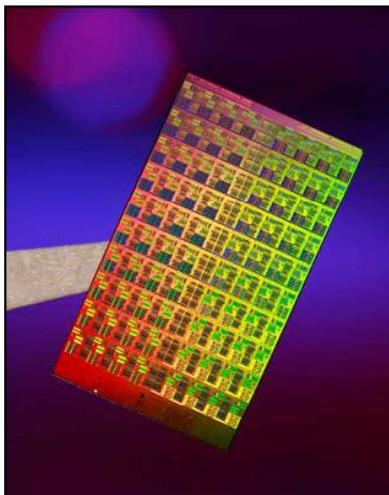


The Future of Many Core Computing: A tale of two processors



IT WAS THE BEST of times, it was the worst of times,
it was the age of wisdom, it was the age of foolishness,
it was the epoch of belief, it was the epoch of incredulity,
it was the season of Light, it was the season of Darkness,
it was the spring of hope, it was the winter of despair,
we had everything before us, we had nothing before us,
we were all going direct to Heaven, we were all going
direct the other way—in short, the period
like the present period, that some of its noisiest
authorities insisted on its being received, for good
or for evil, in the superlative degree of comparison only.

Tim Mattson
Intel Labs

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



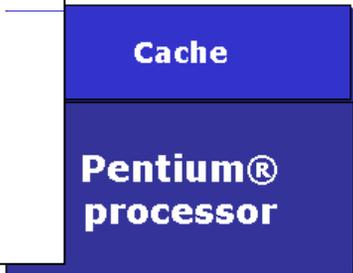
What the Cores will look like:
From a few large cores to many lightweight cores

Optimized for speed

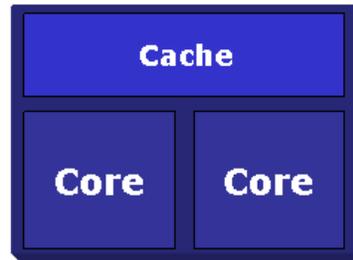
Optimized for performance/watt



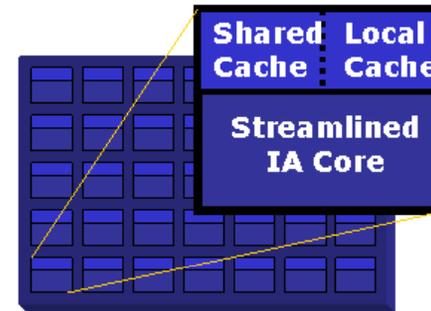
An Intel Exec's slide from IDF'2006



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

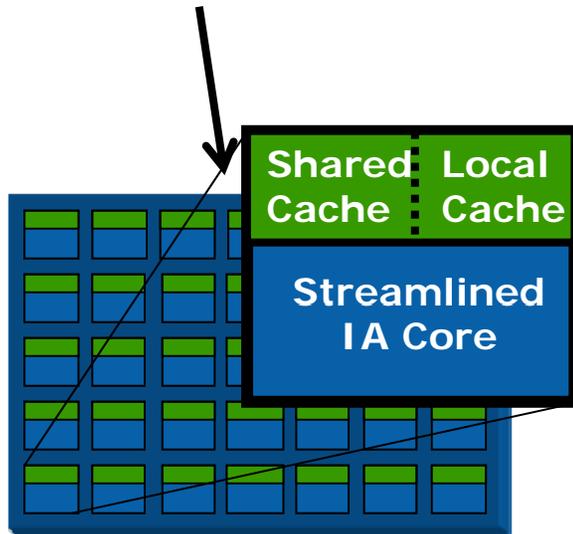


5-10 years: 10s-100s of energy efficient, IA cores optimized for multithreading

Challenging the sacred cows



Assumes cache coherent shared address space!



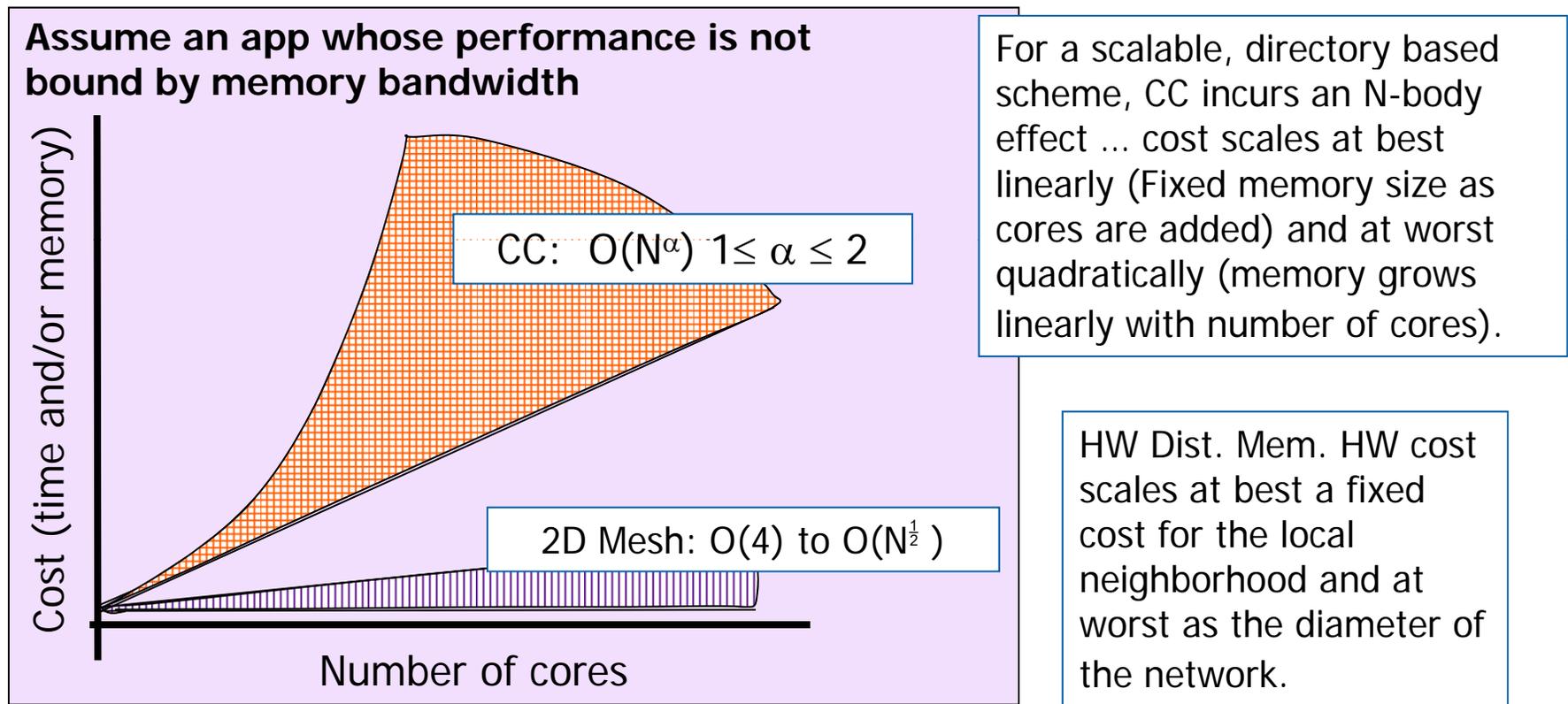
... IA cores optimized for multithreading

- 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



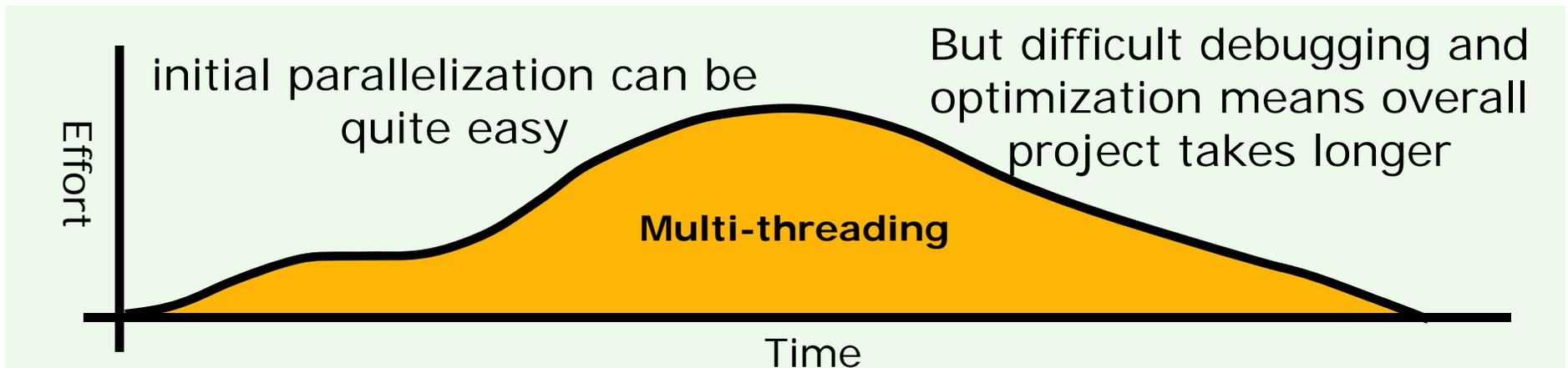
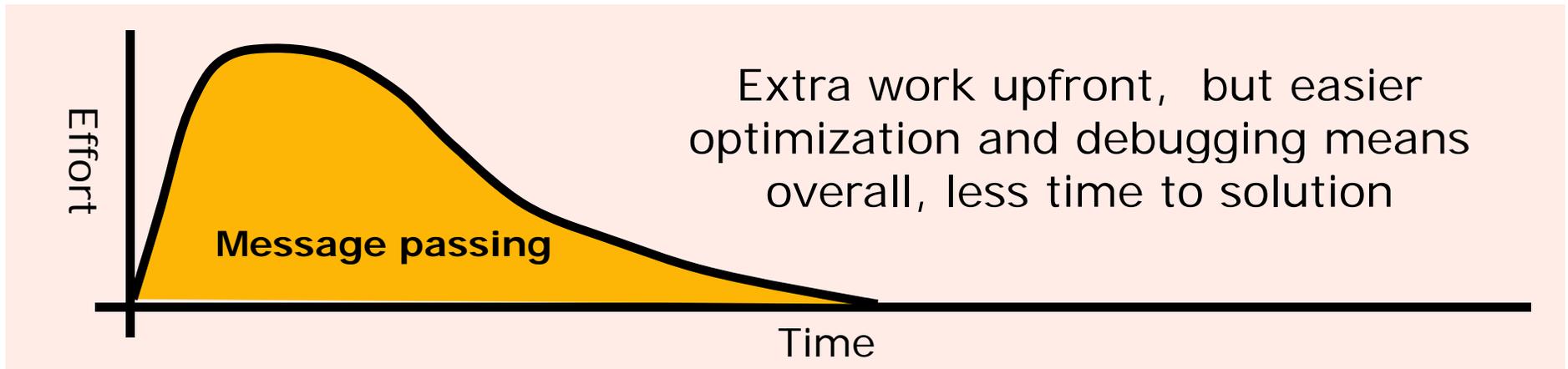
- As you scale the number of cores on a cache coherent system (CC), “cost” in “time and memory” grows to a point beyond which the additional cores are not useful in a single parallel program. This is the coherency wall.



... 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.*

* Rakesh Kumar, Timothy G. Mattson, Gilles Pokam, and Rob van der Wijngaart, “The case for Message Passing on Many-core Chips”, University of Illinois Champaign-Urbana Technical Report, UILU-ENG-10-2203 (CRHC 10-01), 2010.

Isn't shared memory programming easier? Not necessarily.

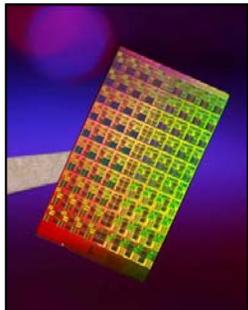


Proving that a shared address space program using semaphores is race free is an NP-complete problem*

The many core design challenge

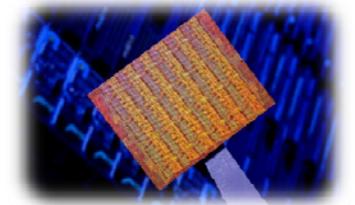


- **Scalable architecture:**
 - How should we connect the cores so we can scale as far as we need ($O(100\text{'s to } 1000)$ should be enough)?
- **Software:**
 - Can “general purpose programmers” write software that takes advantage of the cores?
 - Will ISV’s actually write scalable software?
- **Manufacturability:**
 - Validation costs grow steeply as the number of transistors grows. Can we use tiled architectures to address this problem?
 - For an N transistor budget ... Validate a tile (M transistors/tile) and the connections between tiles. Drops validation costs from $KO(N)$ 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

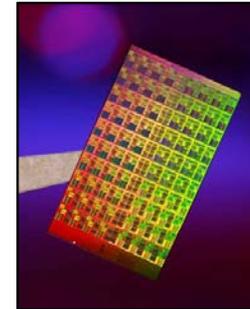


- The 80 core Research Processor
 - Max FLOPS/Watt in a tiled architecture
- The 48 core SCC processor
 - Scalable IA cores for software/platform research
- Software in a many core world

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
- Software in a many core world



Sriram R. Vangal, Jason Howard, Gregory Ruhl, Member, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Member, Tiju Jacob, Shailendra Jain, Vasantha Erraguntla, Clark Roberts, Yatin Hoskote, Nitin Borkar, and S. Borkar, **"An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS,"** IEEE Journal of Solid-State Circuits, Vol. 43, No. 1, Jan 2008.

Tim Mattson, Rob van der Wijngaart, Michael Frumkin, **"Programming Intel's 80 core terascale processor,"** Proc. of the 2008 ACM/IEEE conference on Supercomputing , SC08, Austin Texas, Nov. 2008

Acknowledgements

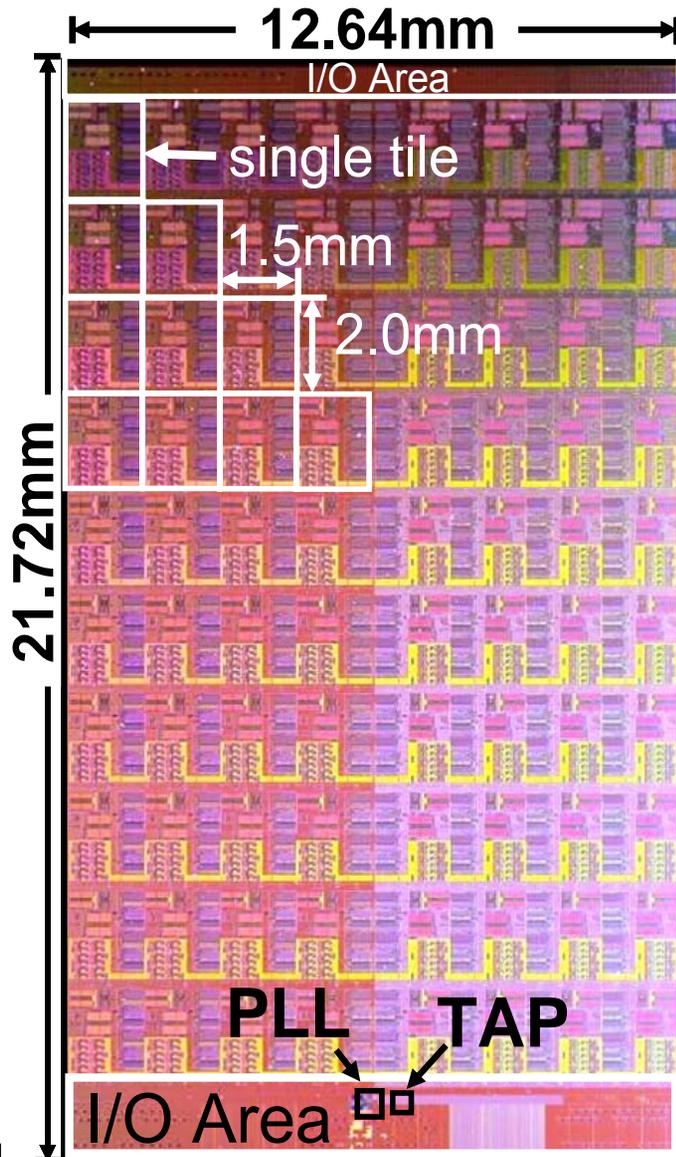


- Implementation
 - Circuit Research Lab Advanced Prototyping team (Hillsboro, OR and Bangalore, India)
- PLL design
 - Logic Technology Development (Hillsboro, OR)
- Package design
 - Assembly Technology Development (Chandler, AZ)
- The software team
 - Tim Mattson, Rob van der Wijngaart (Intel)
 - Michael Frumkin (then at Intel, now at Google)

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

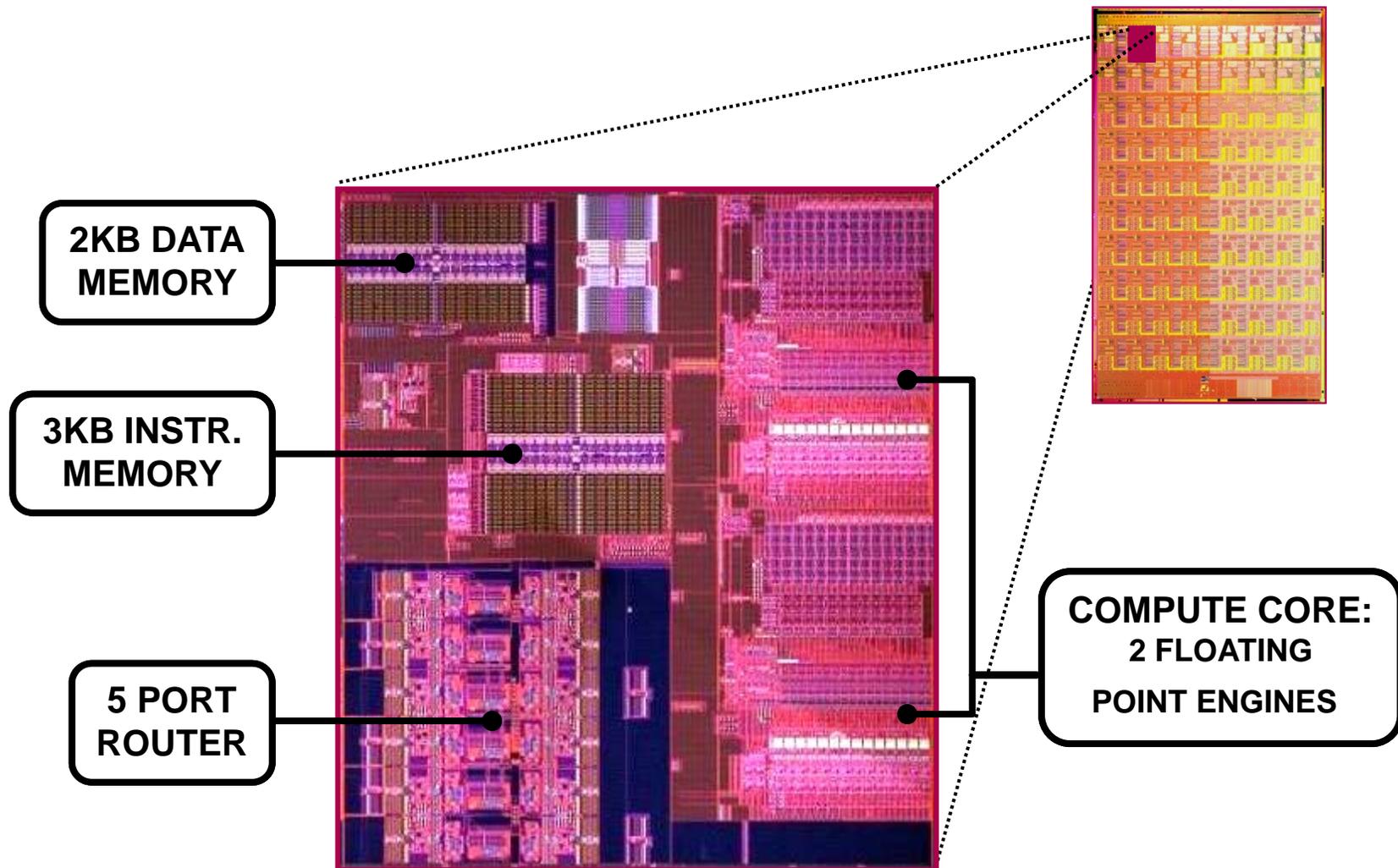
Die Photo and Chip Details



- Basic statistics:
 - 65 nm CMOS process
 - 100 Million transistors in 275 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:

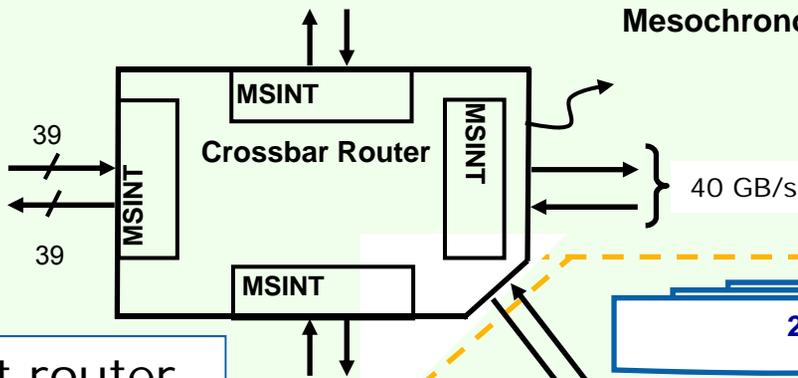
Intel's 80 core test chip (2006)



The "80-core" tile



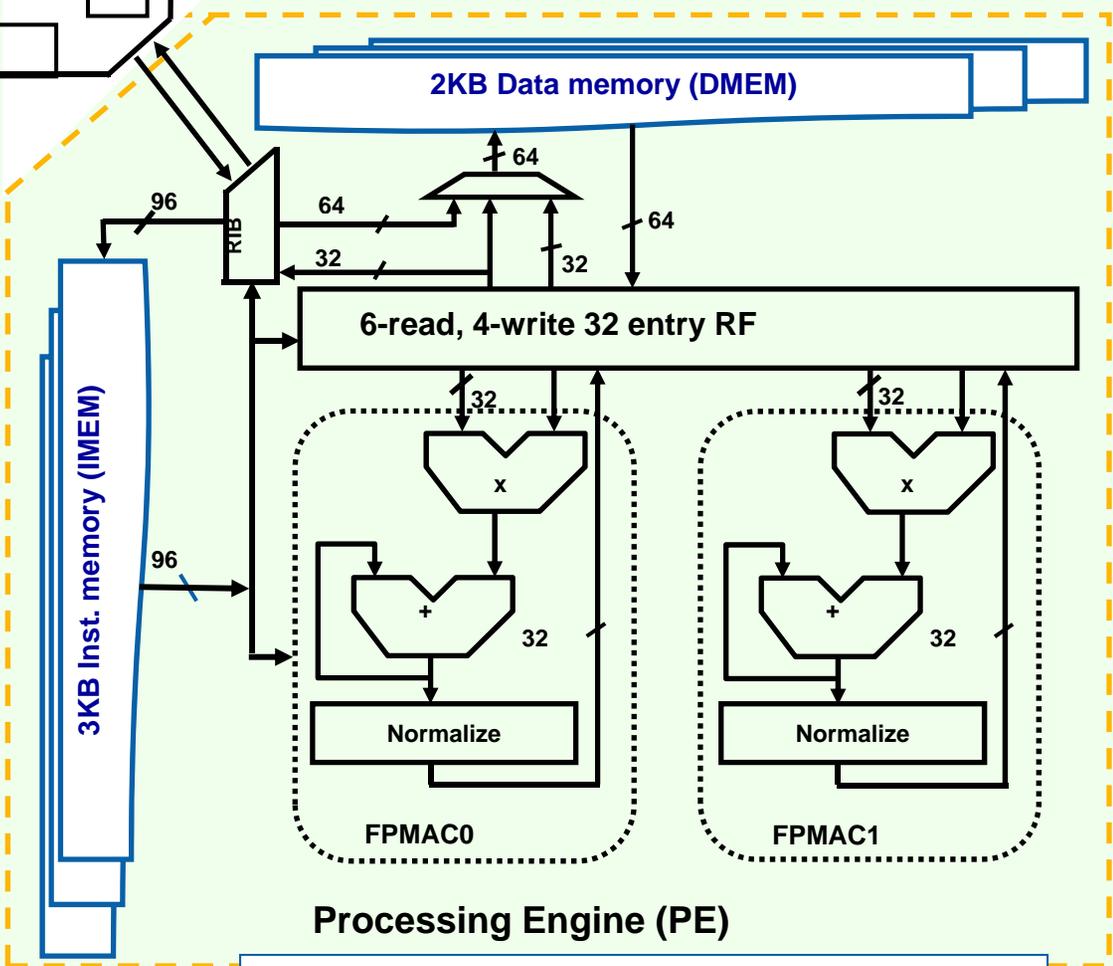
Mesochronous Interface



2 Kbyte Data Memory (512 SP words)



5 port router for a 2D mesh and 3D stacking



3 Kbyte Instr. Memory (256 96 bit instr)

Tile

2 single precision FPMAC units

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

Full Instruction Set



MULT	Multiply operands
ACCUM	Accumulate with previous result

FPU

LOAD, STORE	Move a pair of floats between register file & data memory.
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
SENDD[H A D T]	Send Data header, address, data, and tail
WFD	Stall while waiting for data from any tile.
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
JUMP	Jump to the specified program counter address

Program flow

NAP	Put FPUs to sleep
WAKE	Wake FPUs from sleep

Sleep

Instruction word and latencies



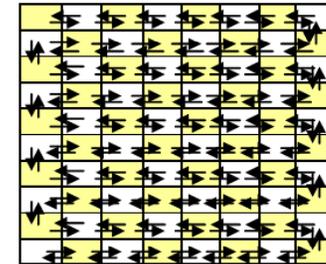
- 96-bit instruction word, up to 8 operations/cycle

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

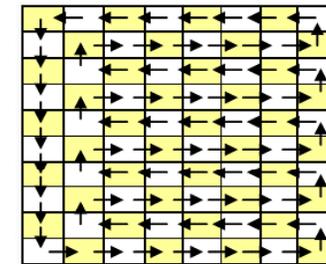
What did we do with the chip?



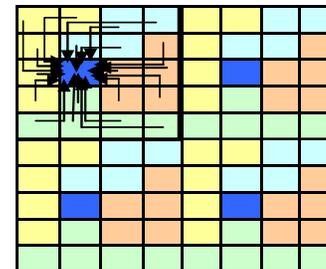
- 4 applications kernels
 - **Stencil**
 - 2D PDE solver (heat diffusion equation) using a Gauss Seidel algorithm



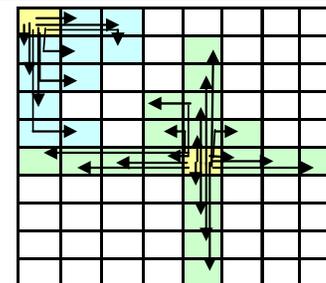
- **SGEMM (Matrix Multiply)**
 - $C = A * B$ with rectangular matrices



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



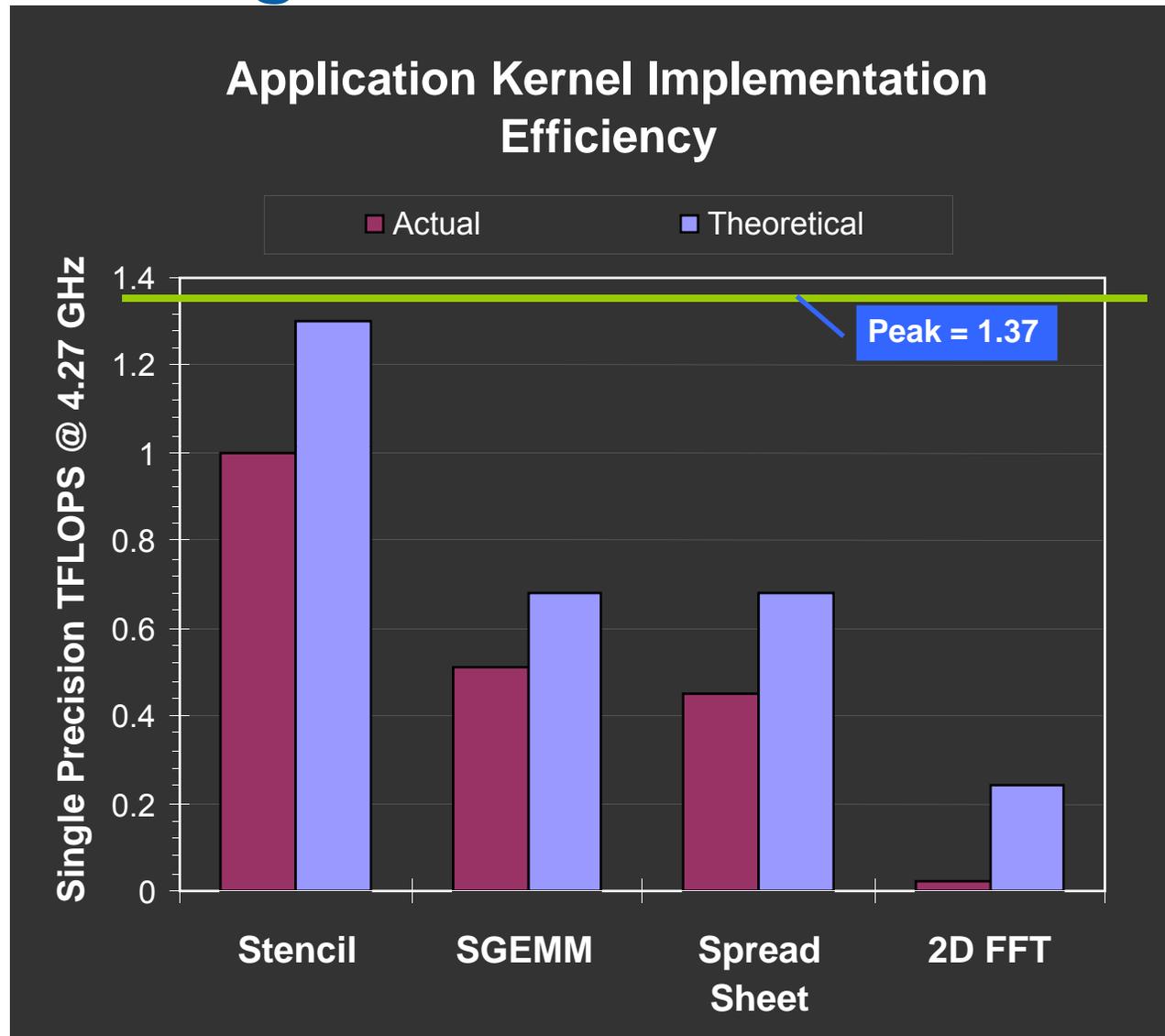
- **2D FFT**
 - 2D FFT of dense array on an 8 by 8 subgrid.



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

Communication Patterns

Programming Results

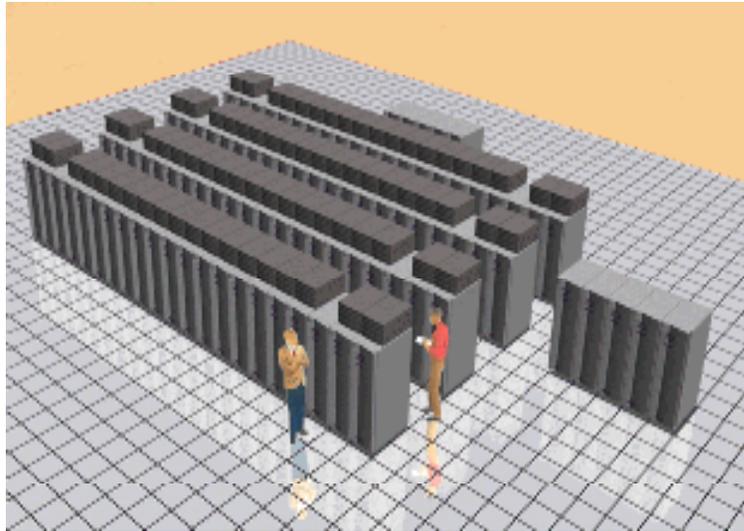


Theoretical numbers from operation/communication counts and from rate limiting bandwidths.

Why this is so exciting!



First TeraScale* computer: 1997



Intel's ASCI Option Red

Intel's ASCI Red Supercomputer

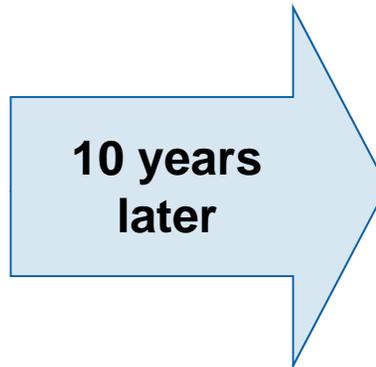
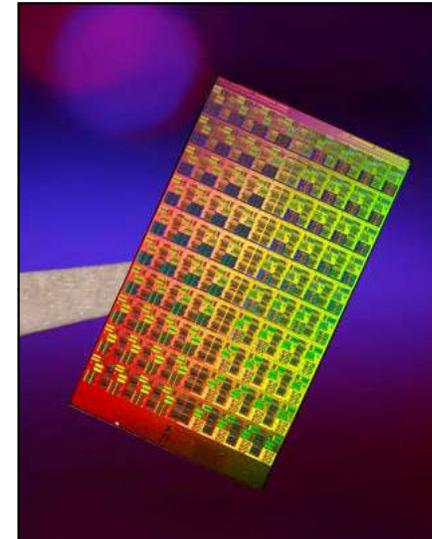
9000 CPUs

one megawatt of electricity.

1600 square feet of floor space.

*Double Precision TFLOPS running MP-Linpack

First TeraScale% chip: 2007



Intel's 80 core TeraScale Chip

1 CPU

97 watt

275 mm²

%Single Precision TFLOPS running stencil

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

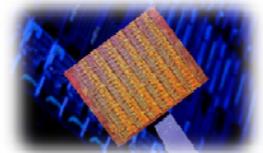


- Minimize message passing overhead.
 - Routers wrote directly into memory without interrupting computing ... i.e. any core could write directly into the memory of any other core. This led to extremely small comm. latency on the order of 2 cycles.
 - Programmers can assist in keeping power low if sleep/wake 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
- Software in a many core world



J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Paillet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De1, R. Van Der Wijngaart, T. Mattson,
"A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS",
Proceedings of the International Solid-State Circuits Conference, Feb 2010

Timothy G. Mattson, Rob F. Van der Wijngaart, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, Saurabh Dighe
"The 48 core SCC Processor: A programmers view"
Submitted to Proc. of the 2010 ACM/IEEE Conference on Supercomputing

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 and
BareMetalC workflow
Linux for SCC
System Interface FPGA development
TCP/IP network drivers

Michael Riepen
Thomas Lehnig
Paul Brett
Matthias Steidl
Werner Haas

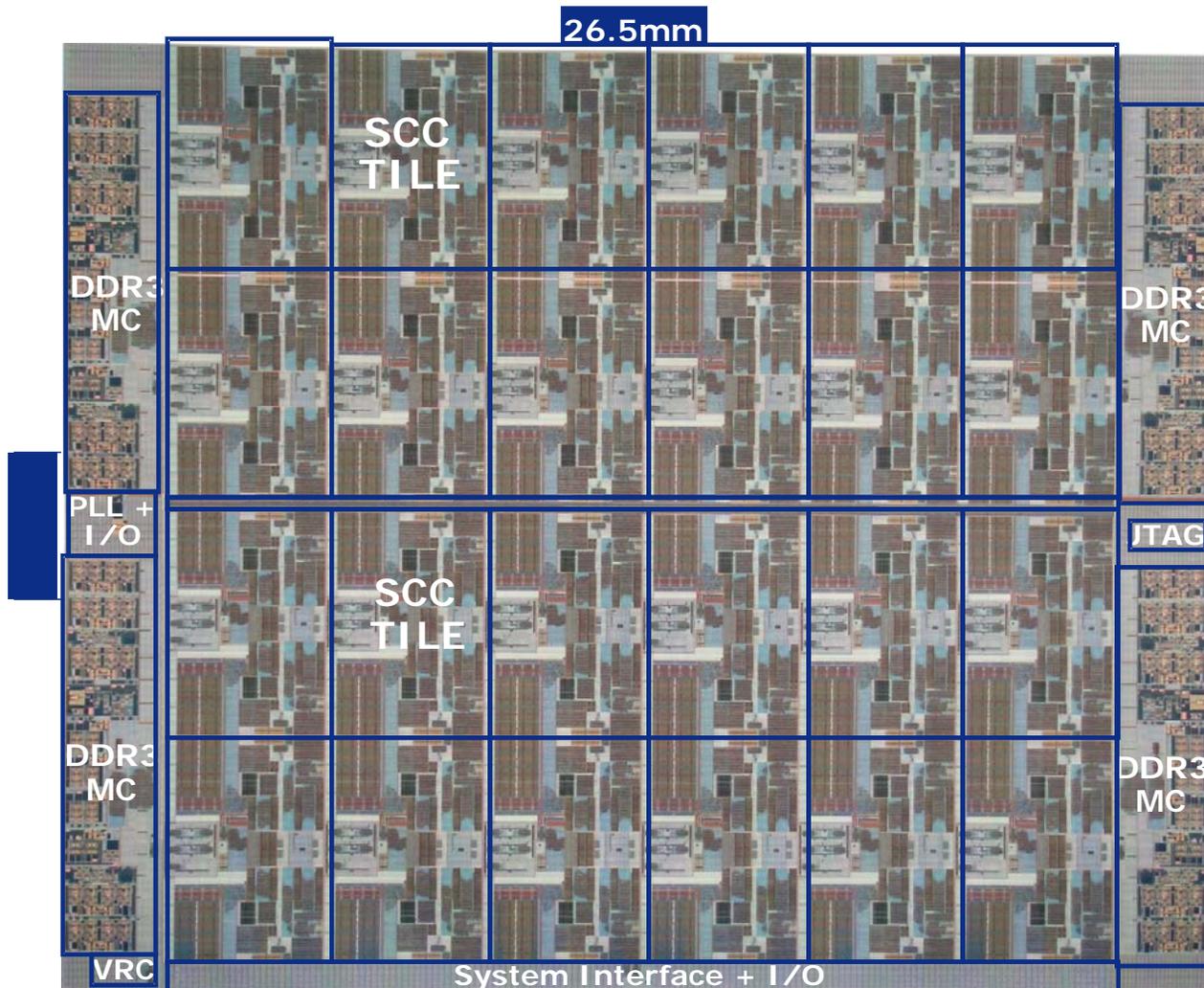
- And the HW-team that worked closely with the SW group:

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



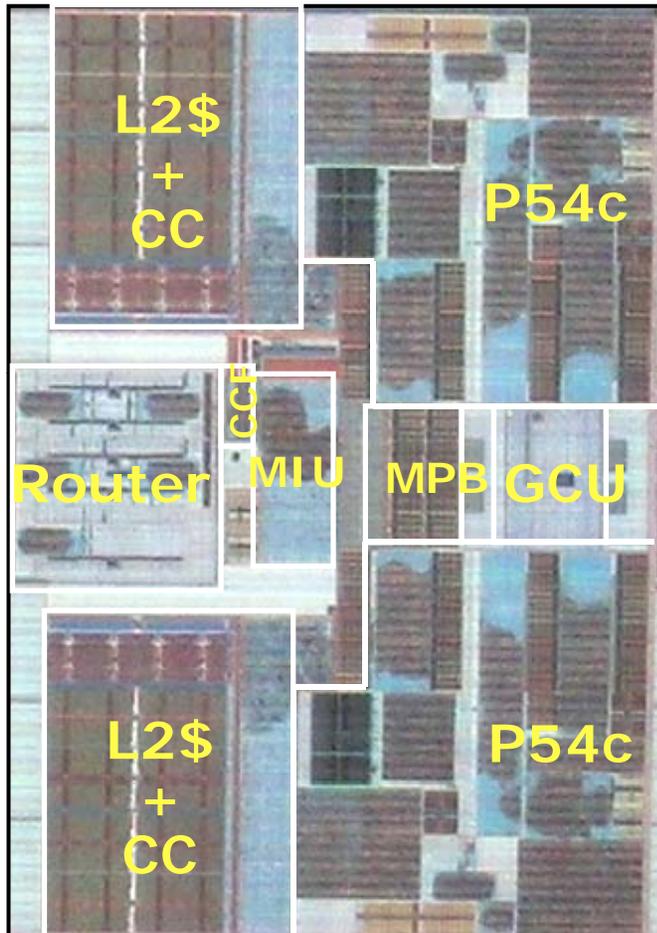
SCC full chip

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



Technology	45nm Process
Interconnect	1 Poly, 9 Metal (Cu)
Transistors	Die: 1.3B, Tile: 48M
Tile Area	18.7mm²
Die Area	567.1mm²

SCC Dual-core Tile

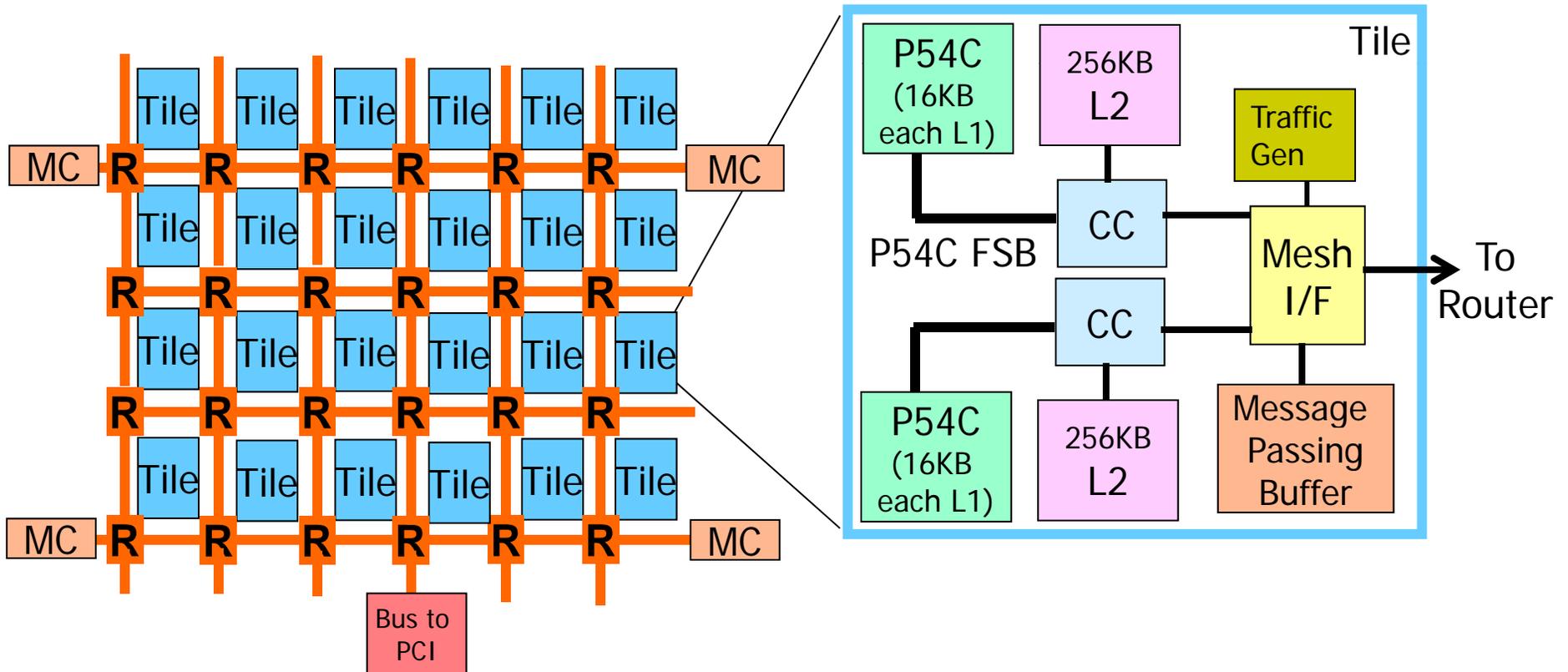


- 2 P54C cores (16K L1\$/core)
- 256K L2\$ per core
- 8K Message passing buffer
- Clock Crossing FIFOs b/w Mesh interface unit and Router

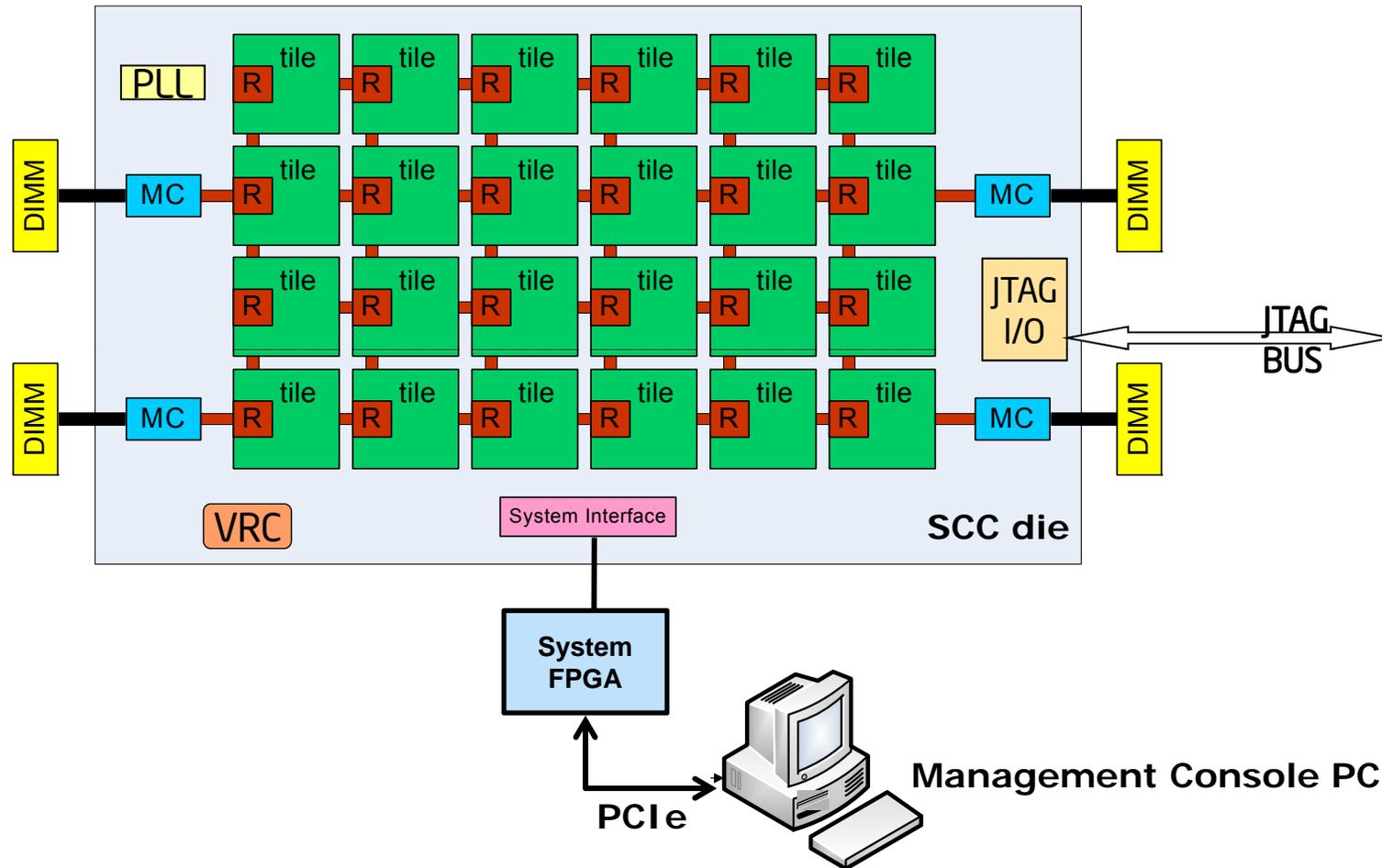
- Tile area 18.7mm²
- Core are 3.9mm²
- Cores and uncore units @1GHz
- Router @2GHz

Hardware view of SCC

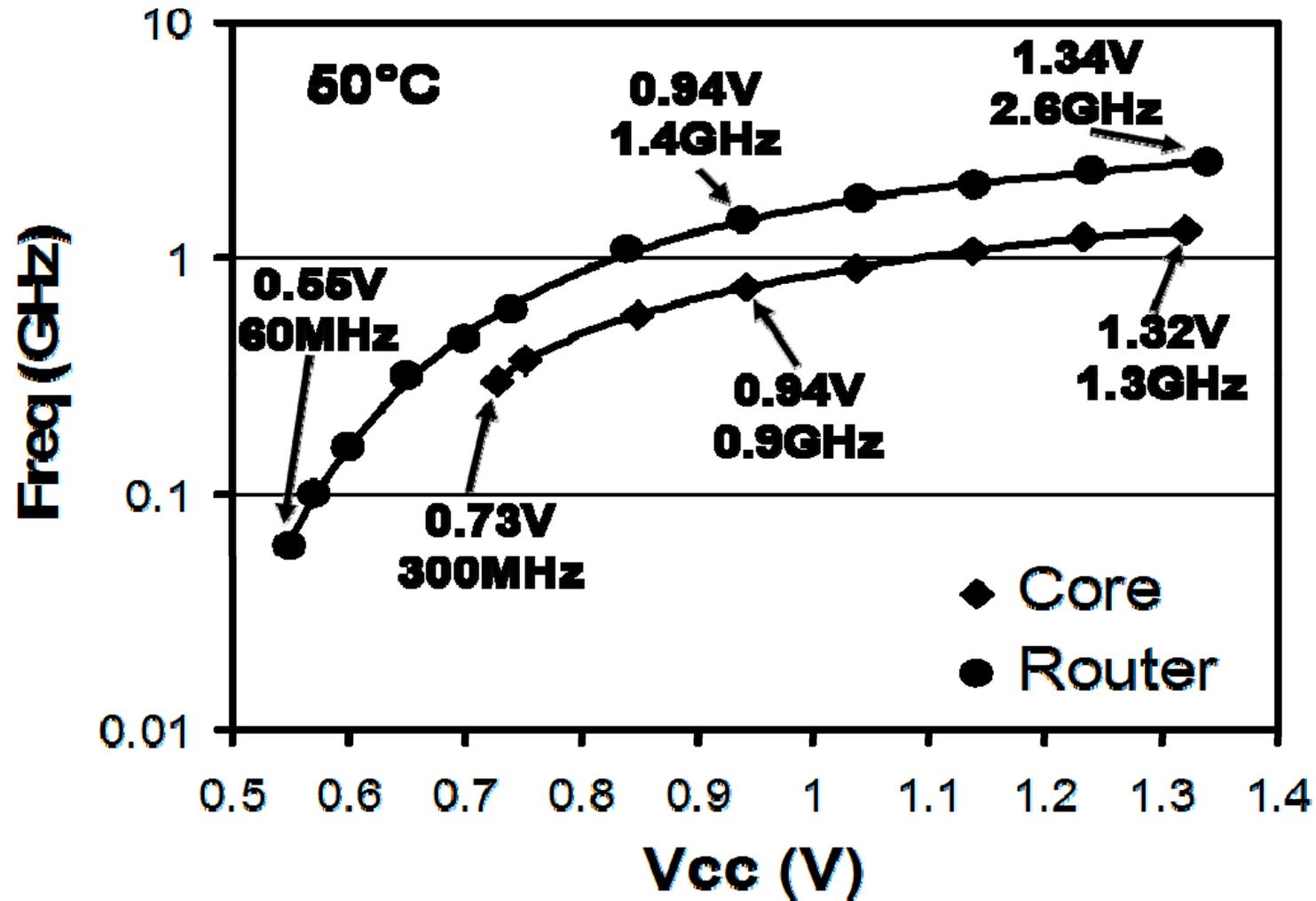
- 48 P54C cores in 6x4 mesh with 2 cores per tile
- 45 nm, 1.3 B transistors, 25 to 125 W
- 16 to 64 GB total main memory using 4 DDR3 MCs
- 2 Tb/s bisection bandwidth @ 2 Ghz



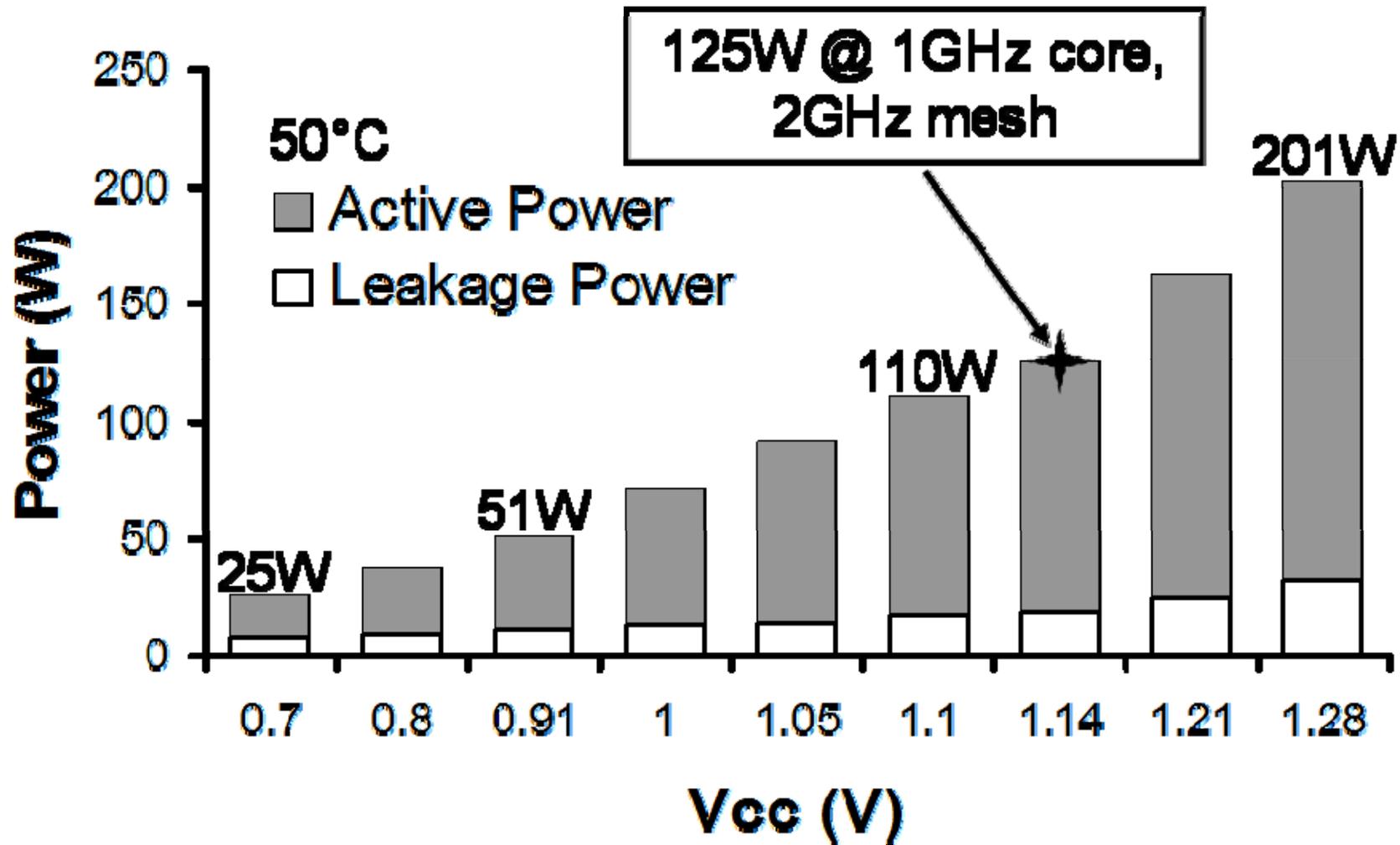
SCC system overview



Core & Router Fmax



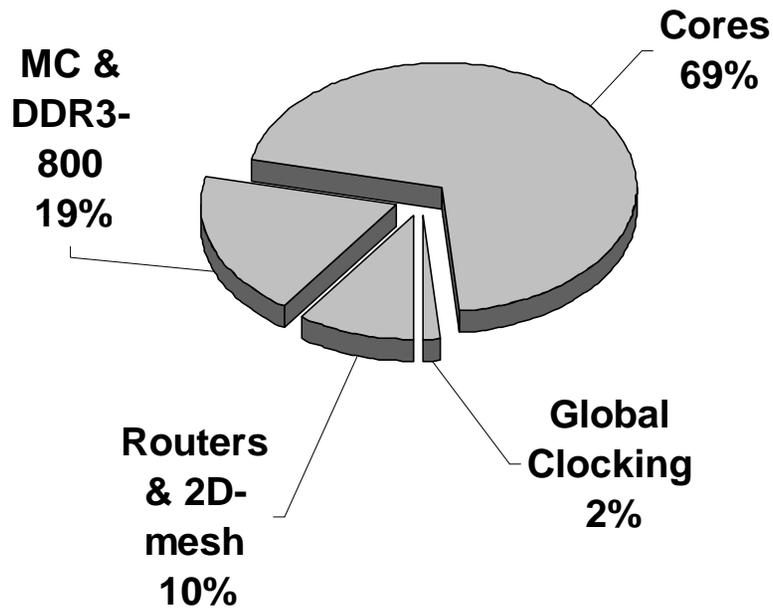
Measured full chip power



Power breakdown



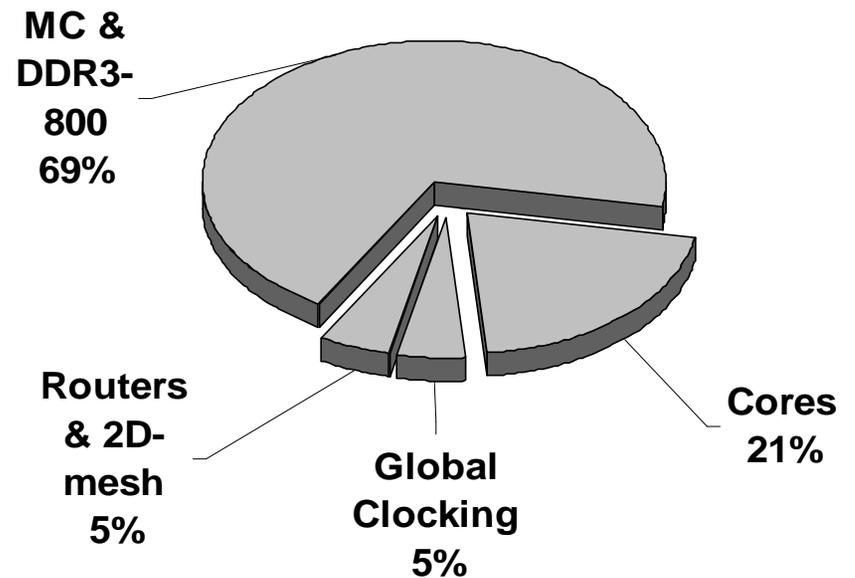
Full Power Breakdown Total -125.3W



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

Cores-1GHz, Mesh-2GHz, 1.14V, 50°C

Low Power Breakdown Total - 24.7W



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

Cores-125MHz, Mesh-250MHz, 0.7V, 50°C

SCC Software research goals



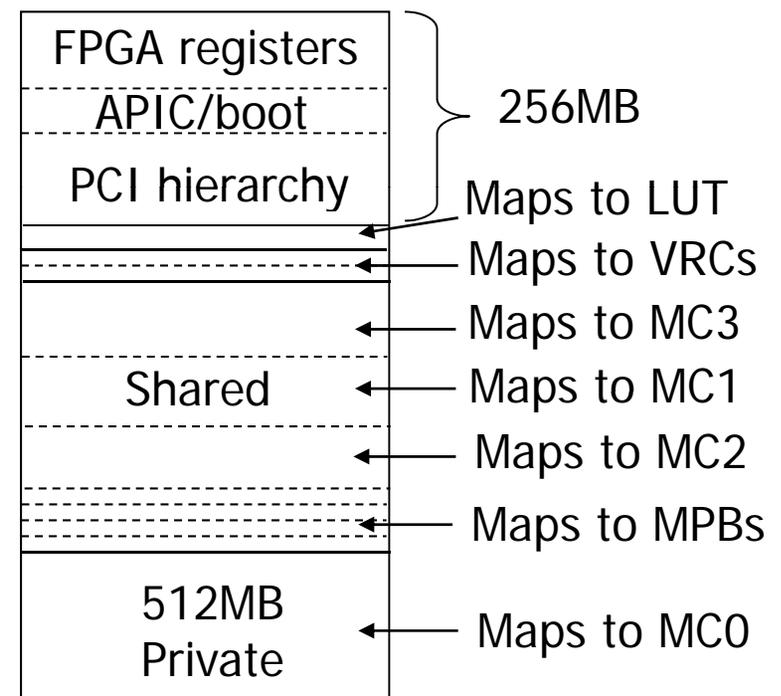
- Understand programmability and application scalability of many-core chips.
- Answer question “what can you do with a many-core chip that has (some) shared non-cache-coherent memory?”
- Study usage models and techniques for software controlled power management
- Sample software for other programming model and applications researchers (industry partners, Flame group at UT Austin, UPCRC, OU ... i.e. the MARC program) Y

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

SW control of 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

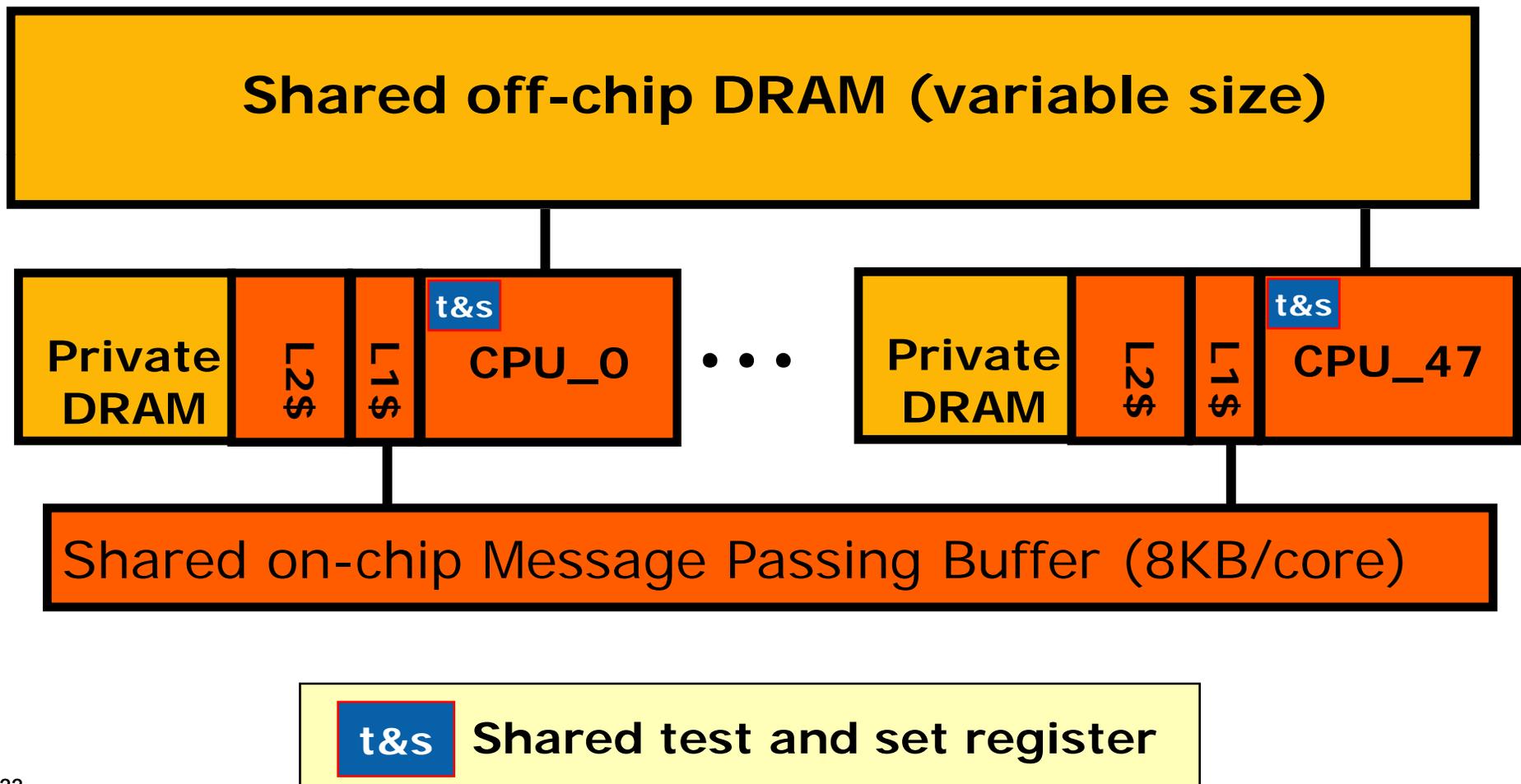


A Programmer's view of SCC



(one of several)

- 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's message passing library: RCCE



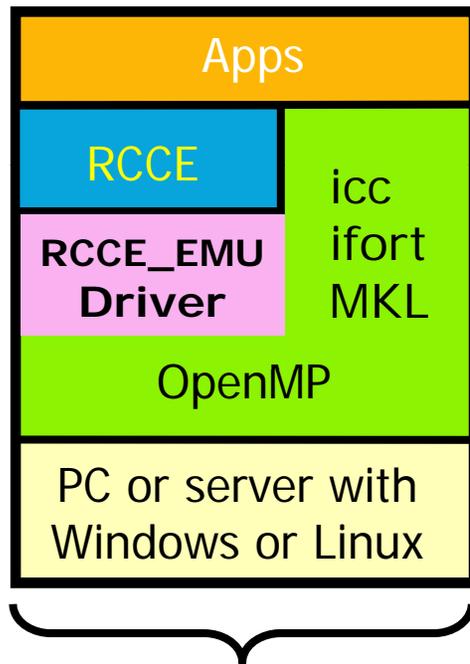
- RCCE is a compact, lightweight communication environment.
 - SCC and RCCE were designed together side by side:
 - ... a true HW/SW co-design project.
- RCCE is a research vehicle to understand how message passing APIs map onto many core chips.
- RCCE is for experienced parallel programmers willing 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 or thread).

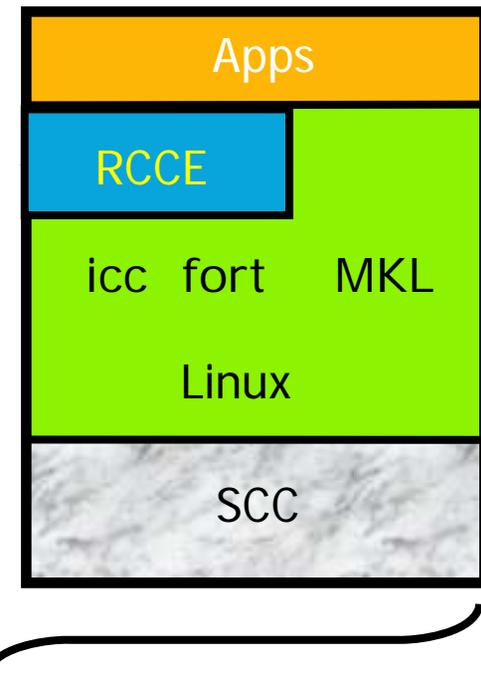


SCC Platforms

- 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)



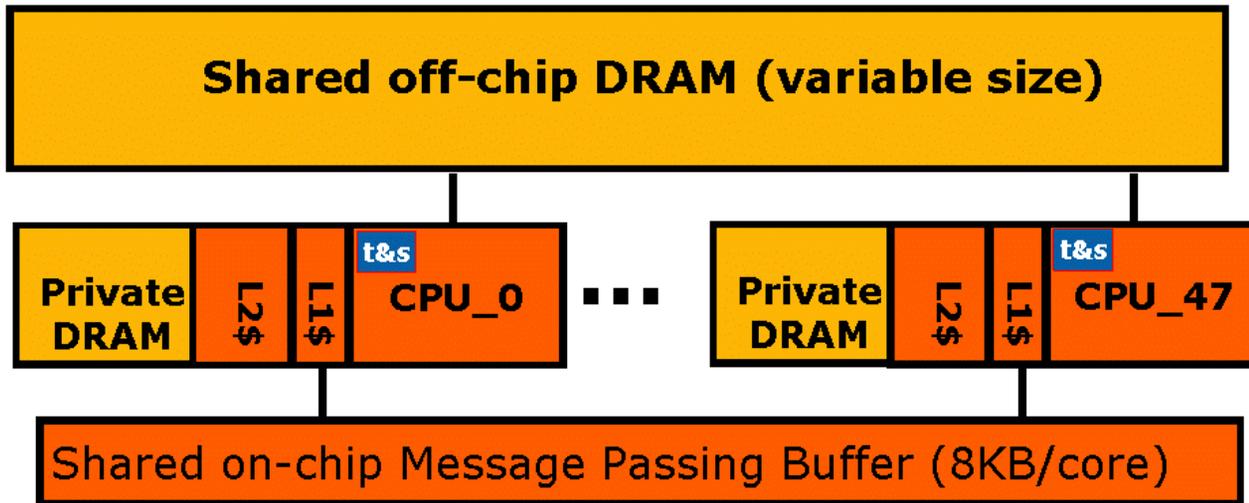
Functional emulator,
based on OpenMP.



SCC board – NO OpenMP

RCCE supports greatest common denominator between the three platforms

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:

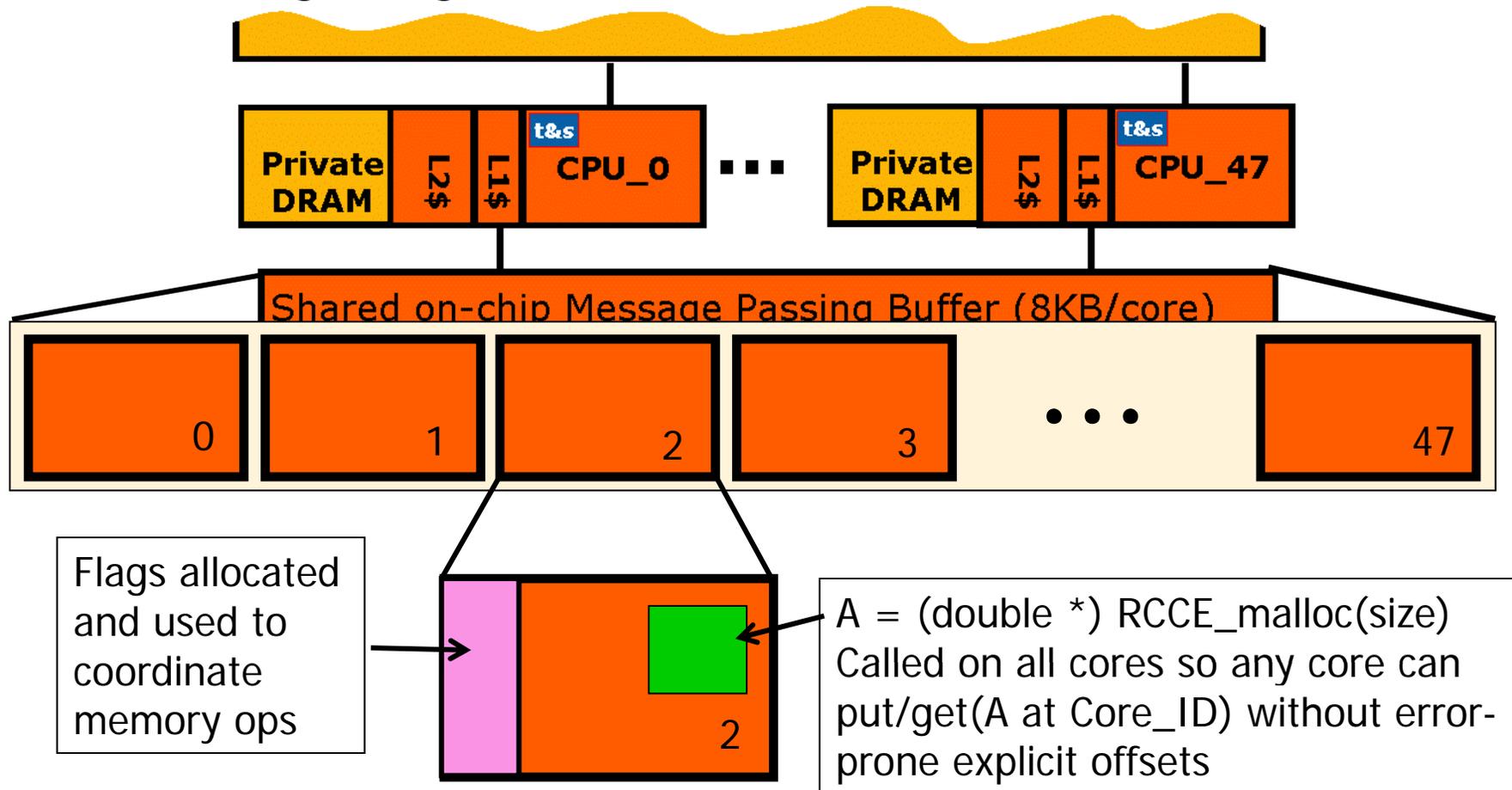
- If data changed by another core and image still in L1, read returns stale data.
 - **Solution: Invalidate before read.**
- L1 has write-combining buffer; write incomplete line? expect trouble!
 - **Solution: don't. Always push whole cache lines**
- If image of line to be written already in L1, write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



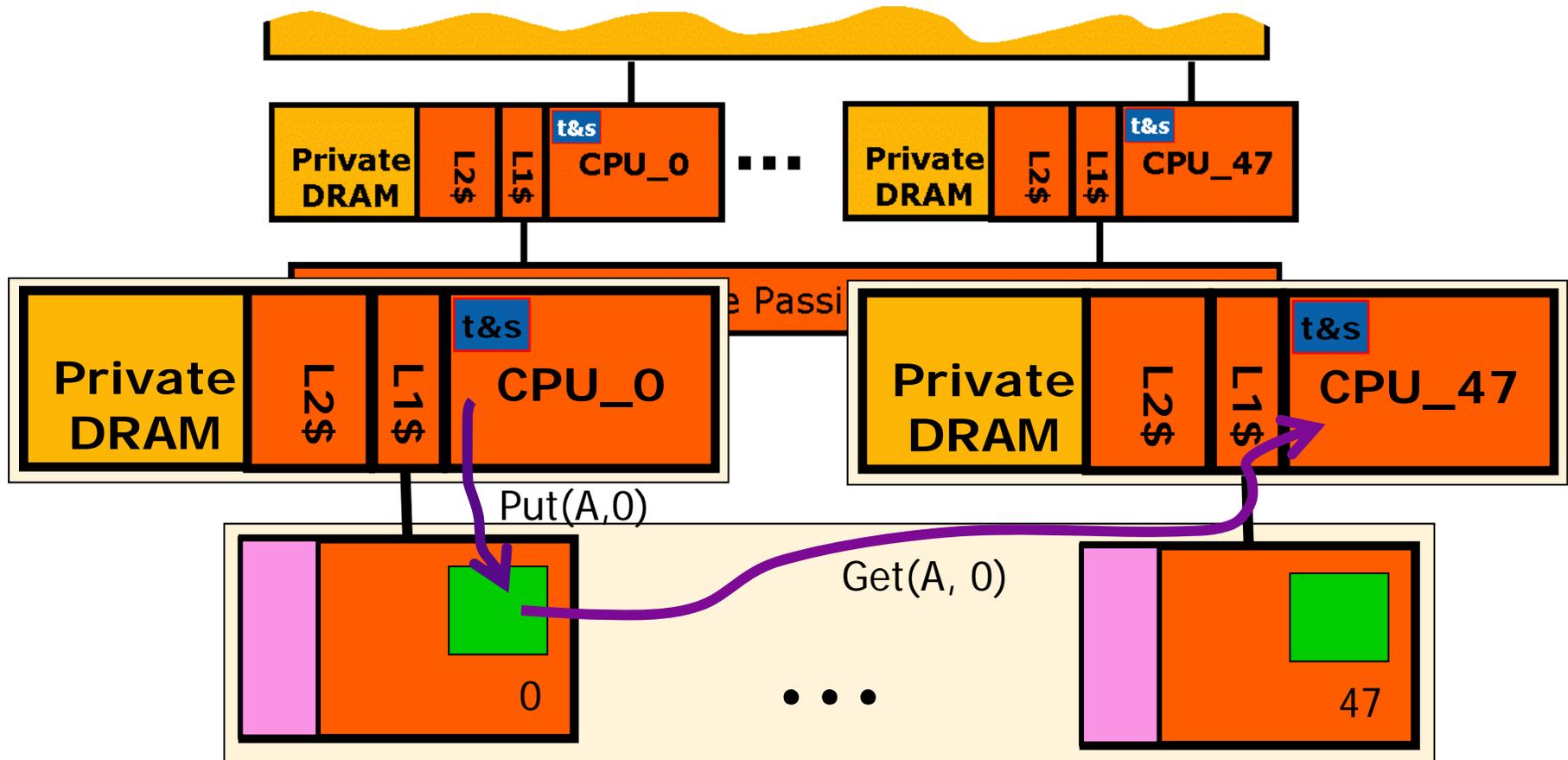
- Treat Msg Pass Buf (MPB) as 48 smaller buffers ... one per core.
- Symmetric name space ... Allocate memory as a collective op. Each core gets a variable with the given name at a fixed offset from the beginning of a core's MPB.



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"

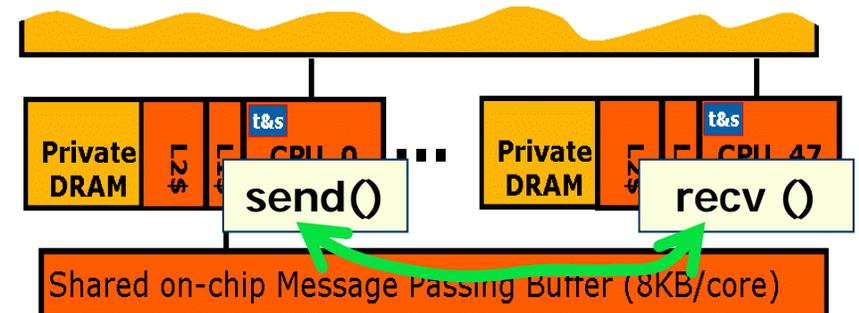
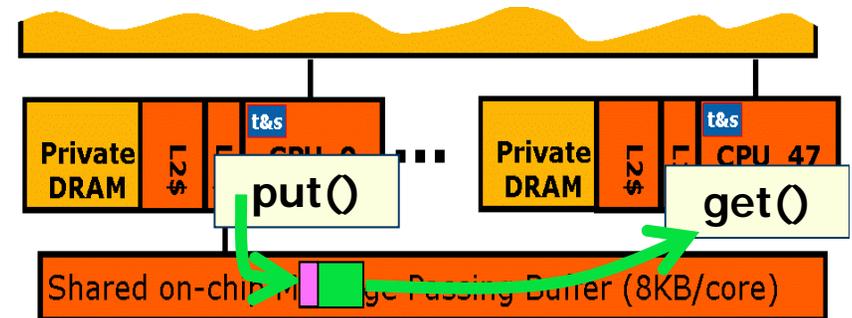
The RCCE library



- RCCE API provides the basic message passing functionality expected in a tiny communication library:

- One + two sided interface (put/get + send/rcv) with synchronization flags and MPB management exposed.
 - The “gory” interface for programmers who need the most detailed control over SCC

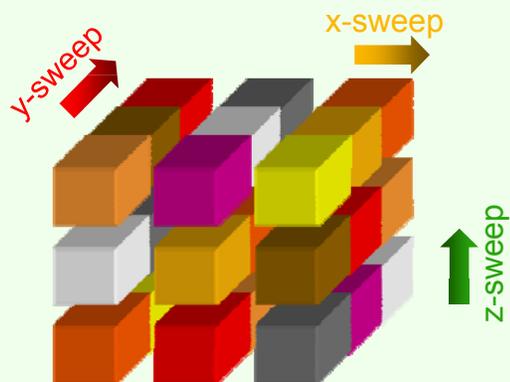
- Two sided interface (send/rcv) with most detail (flags and MPB management) hidden.
 - The “basic” interface for typical application programmers.



Linpack and NAS Parallel benchmarks



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

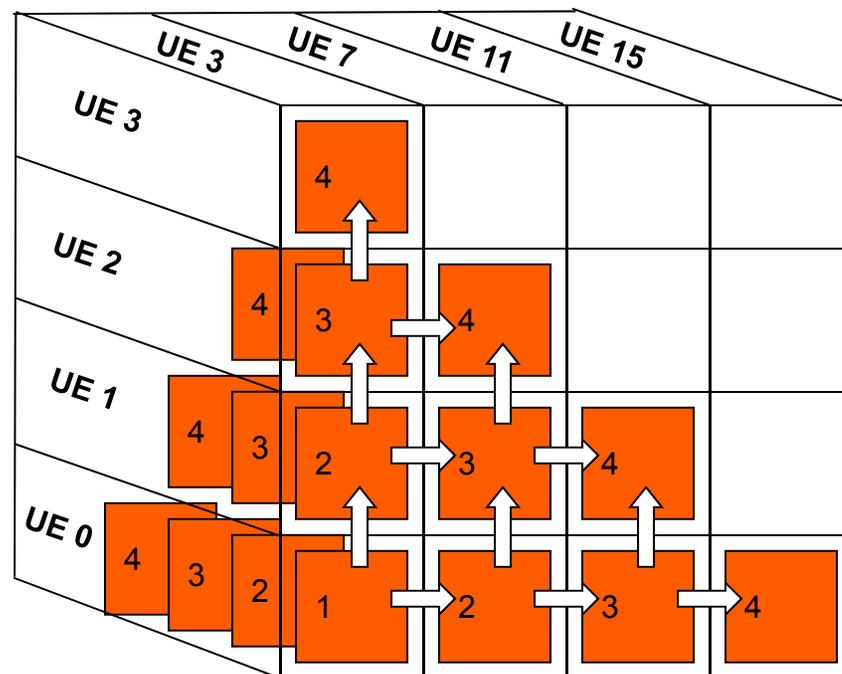


2. BT: Multipartition decomposition

- 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

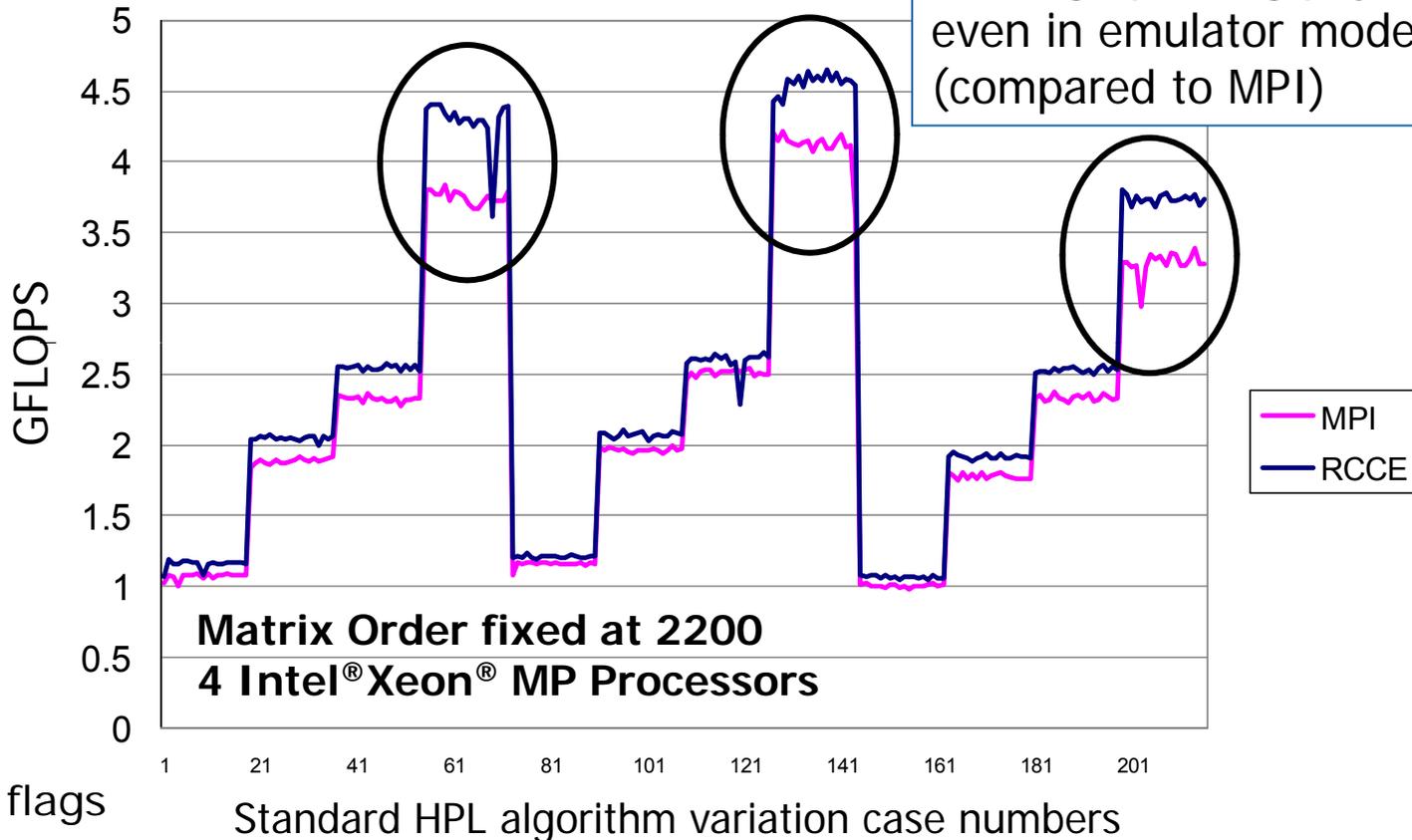
- Define 2D-pipeline process.
 - await data (bottom+left)
 - compute new tile
 - send data (top+right)



RCCE functional emulator vs. MPI



HPL implementation of the LINPACK benchmark



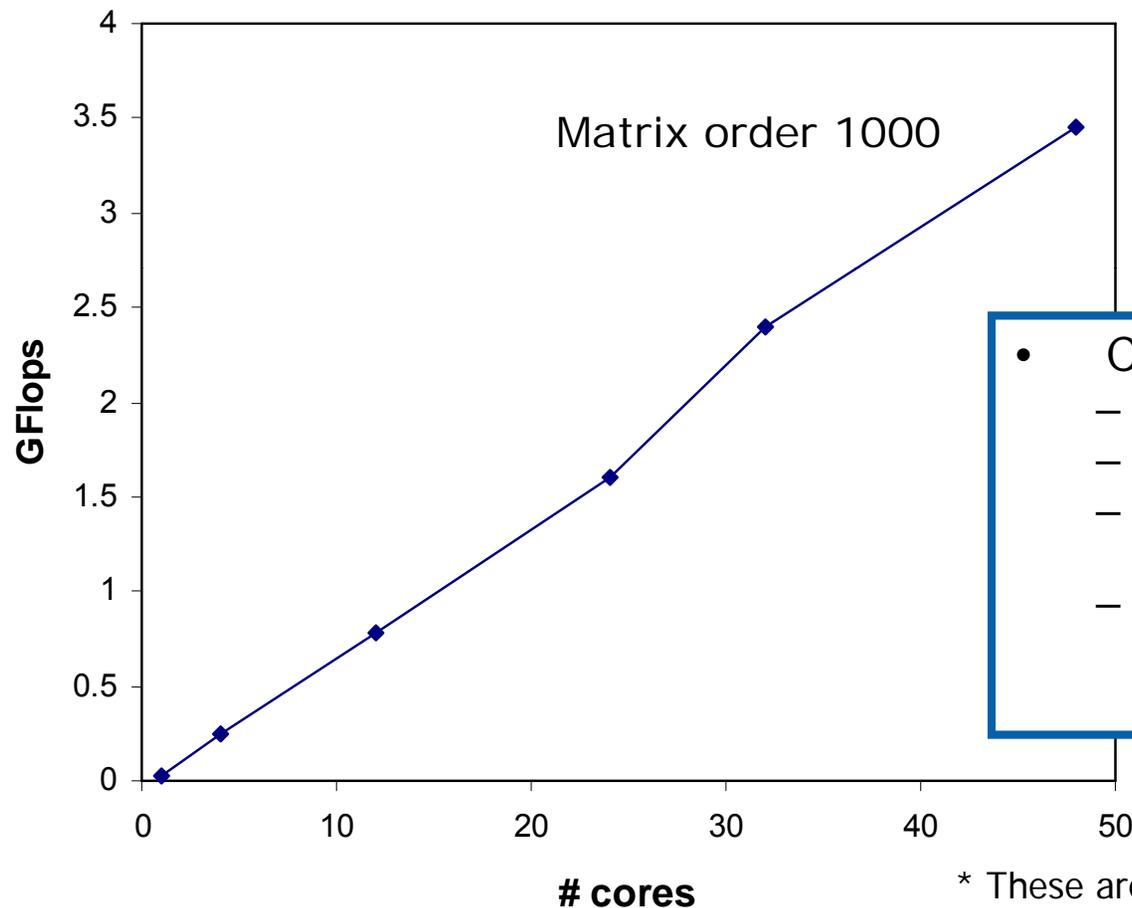
These results provide a comparison of RCCE and MPI on an older 4 processor Intel® Xeon® MP SMP platform* with tiny 4x4 block sizes. These are not official MP-LINPACK results.

*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

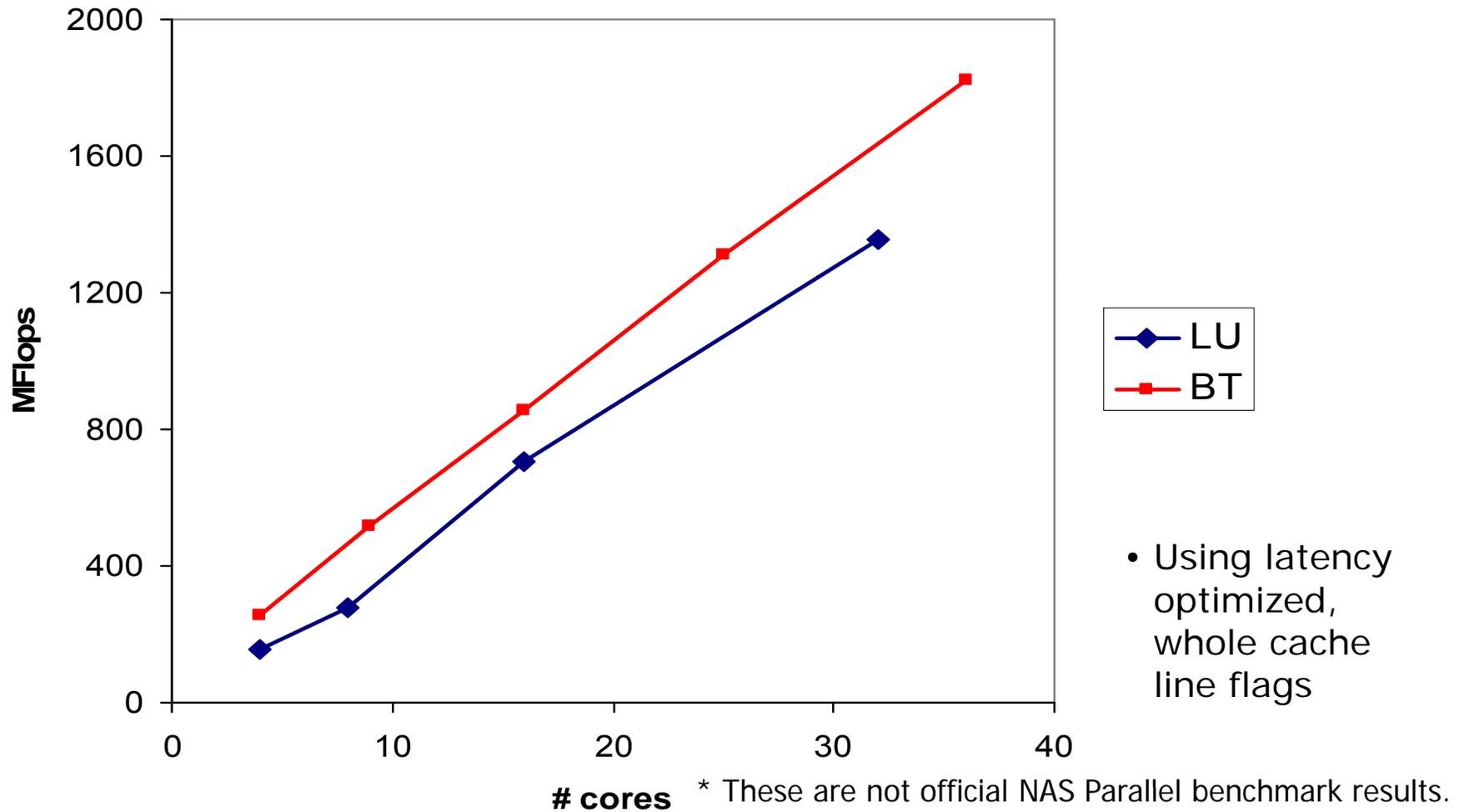
* These are not official LINPACK benchmark results.

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

Third party names are the property of their owners.

LU/BT NAS Parallel Benchmarks,

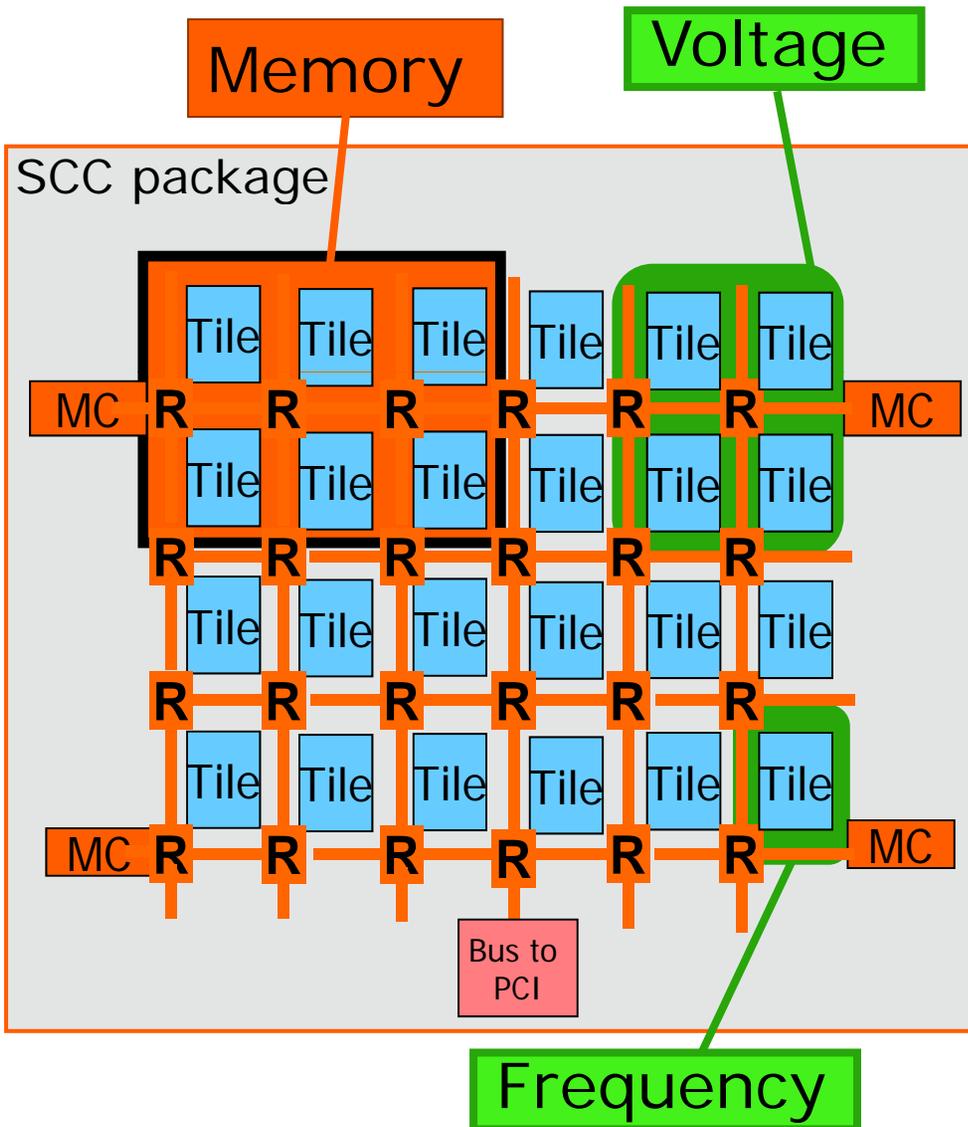
Problem size: Class A, 64 x 64 x 64 grid*



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

Third party names are the property of their owners.

Power and memory-controller domains



$$\text{Power} \sim F V^2$$

- 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



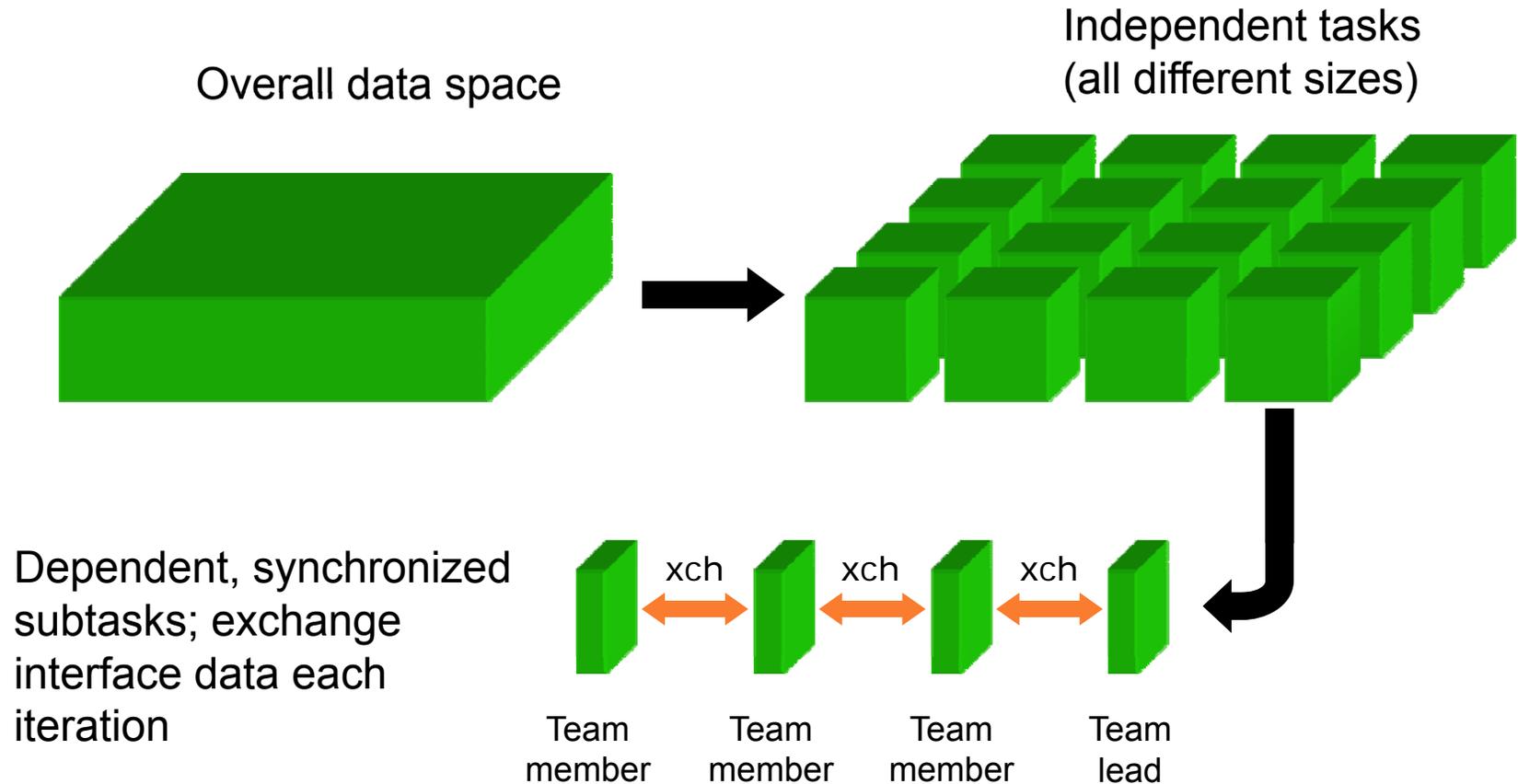
RCCE Power Management API

- 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_iset_power():
 - Input a frequency divisor (2 to 16) setting, and this will set the min voltage consistent with that frequency
 - RCCE_wait_power():
 - returns when power change is done
 - RCCE_set_frequency():
 - Set the frequency divisor (2 to 16)
- 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.



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.

Agenda

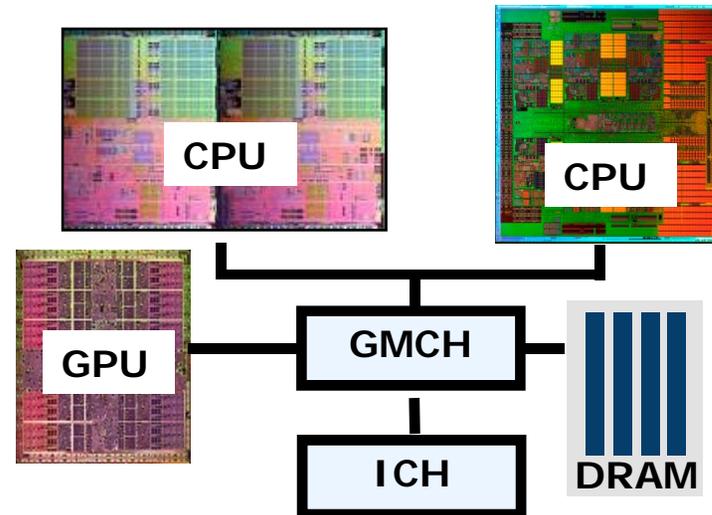


- The 80 core Research Processor
 - Max FLOPS/Watt in a tiled architecture
- The 48 core SCC processor
 - Scalable IA cores for software/platform research
- ➔ • Software in a many core world

Heterogeneous computing



- A modern platform has:
 - CPU(s)
 - GPU(s)
 - DSP processors
 - ... other?

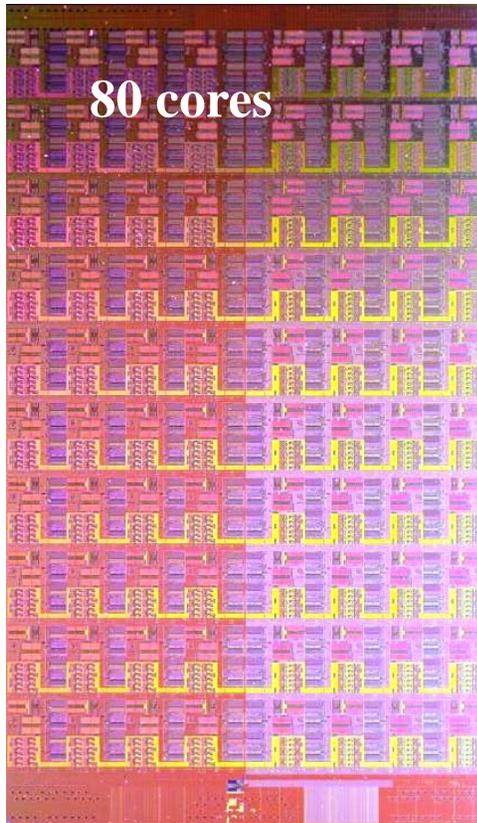


- Programmers need to make the best use of all the available resources from within a single program:
 - One program that runs well (i.e. reasonably close to “hand-tuned” performance) on a heterogeneous mixture of processors.

Microprocessor trends

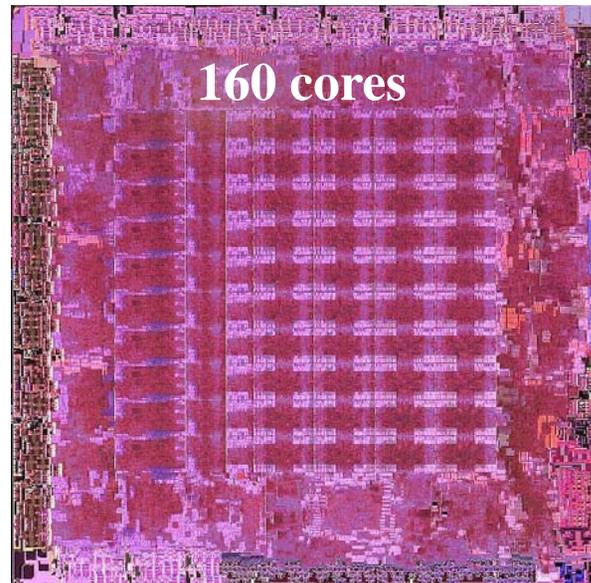


Individual processors are many core (and often heterogeneous) processors.



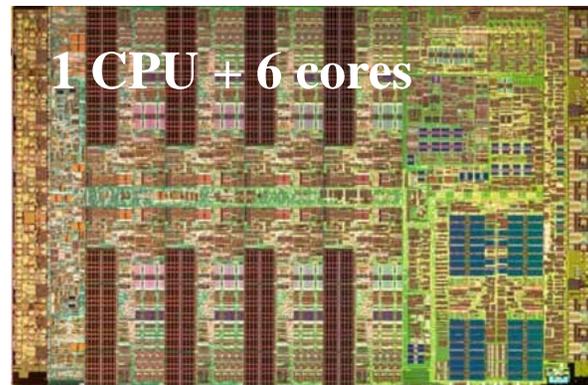
80 cores

Intel Terascale
research chip



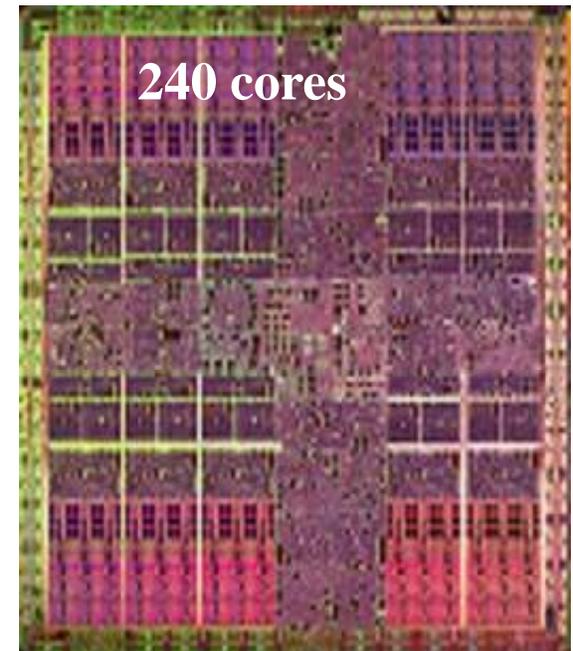
160 cores

ATI RV770



1 CPU + 6 cores

IBM Cell



240 cores

NVIDIA Tesla C1060

The many-core challenge



- We have arrived at many-core solutions not because of the success of our parallel software but because of our failure to keep increasing CPU frequency.
- Result: a fundamental and dangerous mismatch
 - Parallel hardware is ubiquitous.
 - Parallel software is rare

Our challenge ... make parallel software as routine as our parallel hardware.

Solution: Find A Good parallel programming model, right?



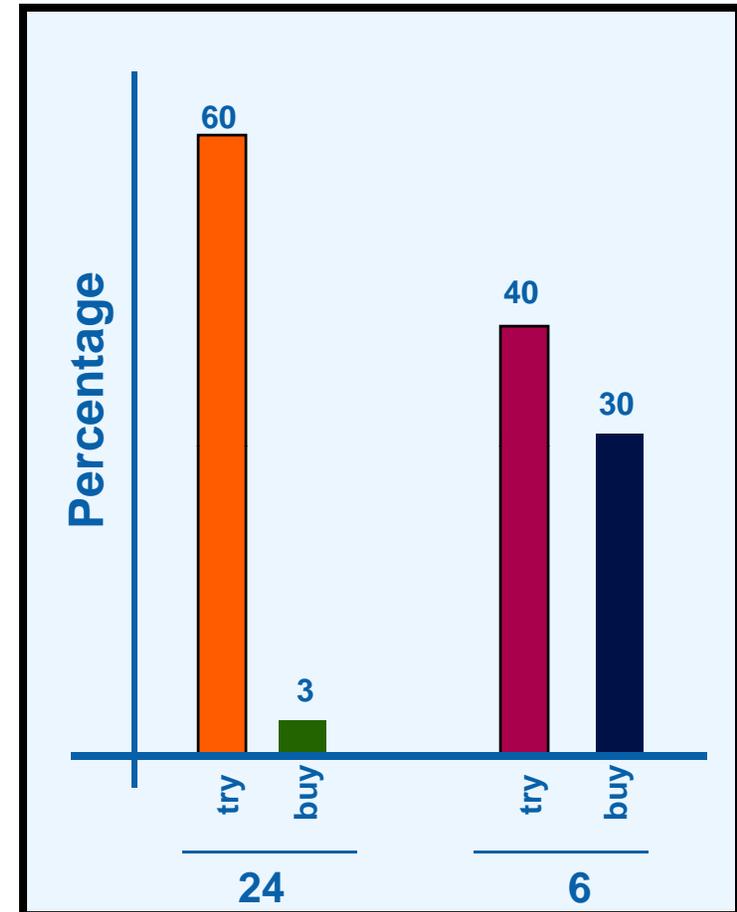
ABCPL	CORRELATE	GLU	Mentat	Parafrese2	pC++
ACE	CPS	GUARD	Legion	Paralation	SCCHEDULE
ACT++	CRL	HASL.	Meta Chaos	Parallel-C++	SciTL
Active messages	CSP	Haskell	Midway	Parallaxis	SDDA.
Adl	Cthreads	HPC++	Millipede	ParC	SHMEM
Adsmith	CUMULVS	JAVAR.	CparPar	ParLib++	SIMPLE
ADDAP	DAGGER	HORUS	Mirage	ParLin	Sina
AFAPI	DAPPLE	HPC	MpC	Parmacs	SISAL.
ALWAN	Data Parallel C	IMPACT	MOSIX	Parti	distributed smalltalk
AM	DC++	ISIS.	Modula-P	pC	SMI.
AMDC	DCE++	JAVAR	Modula-2*	PCN	SONiC
AppLeS	DDD	JADE	Multipol	PCP:	Split-C.
Amoeba	DICE.	Java RMI	MPI	PH	SR
ARTS	DIPC	javaPG	MPC++	PEACE	Sthreads
Athapascan-0b	DOLIB	JavaSpace	Munin	PCU	Strand.
Aurora	DOME	JIDL	Nano-Threads	PET	SUIF.
Automap	DOSMOS.	Joyce	NESL	PENNY	Synergy
bb_threads	DRL	Khoros	NetClasses++	Phosphorus	Telegrphos
Blaze	DSM-Threads	Karma	Nexus	POET.	SuperPascal
BSP	Ease .	KOAN/Fortran-S	Nimrod	Polaris	TCGMSG.
BlockComm	ECO	LAM	NOW	POOMA	Threads.h++.
C*.	Eiffel	Lilac	Objective Linda	POOL-T	TreadMarks
"C* in C	Eilean	Linda	Occam	PRESTO	TRAPPER
C**	Emerald	JADA	Omega	P-RIO	uC++
CarlOS	EPL	WWWinda	OpenMP	Prospero	UNITY
Cashmere	Excalibur	ISETL-Linda	Orca	Proteus	UC
C4	Express	ParLin	OOF90	QPC++	V
CC++	Falcon	Eilean	P++	PVM	ViC*
Chu	Filaments	P4-Linda	P3L	PSI	Visifold V-NUS
Charlotte	FM	POSYBL	Pablo	PSDM	VPE
Charm	FLASH	Objective-Linda	PADE	Quake	Win32 threads
Charm++	The FORCE	LiPS	PADRE	Quark	WinPar
Cid	Fork	Locust	Panda	Quick Threads	XENOOPS
Cilk	Fortran-M	Lparx	Papers	Sage++	XPC
CM-Fortran	FX	Lucid	AFAPI.	SCANDAL	Zounds
Converse	GA	Maisie	Para++	SAM	ZPL
Code	GAMMA	Manifold	Paradigm		
COOL	Glenda				

Models from the golden age of parallel programming (~95)

Choice overload: Too many options can hurt you



- The Draeger Grocery Store experiment consumer choice :
 - Two Jam-displays with coupon's for purchase discount.
 - 24 different Jam's
 - 6 different Jam's
 - How many stopped by to try samples at the display?
 - Of those who "tried", how many bought jam?



Programmers don't need a glut of options ... just give us something that works OK on every platform we care about. Give us a decent standard and we'll do the rest

The findings from this study show that an extensive array of options can at first seem highly appealing to consumers, yet can reduce their subsequent motivation to purchase the product.

Iyengar, Sheena S., & Lepper, Mark (2000). When choice is demotivating: Can one desire too much of a good thing? *Journal of Personality and Social Psychology*, 76, 995-1006.



Less is more

- It's a "good thing" that for the last 10 years parallel computing has been dominated by a small collection of APIs/languages
 - OpenMP
 - MPI
 - Pthreads
- But computer scientists like creating new languages ... we are slipping back into chaos
 - OpenCL
 - PLINQ
 - TBB
 - CUDA
 - Fortress
 - Cilk++
 - Chapel
 - Ct
 - CnC
 - X10
 - Charm++
 - ... and many many more!

This is good for computer science research
but BAD for programmers ... we just want to
"write once, run everywhere"

Solution

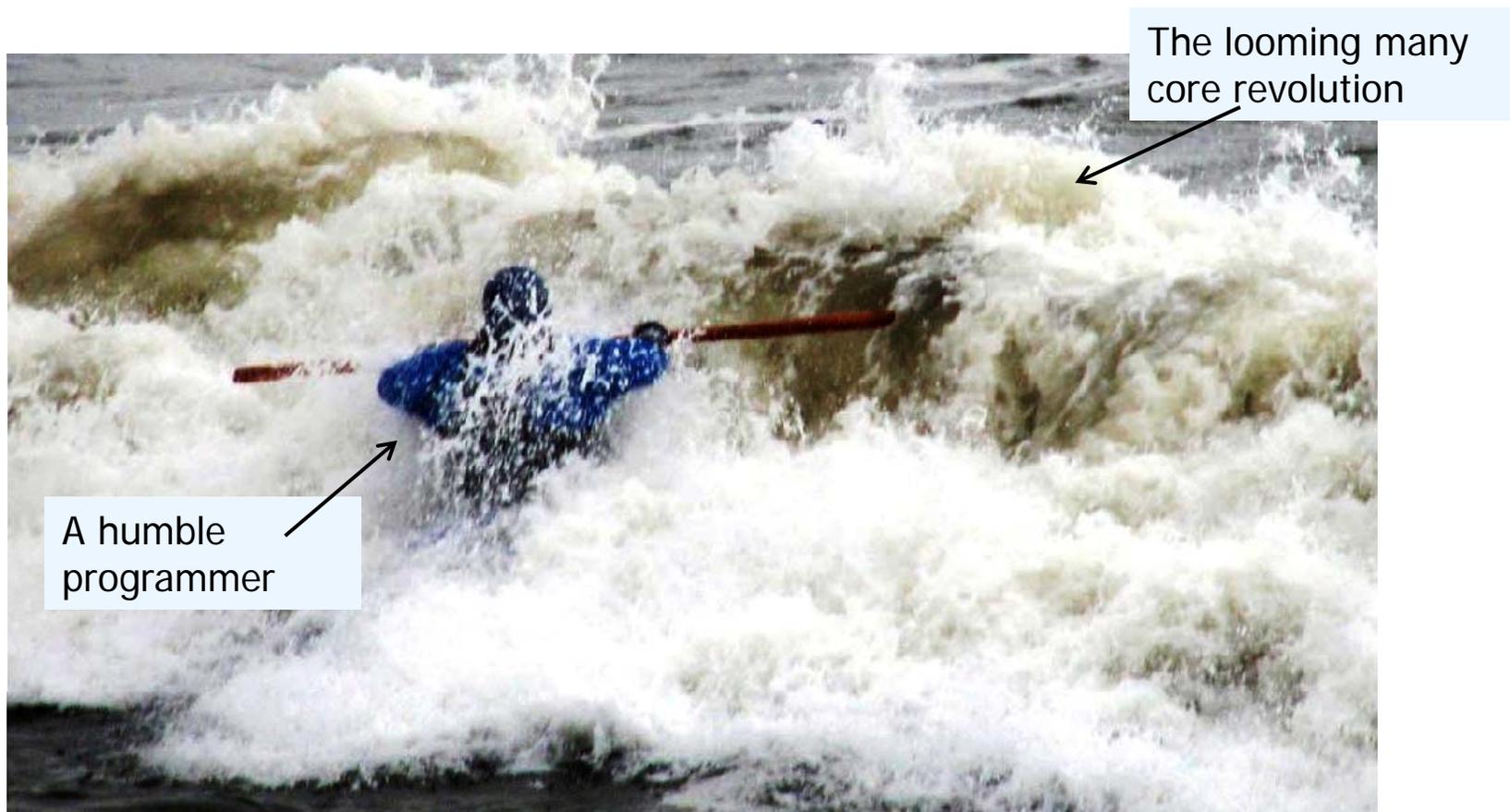


- Application Programmers need to fight back:
 - Demand a small number of industry standards ... refuse to use anything else!
- Computer scientists ... please spend more time figuring out how to make what we have actually work!
 - Study algorithms and how they map onto different models and platforms (Design Patterns)
 - Tools and frameworks that make programmers more productive with existing languages
 - New ideas extending existing parallel languages to make them better.

Conclusion



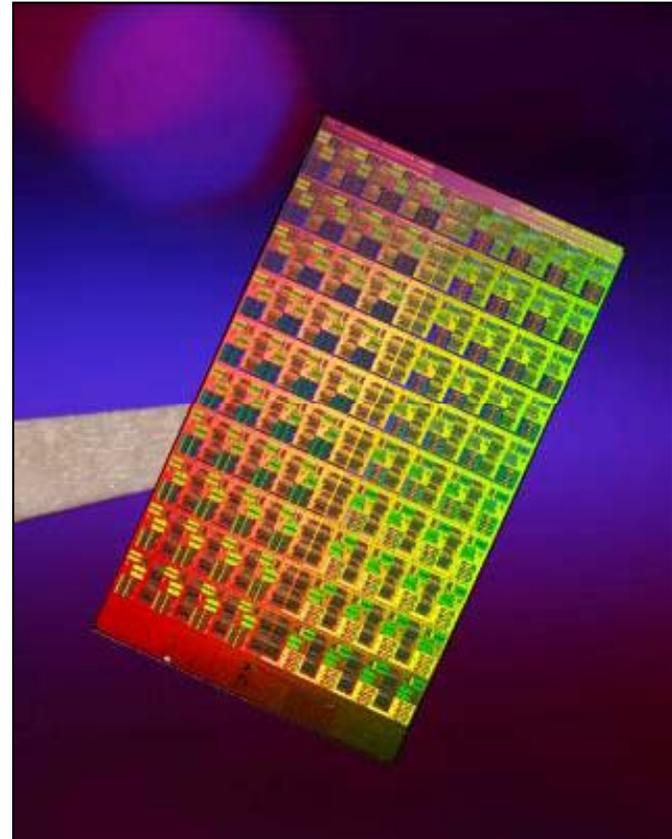
- The many core challenge is a software problem.
 - If you want to have an impact, focus on software.
- Computer scientists will screw it up if application programmers let them.
 - We need the right standards ... to create a software industry to support future heterogenous (many core) platforms



Backup Slides

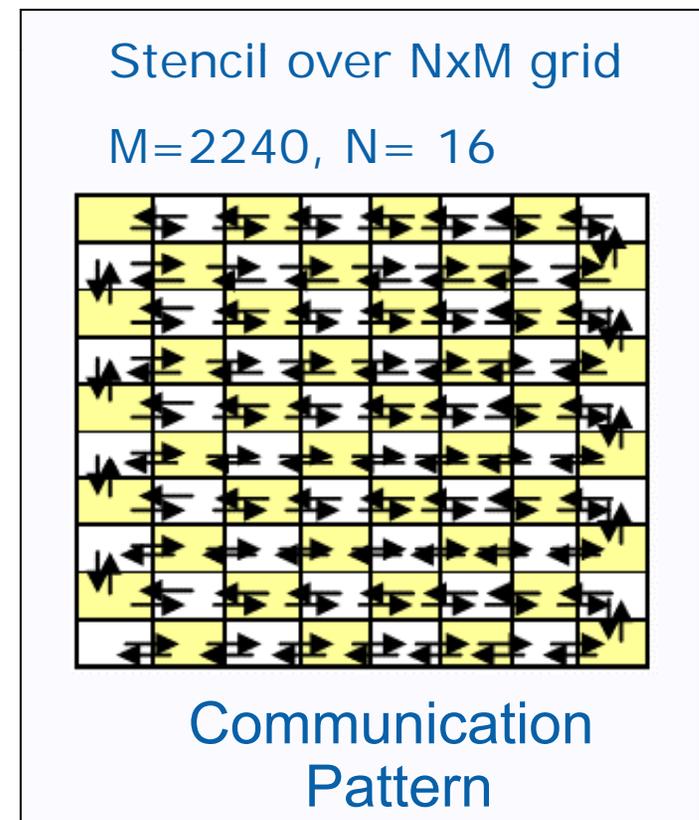
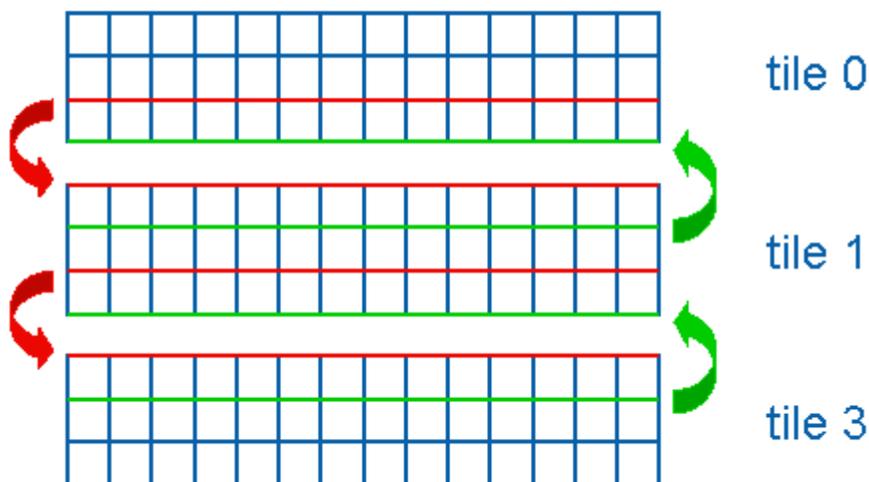


- ➔ • Details on 80 core processor application kernels
- Extra details on SCC
- More on RCCE



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



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

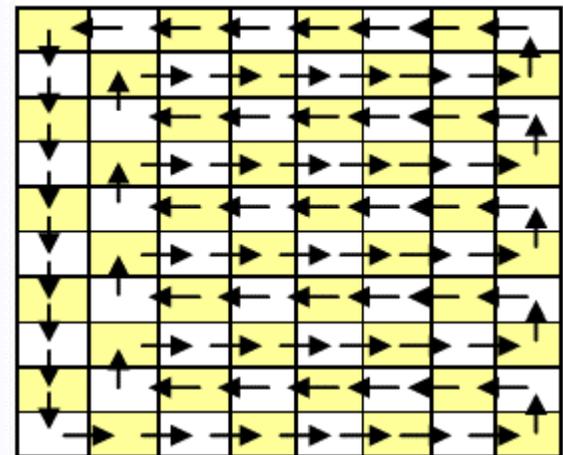
```

On core number i
Loop over j = 1, M
{
  Cij = dot_product (row Ai * column Bj)
  Circular shift column Bj 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

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

$$N = 80, M = 206$$



Communication
Pattern

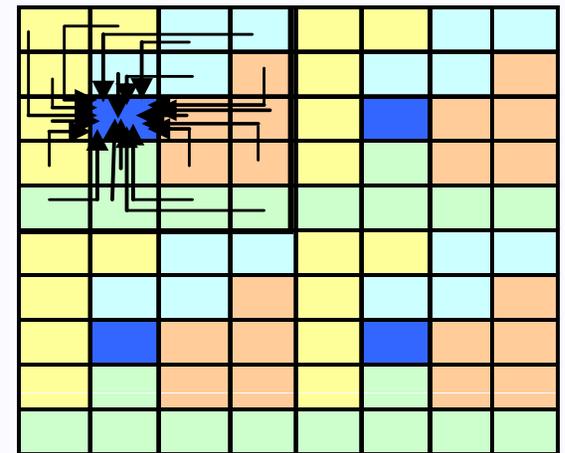
Spreadsheet

- 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

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.

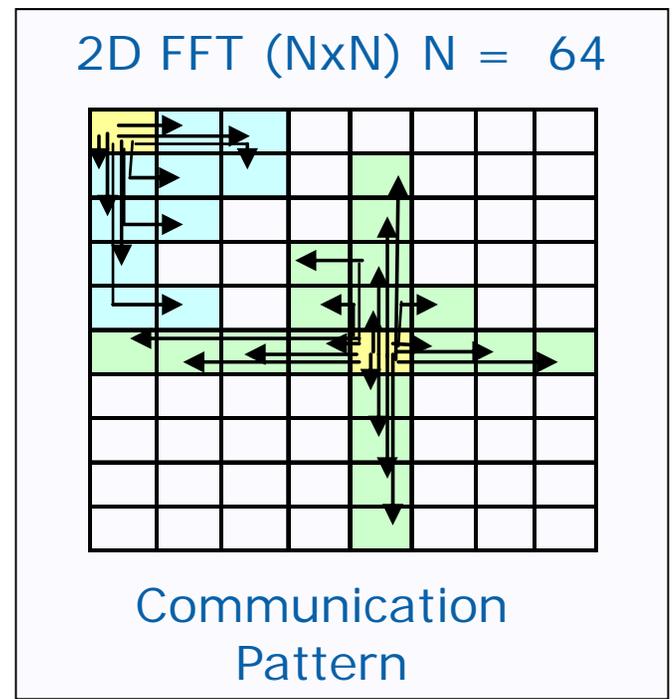
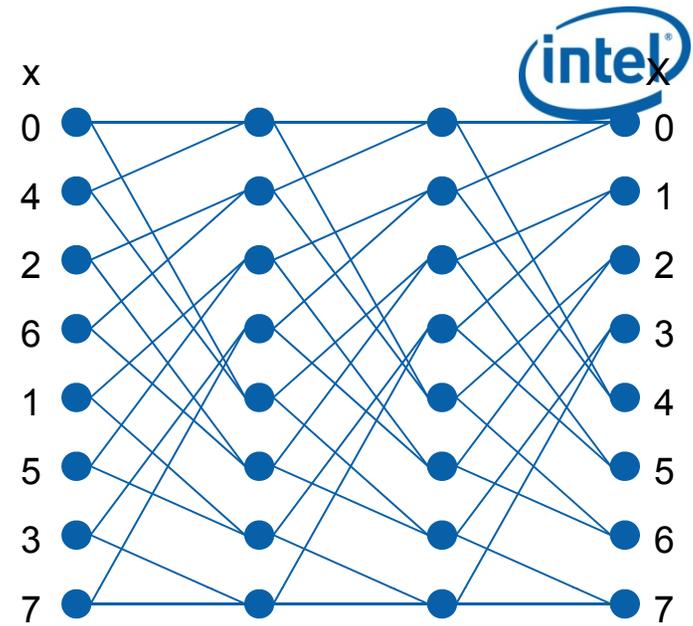
$L \times N$ table of
value/weight pairs.
 $N = 10, L = 1600$



Communication
Pattern

2D FFT

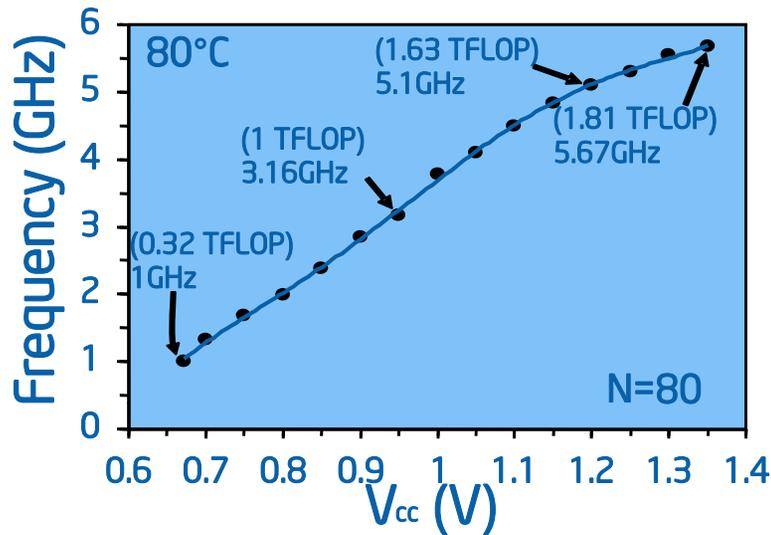
- 64 Point 2D FFT on an 8 by 8 Grid.
- Pease Algorithm
 - “Peers” in each phase are constant ... a constant communication pattern throughout the computation.
- Parallelization:
 - Basic operation FFT of 64 long vector along a column of 8 tiles
 - FFT of 8-long vector in each tile
 - Communication:
 -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



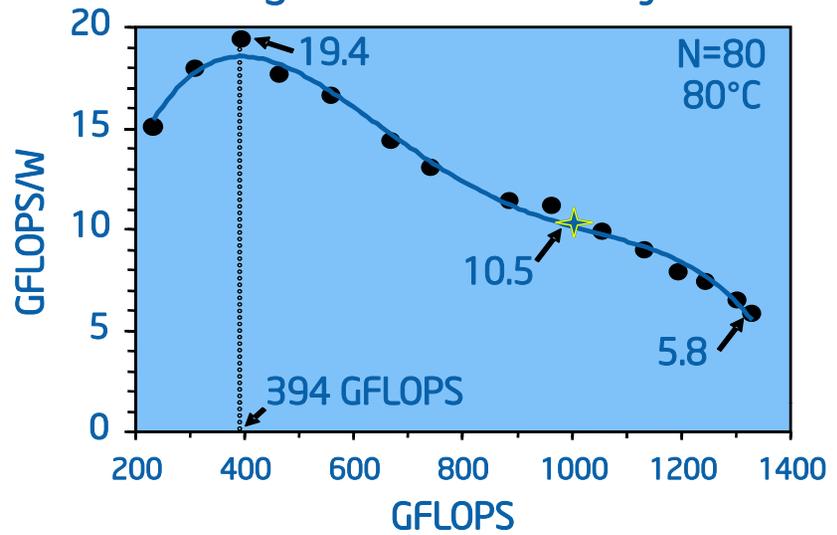
Power Performance Results



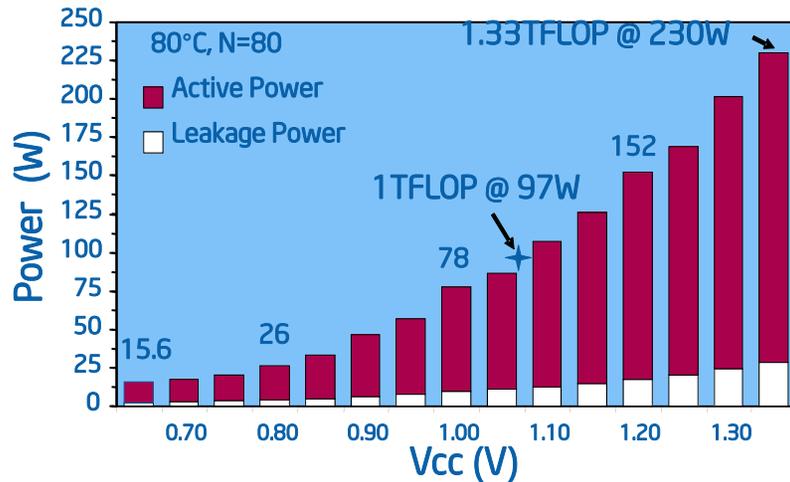
Peak Performance



Average Power Efficiency

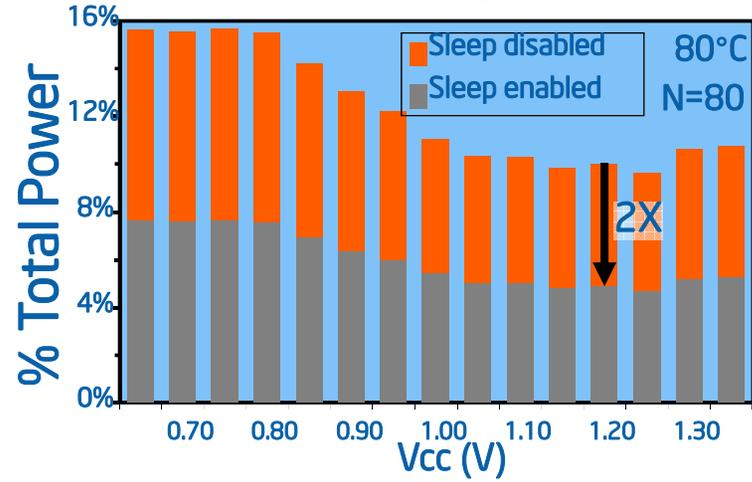


Measured Power



Stencil: 1TFLOP @ 97W, 1.07V;

Leakage

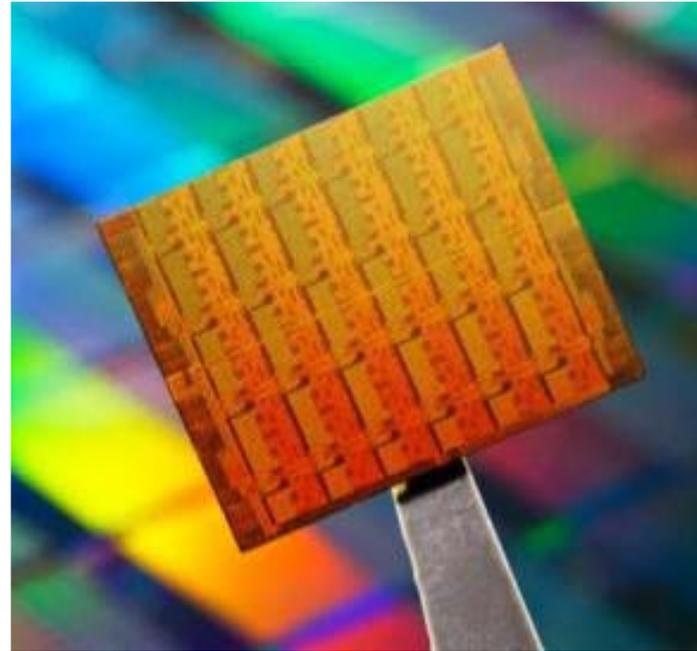


All tiles awake/asleep

Backup Slides



- Details on 80 core processor application kernels.
- ➔ • Extra details on SCC
- More on RCCE

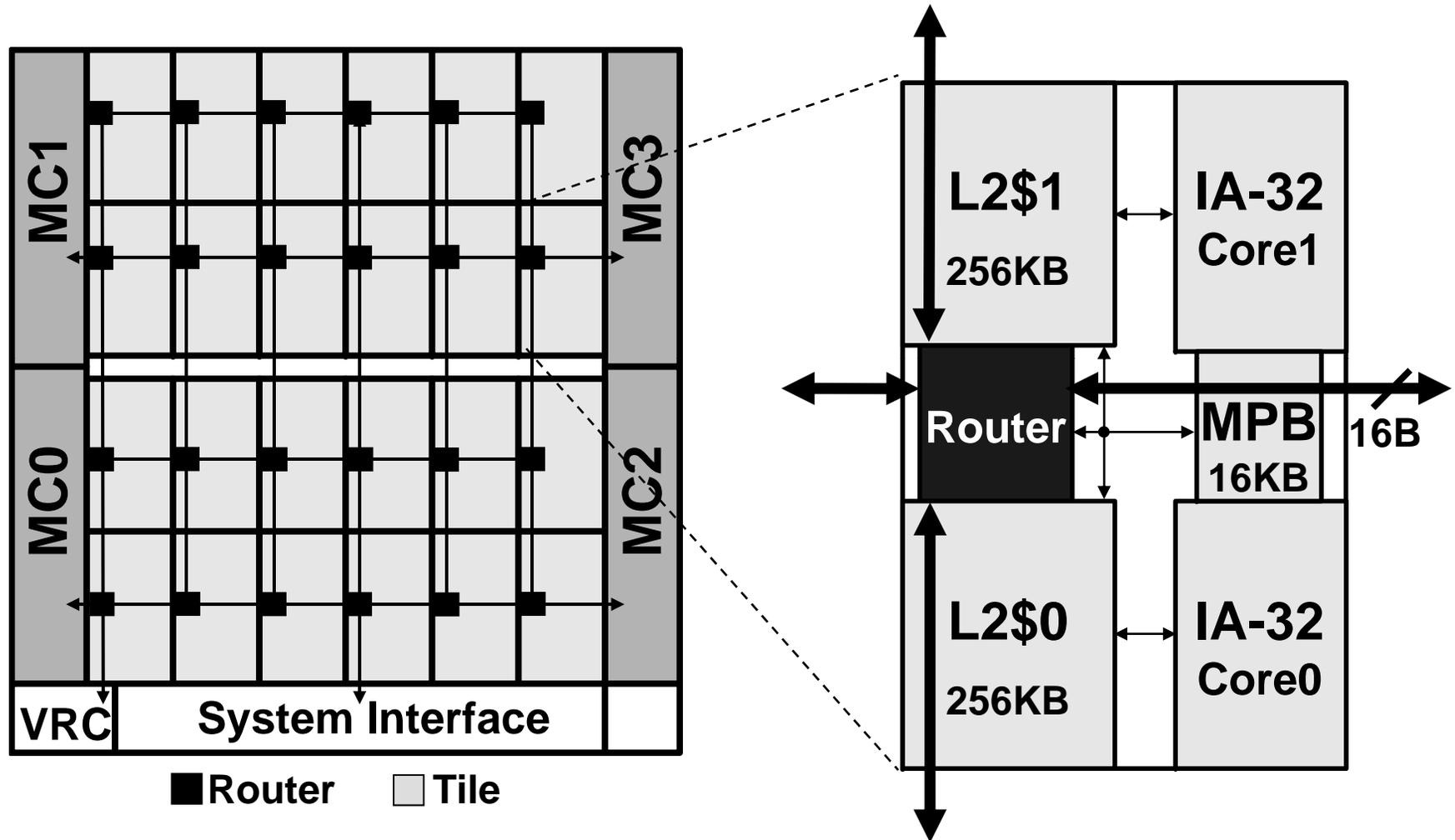


SCC Feature set



- First Si with 48 iA cores on a single die
- Power envelope 125W Core @1GHz, Mesh @2GHz
- Message passing architecture
 - No coherent shared memory
 - Proof of Concept for scalable solution for many core
- Next generation 2D mesh interconnect
 - Bisection B/W 1.5Tb/s to 2Tb/s, avg. power 6W to 12W
- Fine grain dynamic power management
 - Off-die VRs

Die Architecture



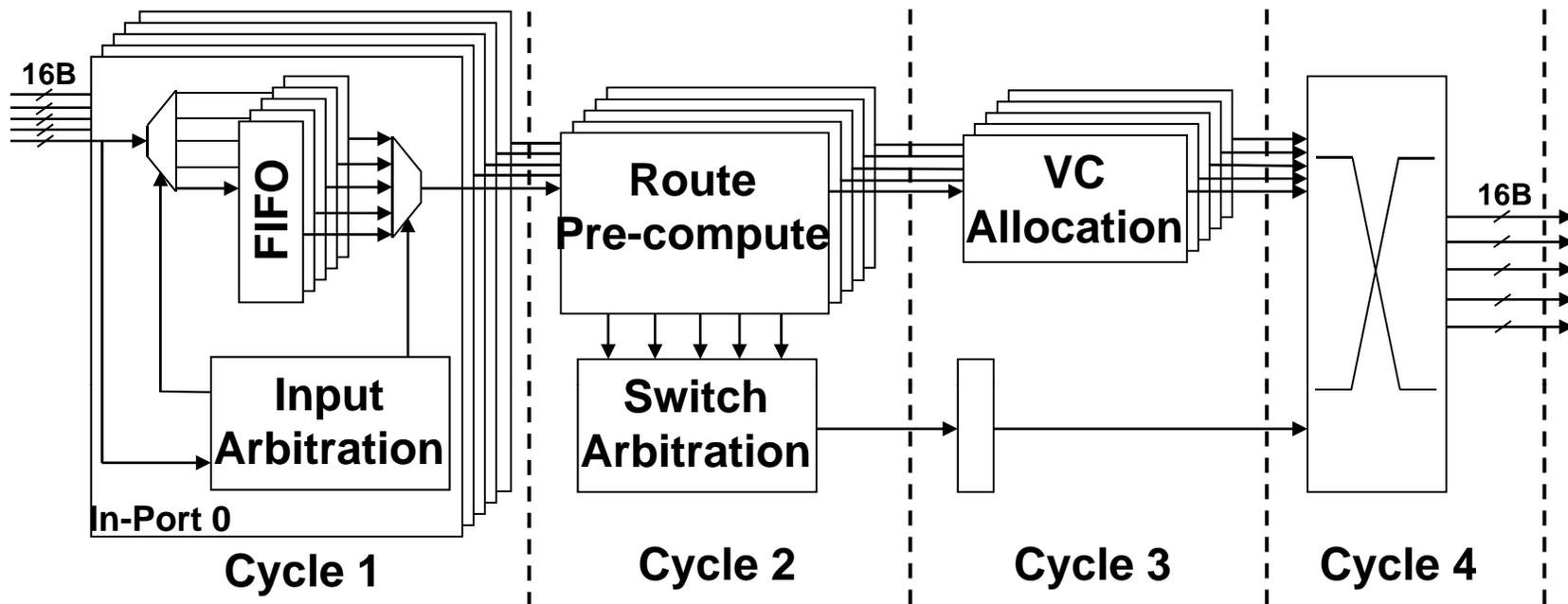
2 core clusters in 6x4 2-D mesh



On-Die 2D Mesh

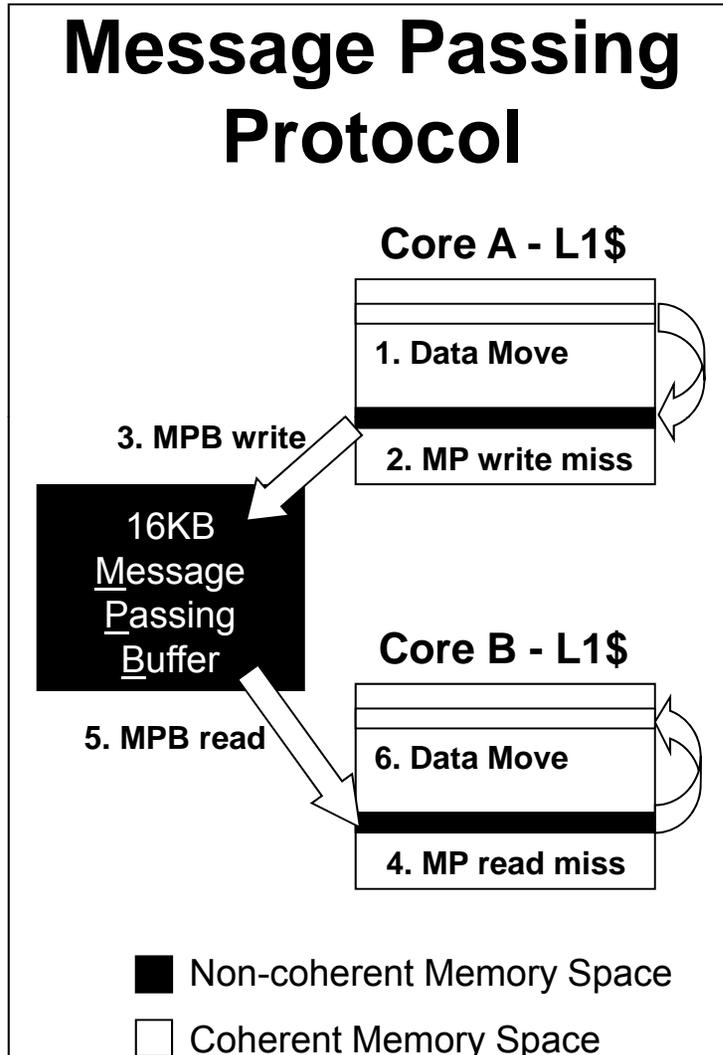
- 16B wide data links + 2B sideband
 - Target frequency: 2GHz
 - Bisection bandwidth: 1.5Tb/s to 2Tb/s, avg. power 6W to 12W
 - Latency: 4 cycles (2ns)
- 2 message classes and 8 VCs
- Low power circuit techniques
 - Sleep, clock gating, voltage control, low power RF
 - Low power 5 port crossbar design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

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

Message Passing Protocol



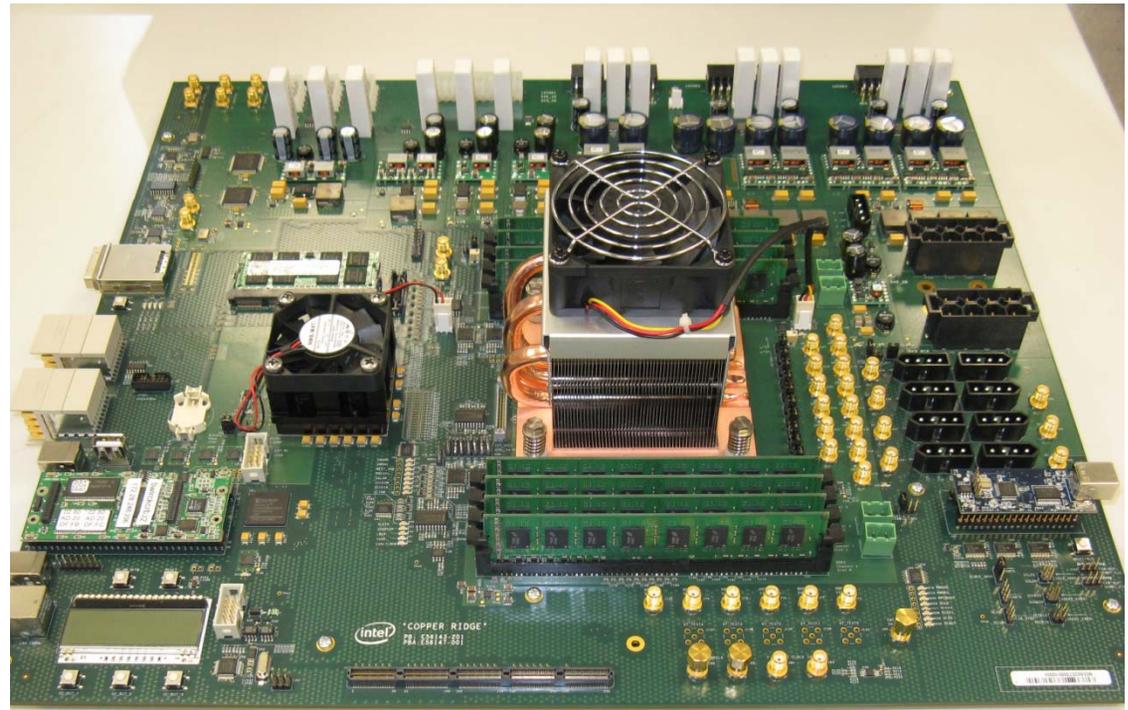
- Cores communicate through small fast messages
 - L1 to L1 data transfers
 - New Message Passing Data Type (MPDT)
- Message passing Buffer (MPB) – 16KB
 - 1 MPB per tile for 384KB of on-die shared memory
 - MPB size coincides with L1 caches

System Interface



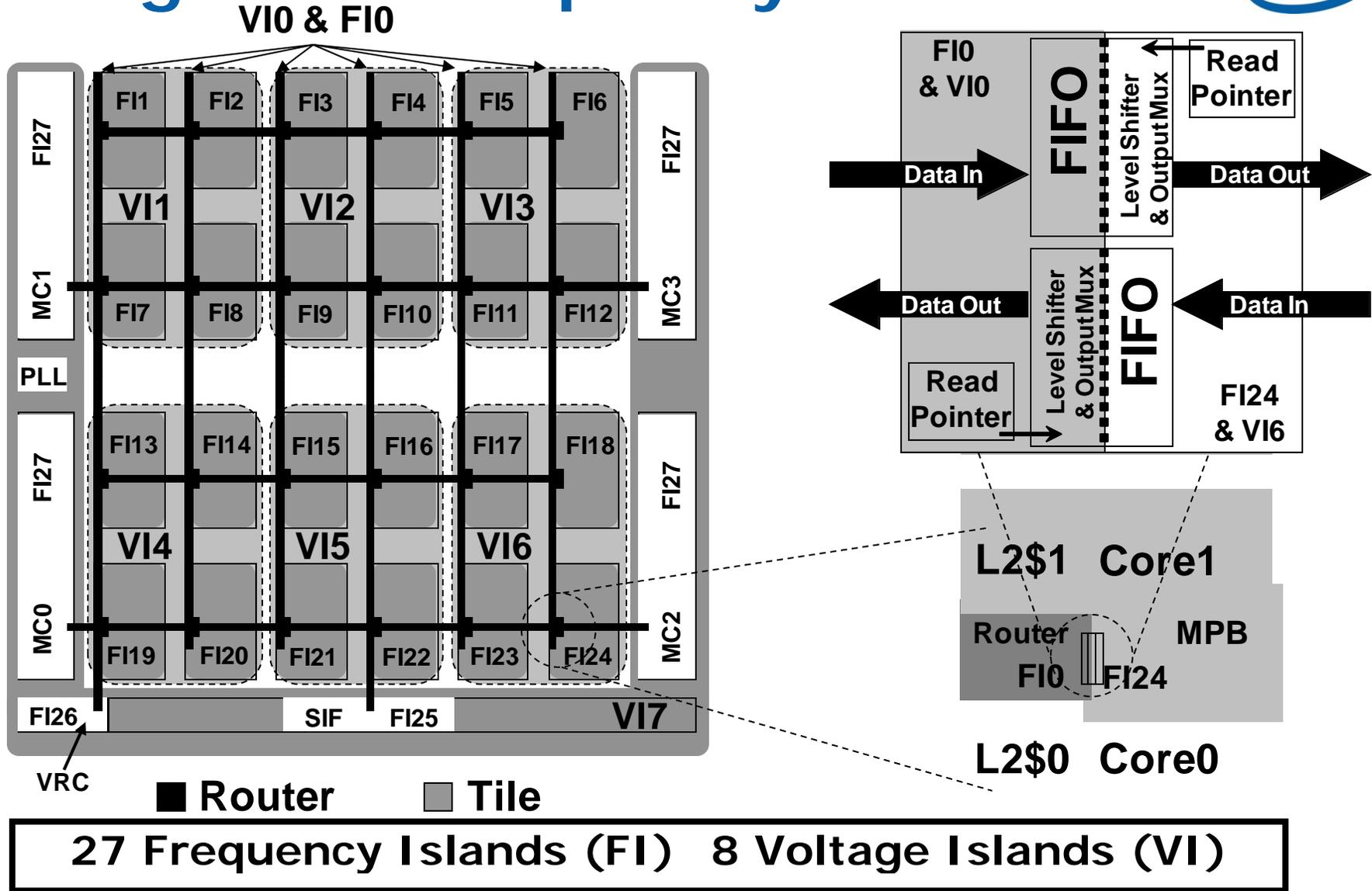
- JTAG access to config system while in reset/debug
 - Done on Power Reset from Management Console PC
 - Configuring memory controller etc.
 - Reset cores with default configuration
- Management Console PC can use Mem-mapped registers to modify default behavior
 - Configuration and voltage control registers
 - Message passing buffers
 - Memory mapping
- Preload image and reset rather than PC bootstrap
 - BIOS & firmware a work in progress

Package and Test Board



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

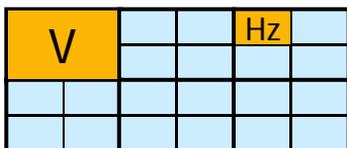
Voltage and Frequency islands



Power Management



- Rock Creek lets the programmer change V(oltage) independently on 4 tiles (8 cores).
- Rock Creek lets the programmer change F(requency) independently on tile (2 cores).
- RCCE supports safe mode: vary $V + F$ within 4-tile power domain.
 - Master core sets $V + F$ for all cores in domain.
 - RCCE_istep_power(): steps up or down $V + F$, where F is max for selected V. If V up, wait until target V reached before increasing F. If V down, first decrease F
 - RCCE_wait_power(): returns when power change is done
 - RCCE_step_frequency(): steps up or down only F
- Issues:
 - ✓ Latency of ΔV commands high ($\sim 1M$ cycles)
 - ✓ Multiple cores in same V domain; must operate cores in synchronized fashion to guarantee shared power benefits
 - ✓ Access to RCK Power Controller (RPC) serialized; cores block on multiple simultaneous RPC requests
 - ☂ Large variability among cores of F_{max} for given V





Official SSC reference

“A 48-Core IA-32 Message Passing Processor with DVFS in 45nm CMOS”, ISSCC 2010.

J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Paillet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, T. Mattson

• Abstract:

- A 567mm² processor in 45nm CMOS integrates 48 IA-32 cores and 4 DDR3 channels in a 6×4 2D-mesh network. Cores communicate through message passing using 384KB of on-die shared memory. Fine grain power management takes advantage of 8 voltage and 28 frequency islands to allow independent DVFS of cores and mesh. As performance scales, the processor dissipates between 25W and 125W.

Backup Slides



- Details on 80 core processor application kernels.
- Extra details on SCC
- ➔ • More on RCCE



Rapidly Communicating Cores Env.
 Reduced Compact Communication Environment
 Research Cores Communication Environment
 Rabble-of Communicating Cores Experiments

R C C E

A small library for many-core communication

Rob van der Wijngaart (Software and Services Group)

Tim Mattson (Intel Labs)

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_rcv(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_rcv(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);
}
```

offsets to "remote" MPB

```
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);}
```

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 (\approx 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 Implementation Details:



Physical core IDs, tile IDs, x-, y, z-coordinates and the 6 power domains



RCCE: Supporting Details



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



RCCE vs MPI

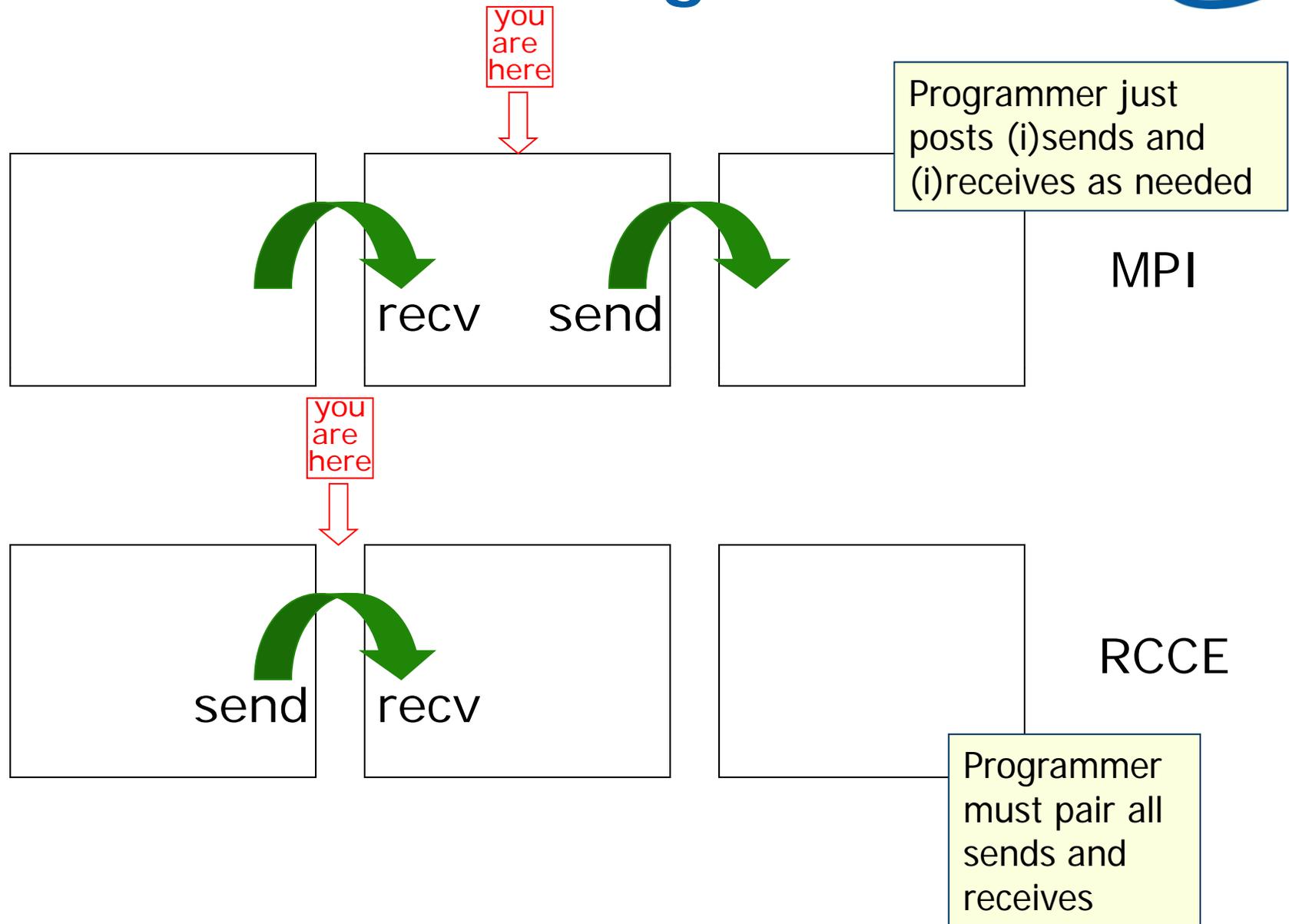
- No opaque data types in RCCE, so no MPI-style handles, only pointers
- No RCCE_datatype, except for reductions
- No communicators, except in collective communications
- Only synchronous communications
 - + No message bookkeeping
 - No overlap of computations/communications
 - Deadlock?
- RCCE has low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



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