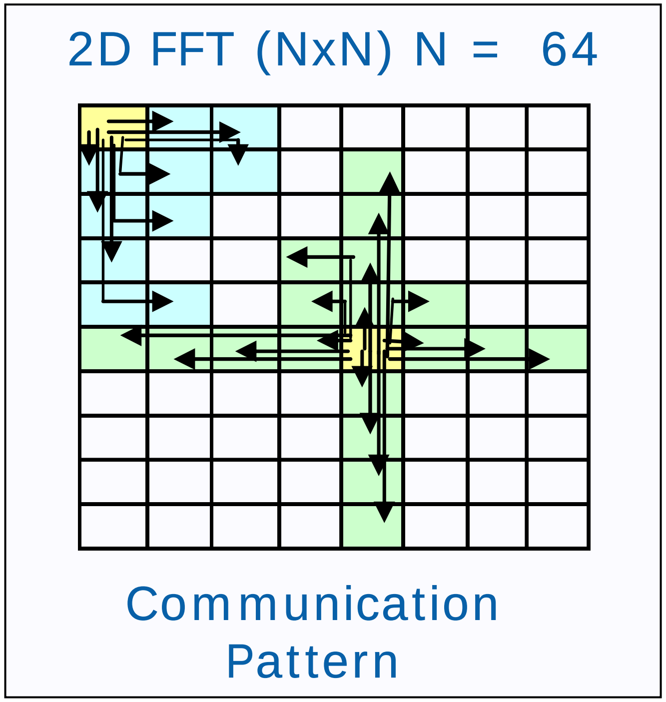


Communication Pattern

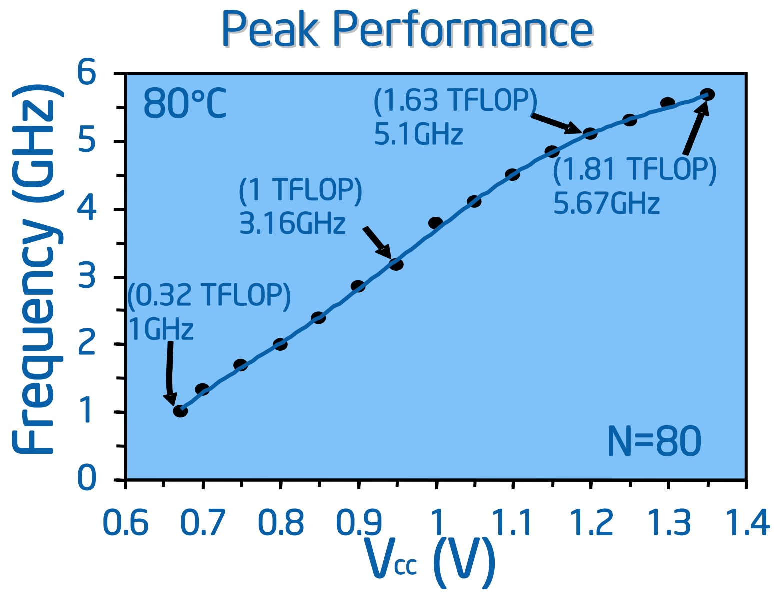
2D FFT

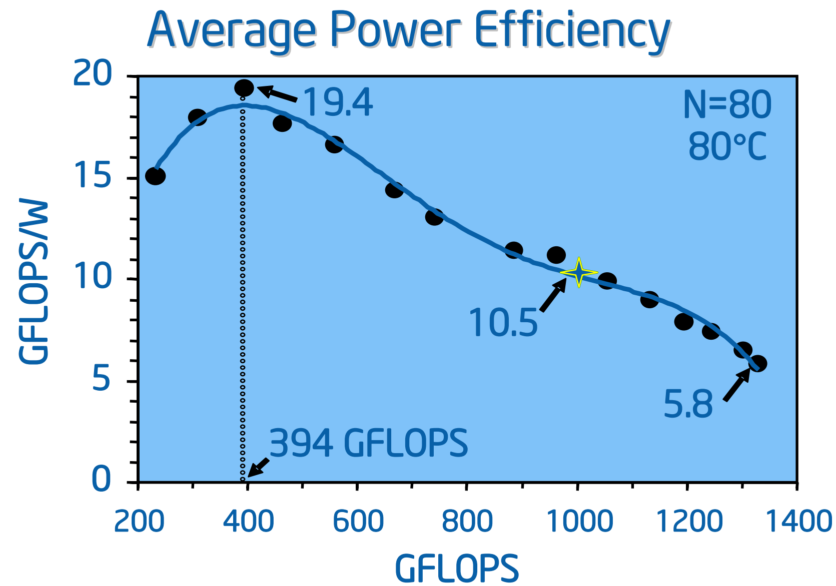


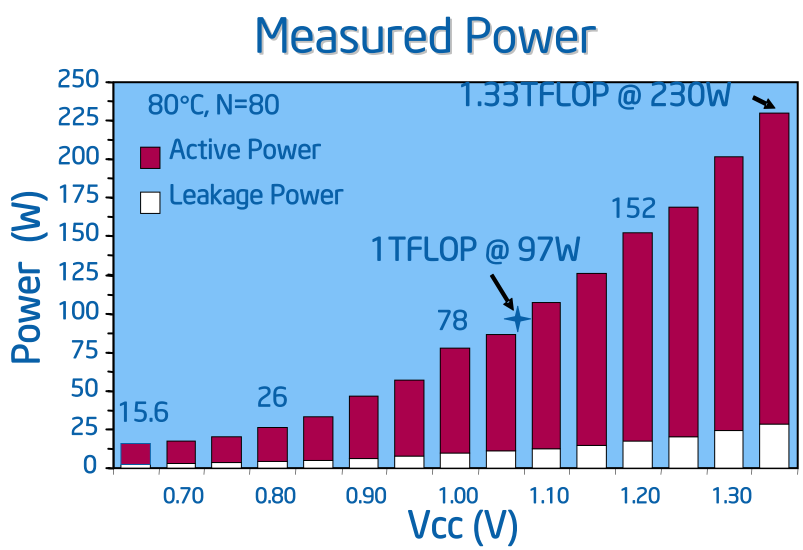
- 64 Point 2D FFT on an 8 by 8 Grid.
- Pease Algorithm
  - “Peers” in each phase are constant … a constant communication pattern throughout the computation.
- Parallelization:
  - Basic operation FFT of 64 long vector along a column of 8 tiles
    - FFT of 8-long vector in each tile
    - Communication:
      - 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

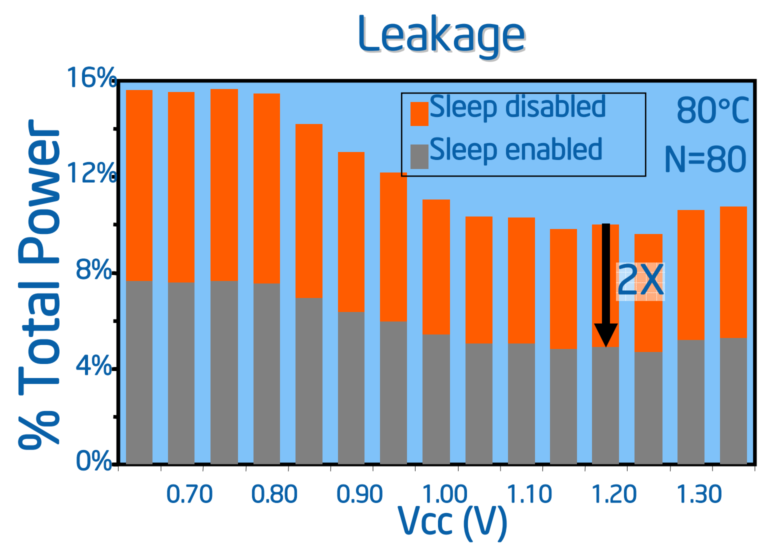


Power Performance Results







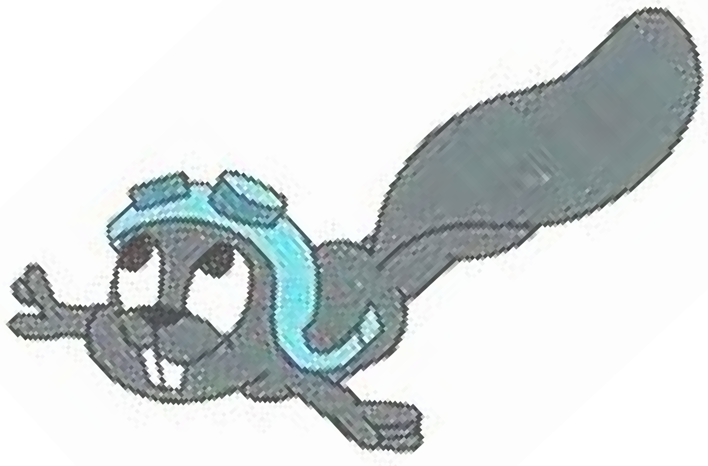


Stencil: 1TFLOP @ 97W, 1.07V;

All tiles awake/asleep

Backup Slides

- Details on 80 core processor application kernels.
- More on RCCE



R C C E

A small library for many-core communication

Rob van der Wijngaart (Software and Services Group)

Tim Mattson (Intel Labs)

Rapidly	Communicating	Cores	Env.
Reduced	Compact	Communication	Environment
Research	Cores	Communication	Environment
Rabble-of	Communicating	Cores	Experiments
Rock	Creek	Communication	Environment
Radically	Cool	Coordination	E-science
Richly	Communicating	Cores	Ecosystem
Restricted	Capability	Communication	Environment
Rorschach	Core	Communication	Express

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 UEs 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

```
#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

RCCE API: Circular Shift one-sided

```
#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 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

```
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);
```

- 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:

RCCE_send(buffer, size, ID); *Send private memory (buffer) to core ID*
RCCE_recv(buffer, size, ID)); *Receive into private memory (buffer) from core ID*

- This is Synchronous message passing … the send and receive do not return until the communication is complete on both sides.

RCCE API: Circular Shift with 2-sided Basic interface

```
#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);
}
```

Hides buffer and flag allocation, messages "packetizing", and flag synchronization.

Anticipate most programmers will use this RCCE version

BUFSIZE may be anything
Message need not fit inside Msg Buf

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);
}
```

offsets to “remote” MPB

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);}
```

HANDSHAKES

sent, ready: synchronization flags stored in MPB

```
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)

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

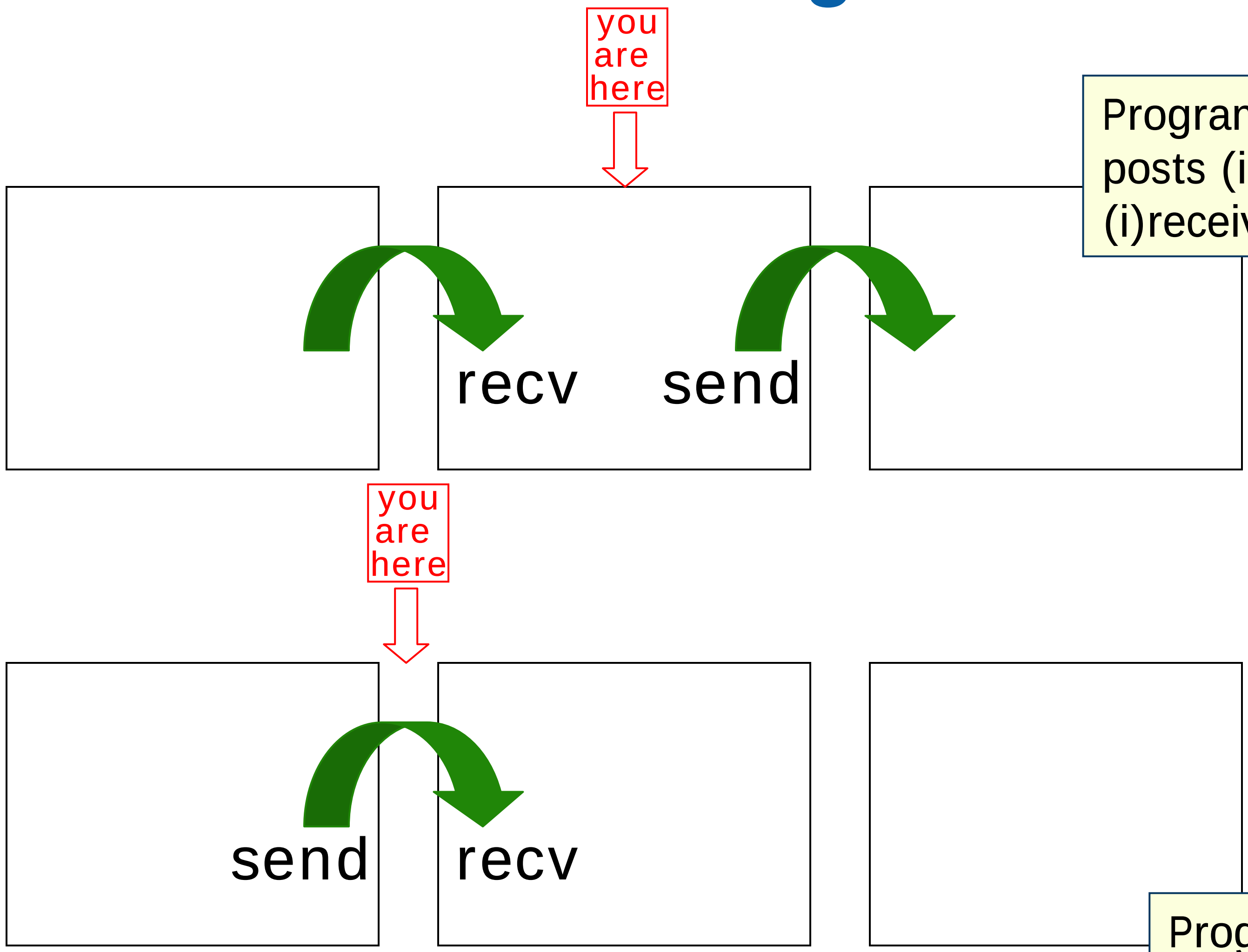
- No opaque data types in RCCE, so no MPI-style handles, only pointers
- No RCCE_datatype, except for reductions
- No communicators, except in collective communications
- Only synchronous communications
  - + No message bookkeeping
  - – No overlap of computations/communications
  - – Deadlock?
- RCCE 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

MPI

RCCE

Programmer must pair all sends and receives

RCCE vs MPI: Avoiding deadlock

– pseudo-code example from HPC application:

```
MPI:  if (!IAM_LEFTMOST) {
          MPI_irecv(from_left);
          MPI_wait(on_isend);
          MPI_wait(on_irecv);
      }
      compute;

      if (!IAM_RIGHTMOST) MPI_isend(to_right);
```

```
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