# Template design

CCP A.A. 2008-2009
M. Danelutto
Aprile 2009

# Contents

- Definition of template
- Template design
  - Separation of concerns
  - Existing technology & theoretical result re-use
  - Template engineering
- Advanced techniques
  - Fault tolerance, Security, Green computing, ...

# Definition of template

- **Parametric configuration of parallel activities computing a well known parallel pattern**
  - **configuration**: precise pattern of parallel activities
  - **parallel activities**: processes, threads, co-processor activities (e.g. GPU)
- **computing a well known pattern**: definitely bound to the pattern (skeleton, design pattern, ...)
- **parametric**:
  - parallelism degree, “worker” code, policies (scheduling, security, fault tolerance, ...)

# Definition of template (cont'd)

- **with its own performance model provided**
- **performance model**:
  - either throughput or latency or both
  - defined in terms of the performance models of the parameters
  - defined in terms of basic architectural parameters
  - usually approximated
  - gives a rough idea of the actual performance to be achieved
  - simple (must be used at run time)

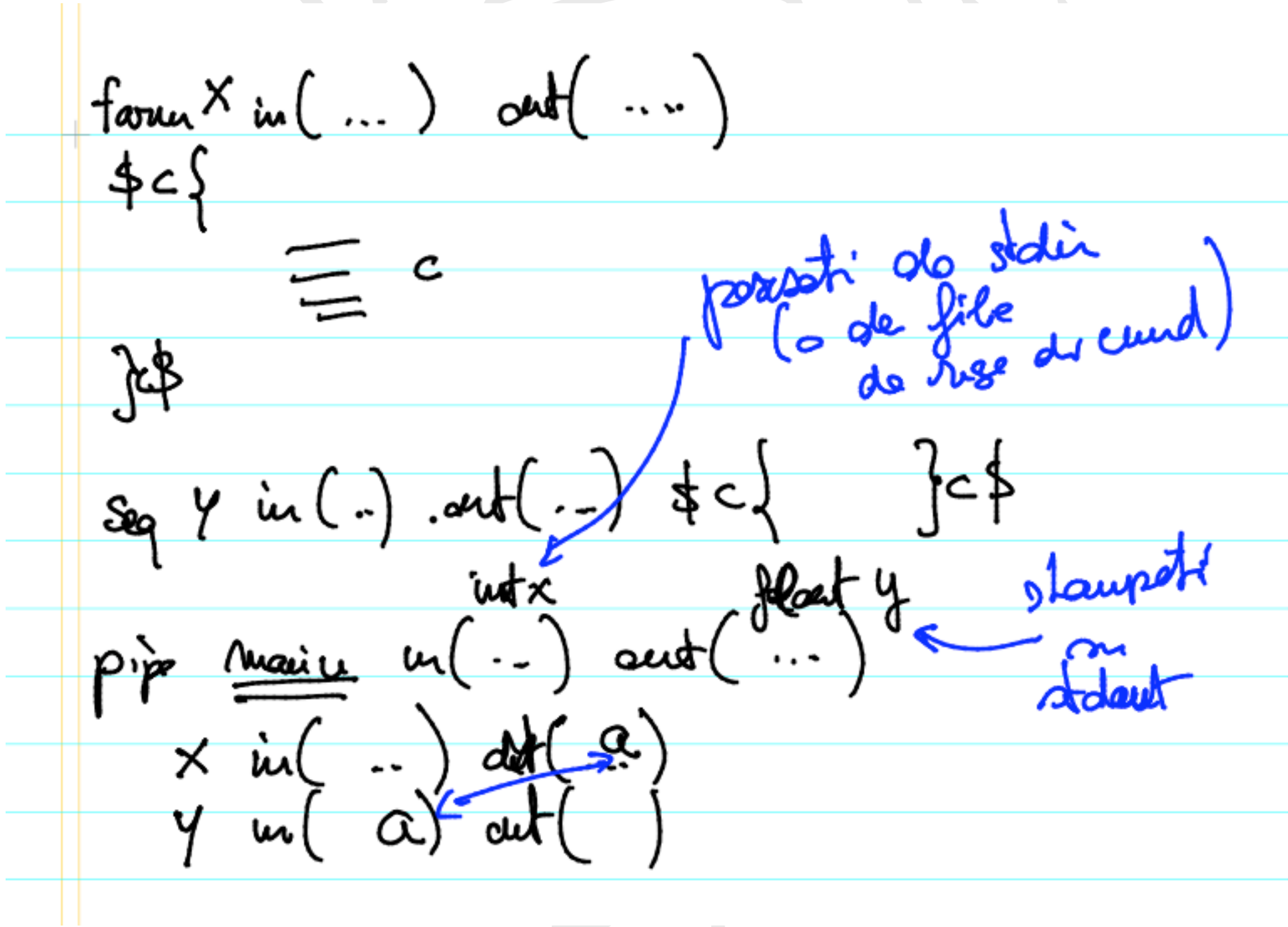
# Template usage

- In a template based structured parallel programming environment
  - parsing of the program → skeleton tree
  - ∀ skeleton: map skeleton to a template
    - composition of the chosen templates
    - optimization of the template composition
    - target architecture code generation
- E.g. in: P3L (anacleto, SkIE, ASSIST), Muesli, eSkel, SkeTo, ... (all but Lithium, muskel, Skipper, Calcium)

# Template usage (parsing)

- Two alternatives
  - declarative approach
    - traditional parsing → <abstract syntax, code parameters>
  - library approach
    - skletons ≈ library calls
      - library code builds skeleton template
- Clearly
  - declarative approach can be a much more efficient solution
  - library approach + JIT ⇒ comparable results

# Sample declarative syntax (P3L)



# Sample library style (muskel)

```
Skeleton s1 = new Stadio1( ... );
Skeleton f1 = new Farm(s1);
Skeleton s2 = new Stadio2( ... );
Skeleton main = Pipeline(f1,s2);
```

} dichiarazione

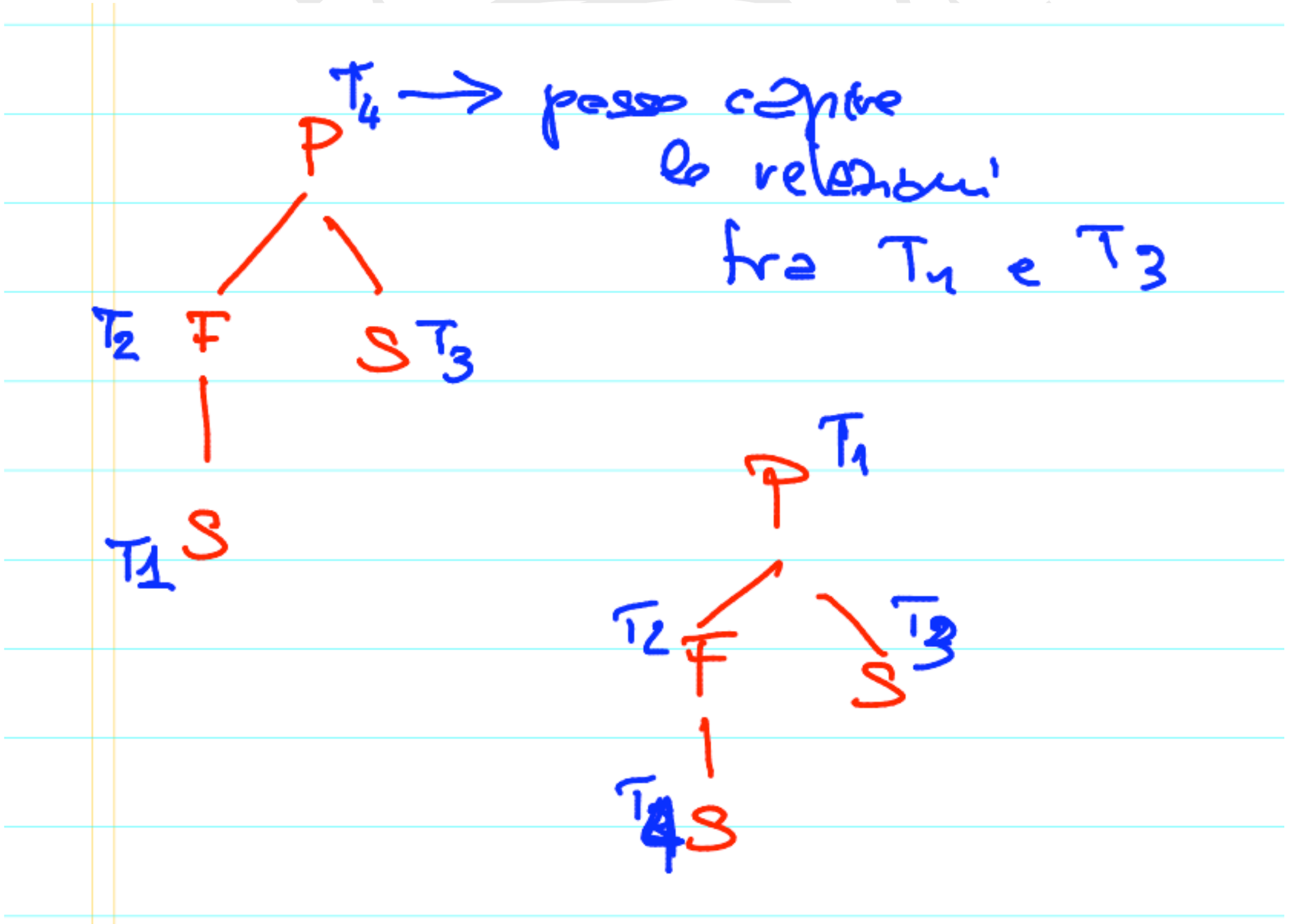
```
Manager mgr = new Manager(... , main, ...)
mgr.run();
```

← "partenza" del programma

# Template (skeleton to template map)

- alternative possibilities
  - to be analyzed
    - analysis *in insulation* often does not lead to good results
    - analysis *in toto* requires post processing of the mapping (with backtracking)
- alternatives from parameter fixing
  - e.g. policies
- alternatives from completely different templates
  - disappearing due to architecture targetting
    - e.g. multithreaded pattern on COW/NOW/Grid

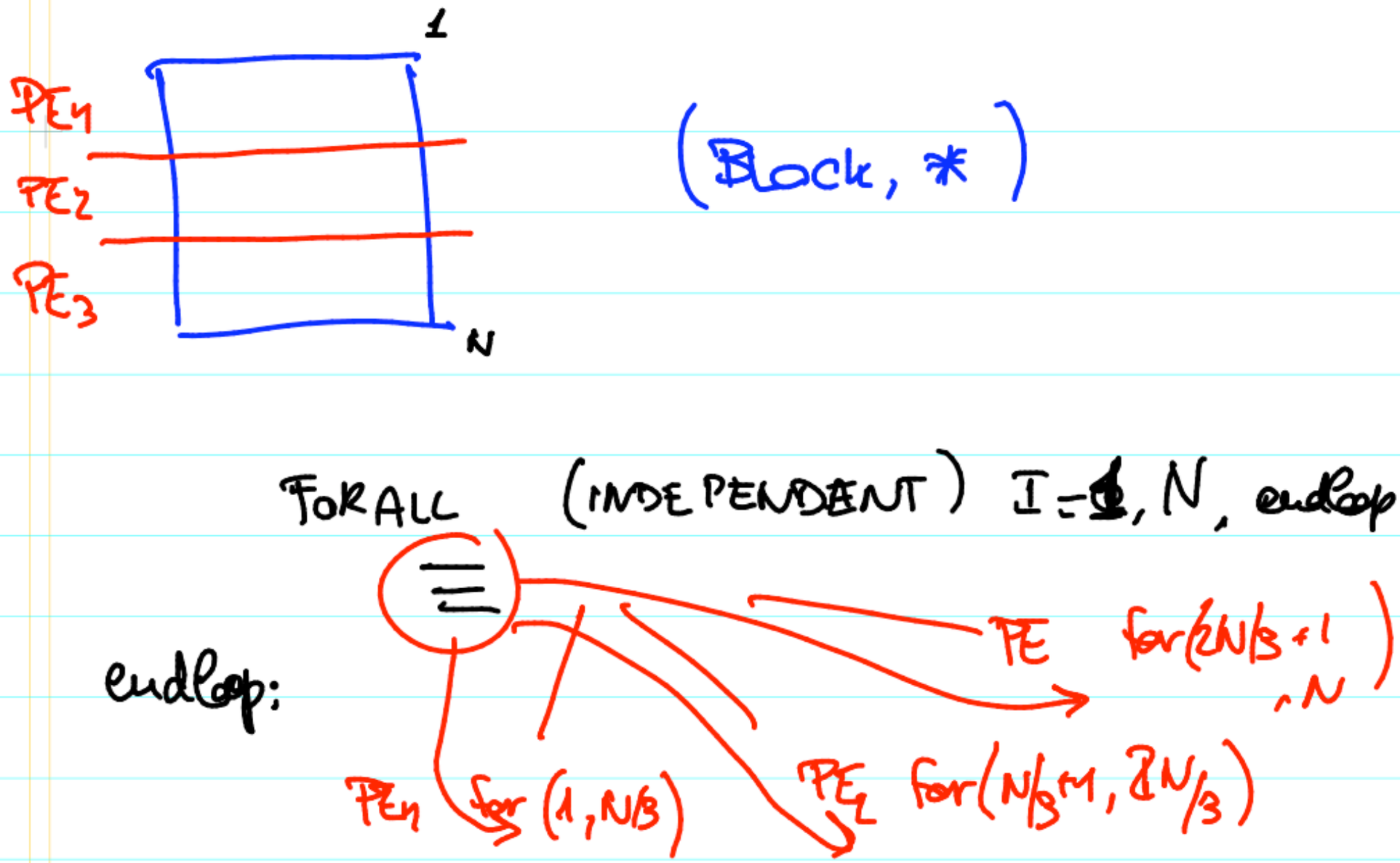
# Sample template assigment



# Template (composition)

- Alternatives:
  - composition through *universal connectors*
    - e.g. all templates have a single in-chan & out-chan
  - composition through *context dependent adaptors*
    - e.g. map template after map template
      - adaptor → remove gather+distribute again
    - known sample in the past
      - optimization of consecutive FORALL in HPF (late ’90)

# HPF forall


1
N
PE1
PE2
PE3
(Block, *)
FORALL (INDEPENDENT) I=1, N, endloop
endloop:
PE1 for (1, N/3)
PE2 for (N/3+1, 2N/3)
PE for (2N/3+1, N)

# Template (composition optim)

- Needed in case of *universal connectors*
  - pipe(farm,farm)
    - collapse collector of the first farm and emitter of the second one
      - communications in memory rather than actual inter PE communications
- Can be used also when *context dependent adaptors* are used
  - to post-fix details
    - map after map → change the computation schema of the first map to accomplish distribution needed in the second

# Template (code generation)

- two phase process
  - template code generated out of the templates
    - ∀ target architecture needs a template code library
  - template code specialization and assembly
    - requires global optimization as usual in code generation
- e.g.
  - template code in HLL (e.g. C, C++)
  - target code with plain **gcc -Ox** of the template code

# Performance model

- Already covered in previous part(s) of this course
- Performance model is needed
  - to assess regular progress of the parallel computation
    - at run time
  - to devise suitable parallelism degree in templates
    - at compile / deploy time

# Performance model (2)

- Exact performance models
  - should take into account **a lot of** parameters concerning
    - parallel target machine, processors, memory, application parameters, etc.
  - are complicate to compute
  - small errors can lead to completely wrong decisions
    - both at compile and at run time
  - in general are complicate to compute
    - several distinct probes + complex formulas

# Performance model (2)

- Approximate performance models
  - easier to devise and compute
  - approximate enough to lead to reasonable results
    - at run time as well as at compile / deployment time
- Usually derived
  - separating phases and applying pipeline results
    - balance performance among stages
  - evaluating parallel phases according to the farm models
    - devise the most suitable parallelism degree

# Performance model (3)

- Different according to the measure taken into account
  - throughput
    - minimize stage service time
  - latency
    - minimize stage latency

# Template design principles (1 of 3)

- **Separation of concerns**
  - in a template:
    - data distribution/collection
    - code deployment
    - parallel activity computations
- all require dedicated and peculiar techniques
  - to improve efficiency
  - to exploit target architecture features

# Template design principles (2 of 3)

- **existing knowledge re-usage**
  - technology
    - e.g. MPI, RMI, ...
    - e.g. non linear broadcast algorithms, ...
  - theoretical
    - e.g. transformation rules (map(pipe) = pipe(map))
- this is suggested **and** needed
  - to avoid duplicate work and effort
  - to compare with state-of-the-art solutions

# Template design principles (3 of 3)

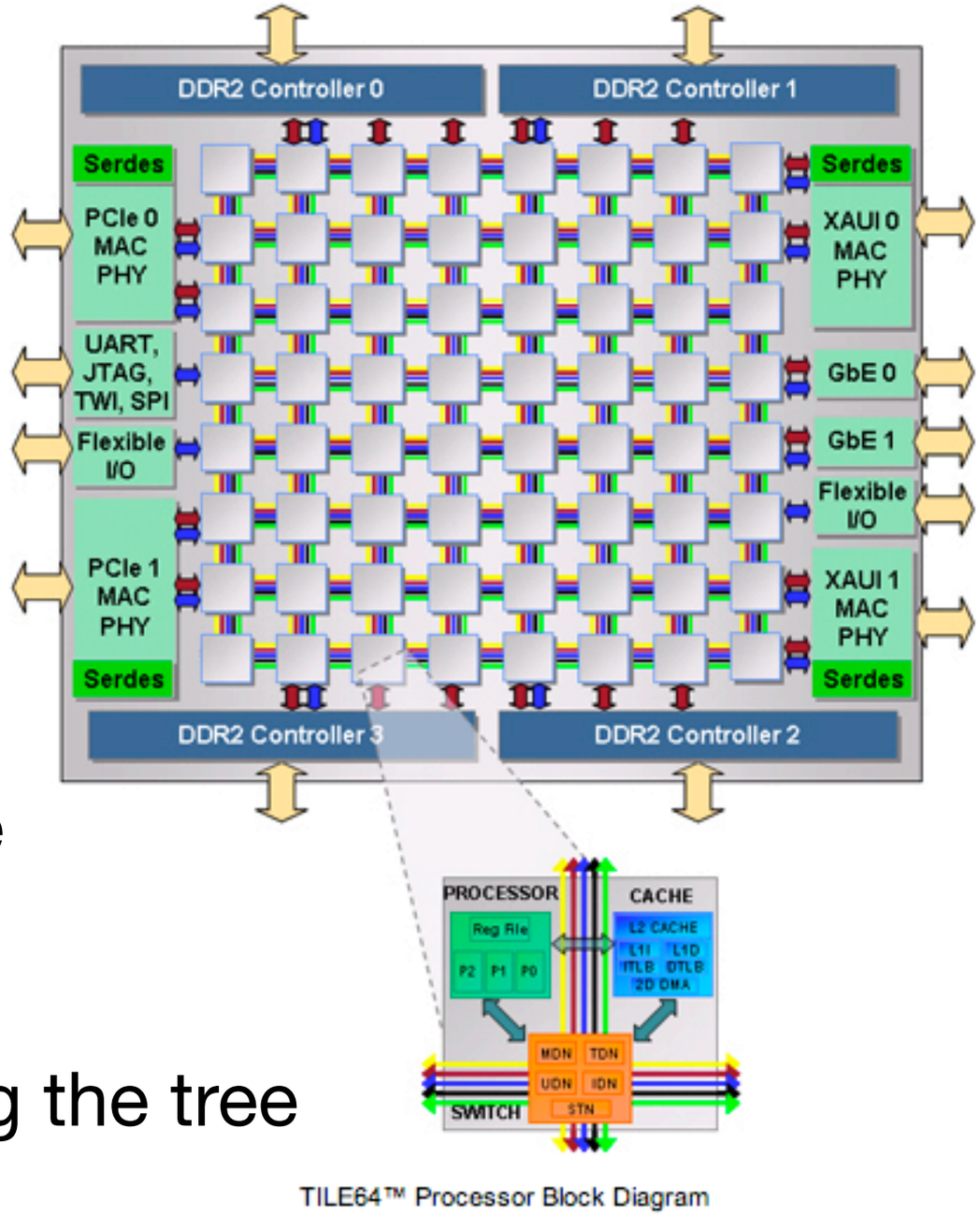
- **template engineering**
  - usage of software engineering techniques
    - template design
      - requirement analysis, specification, ...
    - design pattern exploitation
      - façade, factory, ...
  - verification & validation of the results
    - full V&V → *hard !*
    - testing can be performed much more simply (JUnit)

# Separation of concerns: sample (A)

- Data distribution
  - a number of independent activities to be performed at a number of remote machines
  - each independent activity needs a partition of the input data
- Impact of the interconnection network
  - broadcast media (Ethernet-like)
    - 1 communication at a time involving any of the PEs
    - linear partition distribution time
      - Emitter schedules partitions to all the Workers

# Separation of concerns: sample (A.2)

- torus (e.g. Intel 80 core chip, Tilera Tile64)
  - any number of independent communication not involving the same PEs (cores)
    - linear partition distribution
      - systolic “train” of packets through workers, then compute
- fat tree (e.g. QSW networks)
  - several independent comms along the tree
  - log partition distribution




TILE64™ Processor Block Diagram

# Separation of concerns: sample (A.3)

- Distribution strategies are independent of the way used to compute in parallel
- Not actually independent on the collection strategy possibly needed to deliver final results
- therefore
  - should be analyzed and implemented independently
  - to increase efficiency
  - to exploit target architecture features

# Separation of concerns: sample (B)

- Code deployment
  - usually target nodes are not dedicated
  - code to be executed is to be deployed each time
- Several optimizations here
  - broadcasting techniques
    - improves efficiency (with Ethernet O(1) vs. O(nw) !)
  - code caching
    - improves efficiency of re-running code
    - provided an adequate amount of main store is available

# Existing knowledge re-usage: sample (A)

- Scheduling tasks to workers
  - round robin (Wi takes i, i+nw, i+2nw, ...)
    - static scheduling very effective if:
      - same length tasks (→ load balancing)
      - effective buffering (→communication hiding)
- very inefficient
  - in presence of variable length tasks
    - burst of “long” tasks at the end of Wi
      → greatly impair efficiency

# Existing knowledge re-usage: sample (A.1)

- auto scheduling
  - Wi asks task when idle
    - dynamic scheduling
      - perfect load balancing
        - inefficient in masking communications
- auto scheduling with pre-fecth
  - Wi asks task while computing previous task
    - slightly poorer load balancing
    - quite efficient communication hiding

# Knowledge re-usage: sample (B)

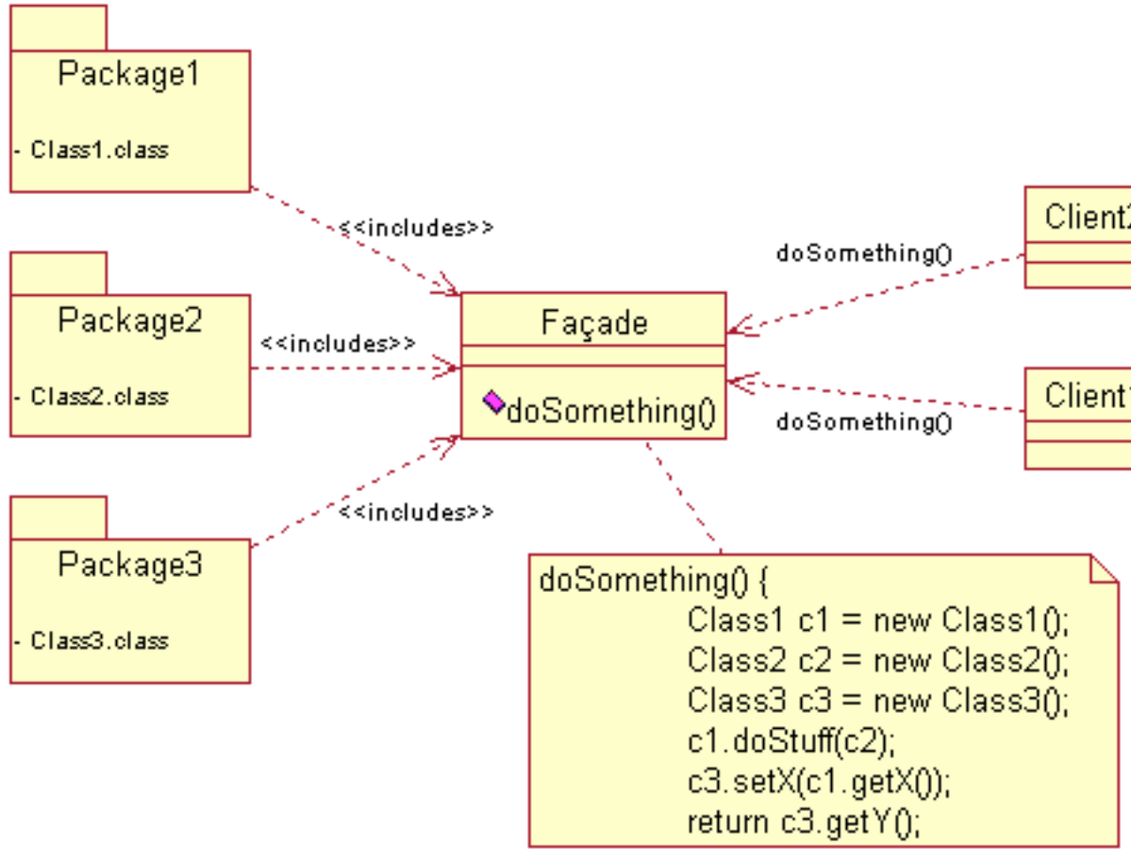
- Communications on a broadcast network
- Communication grouping
  - many small data items to be sent to the same dest PE
  - 1 single communication vs. n communications
    → greatly improves efficiency
- can be used when delivering results to the collector in a map
  - all results needed *before* being delivered onto collector output stream
  - needs mechanisms to assess termination (send anyway on End-Of-Stream)

# Knowledge re-usage: sample (B.1)

- Immediate communications
  - Nagle algorithm in TCP/IP actually groups small packets
  - needs to be disabled in certain cases
    - e.g. telnet
    - e.g. when synchronization messages are involved
      - immediate delivery is needed
      - otherwise computation timing may be impaired

# Template engineering sample (A)

- Template code implemented through a façade pattern
  - decouple physical implementation from interface
  - allows the template to be used in several different contexts



# Template engineering sample (B)

- Extreme programming techniques
  - used in well known, different (w.r.t. structured parallel programming) contexts
- some techniques may be inherited
  - e.g. Unit tests
    - each time you produce a class
    - you *must* produce also the unit test code
      - i.e. the code verifying the class is working as expected

# Template engineering sample (B)

Many books have already been written about automated testing, but very few of them pay attention to the question of how to organize such tests. As more tests are written, it becomes harder to know where to put a test or even what to call it. This has become a significant issue with the rise of test-driven development (TDD), which has been popularized by Extreme Programming (XP). You can think of TDD as "development through testing."

The major provisions of TDD are:

- Before writing any code fragment, an automated test must be written to check the functioning of this code. Since the code does not exist yet, it initially fails the test.
- After the test begins to pass, duplicate code must be removed.

Such an approach can be used by any programmer and does not require the use of a specific methodology. But before we get into writing tests, it would be desirable to first look at how to organize the automated tests.

There several different kinds of tests we should consider:

- **Unit tests**: These serve to check the correctness of several modules (i.e., classes). If the object needs access to some external data source, like a database, these are simulated with *mock objects*, but only in cases in which re-creating the actual data source would be difficult in a test environment.

- **Customer's tests**: These are functional, system, and acceptance tests. All of them check the behavior of the system as a whole. In the XP methodology, these tests are written by the customer, given the program skeleton.

- **Integration tests**: These are like a cross between the customer tests and unit tests. Integration tests help test the interaction of several levels of the application. Typically, mock objects are not used in integration testing, which may increase testing time. Also, integration tests often demand the presence of a special test environment, such as a database seeded with test data. Integration tests may also use special external libraries. An example of such a library for the integration of J2EE applications is Cactus. Explanation of these tests is going beyond of this article, and requires much detailed theory information, so you just take it as information that such kind of tests exists.

- **Developer's tests**: These are just tests that developers use to verify whole code, new pieces of code, and/or new functions. For each developer, it is important to generate new tests to check code whenever possible. Organizing these tests can be as important as organizing the code base itself.

# Template engineering sample (B.1)

- JUnit
  - package to support test development
  - and (multiple) test execution
  - unit tests
    - just test the single building blocks, not the overall behaviour of an application
- version 4. of JUnit
  - class A
  - class testA (@Test methods + assertTrue(expr))
  - javac -cp JUnit.jar testA.java
  - java -cp JUnit.jar JUnitCore testA
    - gives results: pass, not passing test due to ...

# Advanced techniques

- In general techniques related to non functional concerns
  - non functional concern ≡ concerns impacting how the results are computed rather than what is being computed
- why *advanced* ?
- initially
  - compete with hand written code ⇒ performance was the only concern in addition to expressive power
- then
  - restrictions on parallel patterns used ⇒ possibility to improve other aspects / concerns

# Advanced techniques (2)

- In particular
  - fault tolerance
    - development of *ad hoc* techniques much more effective and less intrusive than traditional ones
  - security
    - confine of security to places where it is precisely needed to avoid extra (unnecessary) overhead
  - green computing
    - recent “must” keyword
    - use watts only when actually needed

# Template: Fault tolerance

- Individuate the possible failures
  - evidenced due to the template structure
- Individuate the possible failure probes
  - heartbeat, connections (closed), ...
- Design feedback loop
  - probe failed → countermeasure
- Overall process efficiency improved due to the limited application field: the template

# Heartbeat technique

- *Controller*
  - periodically queries the controlled entity
    - e.g. sends a message and awaits an answer, pings the machine, calls a remote method/procedure, ...
- *Controlled*
  - sets up a procedure to “answer” heartbeat
    - e.g. thread waiting for messages and sending answers, RMI/RPC server with predefined heartbeat methods, ...
- individuates a general problem in connection
  - remote resource problem or network problem

# TCP/IP connection

- try {
  Socket s = new Socket(...);
  int ch = s.getInputStream().read();
  ...
  } catch (Exception e) {
  ... // handle problem with socket
  }
- java.net.SocketException: Connection reset
- only with non buffered streams !!!
- individuates a general problem in connection
  - remote resource problem or network problem

# Template: Fault tolerance (sample)

- map template
  - worker failure
  - probes
    - heart beat
    - connection closed on the partition distribution channel
- countermeasure
  - recruit new worker PE + deploy code + distribute partition
  - possible if and only if
    - “lost” partition was saved on the emitter

# Template: Fault tolerance (sample 2)

- Otherwise (classical approach)
  - checkpoint template at regular intervals
    - pb: what are checkpoint safe points in the code?
    - pb: fault safe checkpoint repository?
  - probe fault
    - stop computation
    - recruit new resources
    - restart from last checkpoint

# Template: Security

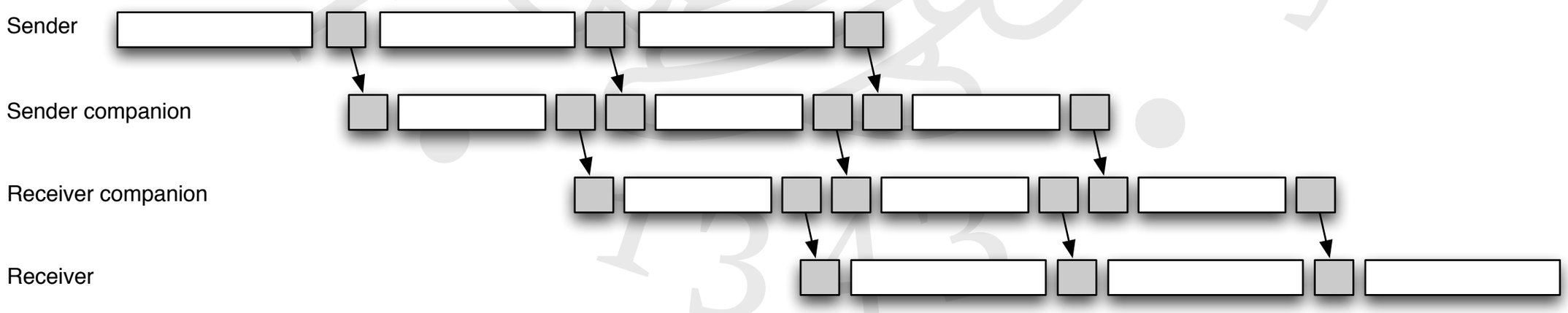
- Code securing & data securing
  - communications
    - use proper protocols
  - storage
    - use proper data format
- whenever subject to uncontrolled access
  - when using public networks
  - when leaving data on remote resource disks/storage

# Typical secure communication (SSL)

- SSL protocol
  - public key infrastructure used to negotiate a private session key
  - private session key used to actually transfer data & code
- cost:
  - negotiation phase: paid once and for all
  - cyphering with private session key (paid at each send)
  - de-cyphering with private session key (paid at each receive)

# Pipelined communication securing

- sending process has a companion cyphering process on a node connected through a secure network link
- receiving process has a companion de-cyphering process on a node connected through a secure network link
- securing and un-securing happens in pipeline
- works with one way communications
  - not working with RMI/RPC



# Pipelined communication securing (2)

- with modern multi core processors
  - multi threaded process template
    - 1 thread prepares the message (receives and decyphers)
    - 1 thread cyphers the message and sends it remotely (consumes the message)

# Green computing

- Forthcoming multi-core chips
  - allow core-by-core switch-off (clock setup)
- Already possible
  - switch-off blades / PEs in a cluster
- In a template:
  - figure out the number of machines needed
  - according to the performance mode of the template
  - *and* the current measures on the application behaviour
  - then dimension resources accordingly (+ and -, in case)

# Support mechanisms

- in many core architectures
  - clock per core
    - standby → low consumption → high performance
- also
  - core by core
    - shutdown with insulation in case of fault
    - currently being investigated

# Proactive Peak Power Management for Many-core Architectures

John Sartori and Rakesh Kumar
Coordinated Science Laboratory
1308 West Main St
Urbana, IL 61801

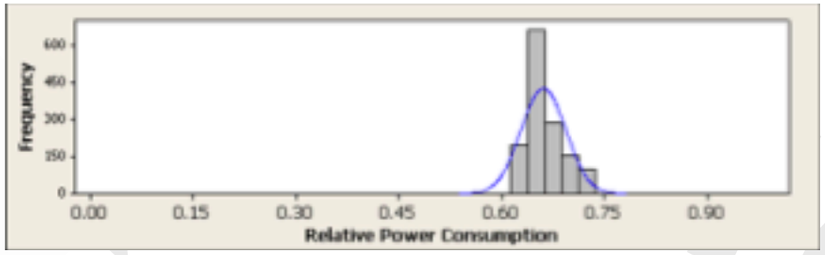
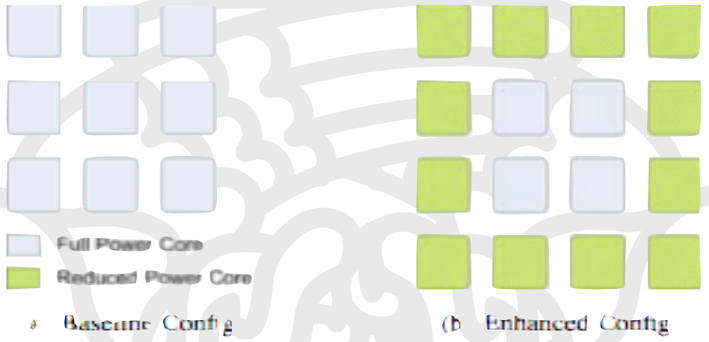



Figure 1. Relative Power Consumption by a Multi-core Processor




(a) Baseline Config (b) Enhanced Config

Figure 2. Power Equivalent Processor Configurations.

theory we should be able to add approximately 50% more cores *running at full power* and still remain below the peak power on average.

We propose an architecture with *proactive peak power management* that has several more cores on the die than the baseline processor. The average power of the new architecture is just below the peak power of the baseline processor while a proactive peak power management mechanism *guarantees* that the peak power of the baseline will never be exceeded, even though the processor contains several more cores than the power budget would normally allow. The proactive mechanism provides this guarantee by intelligently scaling down the power for a subset of cores. Power can be scaled down through the application of V/f scaling, clock gating, or power gating. The throughput of this architecture is higher than the baseline processor, due to the increased number of cores.

For example, figure 2 shows a 9 core baseline processor with all cores running at full power and a 16 core processor with a proactive peak power management mechanism that runs 4 cores at full power and 12 cores at reduced power through the use of V/f scaling. The aggregate peak power requirement of both processors is the same. Nevertheless, the throughput of the 16 core processor can be up to 78% higher, assuming linear dependence with the number of cores on the die. The observed performance gains, of course, will be somewhat less than this optimal value due to overhead costs of our power management techniques, and since throughput actually does decrease with V/f scaling in most cases.

# Sample (experimental) techniques

## Exploring Power Management in Multi-Core Systems

Reinaldo Bergamaschi[1], Guoling Han[2], Alper Buyuktosunoglu[1], Hiren Patel[3], Indira Nair[1],
Gero Dittmann[1], Geert Janssen[1], Nagu Dhanwada[4], Zhigang Hu[1], Pradip Bose[1], John Darringer[1]
[1] IBM T. J. Watson Research Center, Yorktown Heights, NY 10598; [2] University of California, Los Angeles, CA 90095;
[3] Virginia Tech, Blacksburg, VA 24060; [4] IBM STG, East Fishkill, NY 12533
berga@us.ibm.com

- @inproceedings{1356973,
  author = {Bergamaschi,, Reinaldo and Han,, Guoling and Buyuktosunoglu,, Alper and Patel,, Hiren and Nair,, Indira and Dittmann,, Gero and Janssen,, Geert and Dhanwada,, Nagu and Hu,, Zhigang and Bose,, Pradip and Darringer,, John},
  title = {Exploring power management in multi-core systems},
  booktitle = {ASP-DAC '08: Proceedings of the 2008 conference on Asia and South Pacific design automation},
  year = {2008},
  isbn = {978-1-4244-1922-7},
  pages = {708--713},
  location = {Seoul, Korea},
  publisher = {IEEE Computer Society Press},
  address = {Los Alamitos, CA, USA}
  }

# Processor model

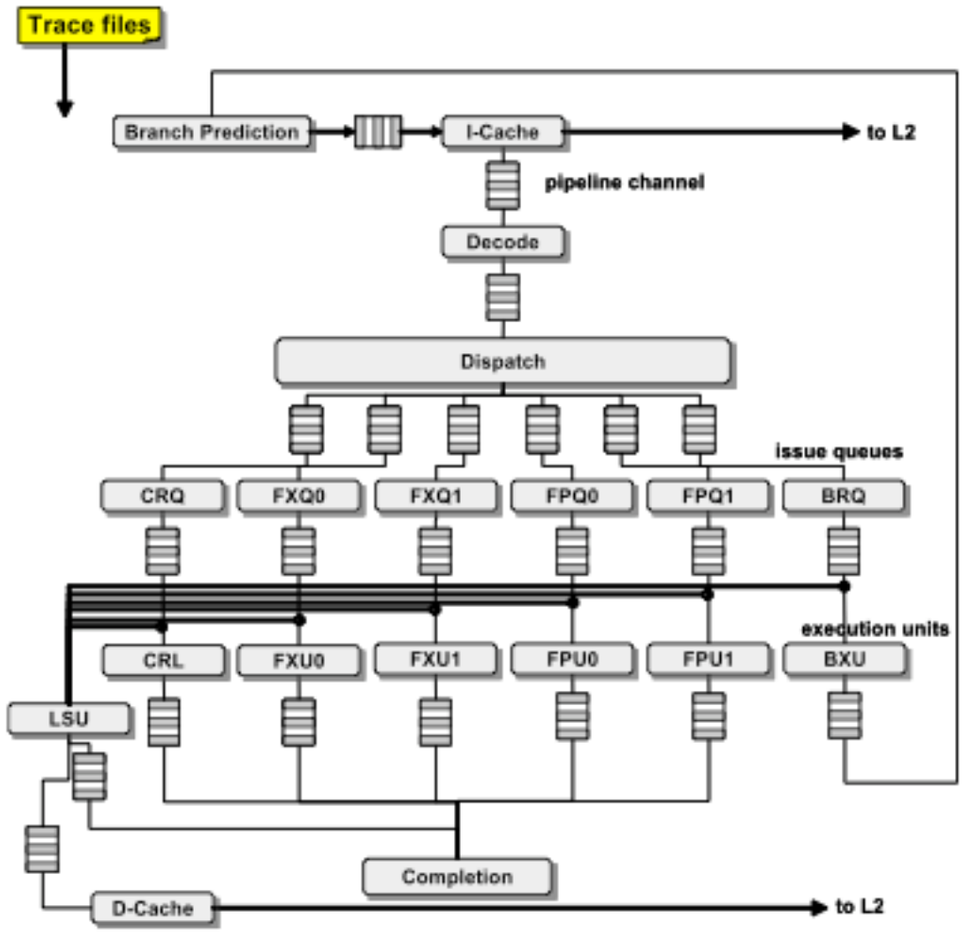
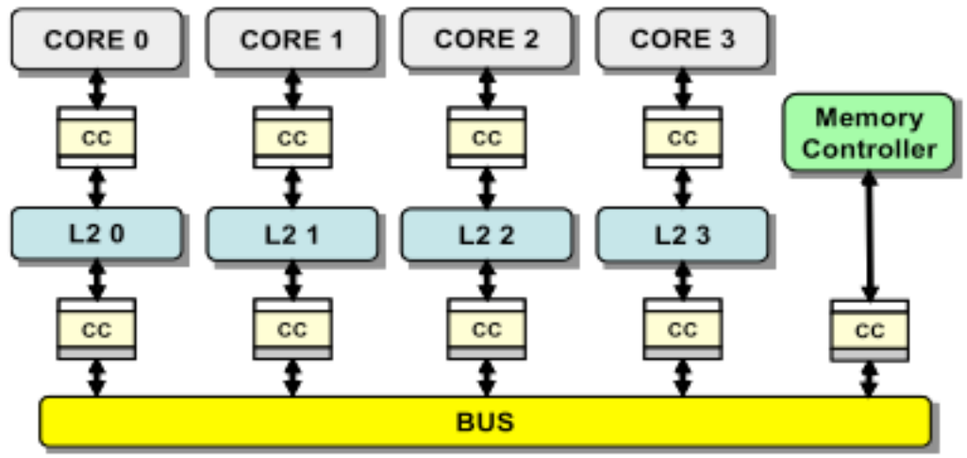
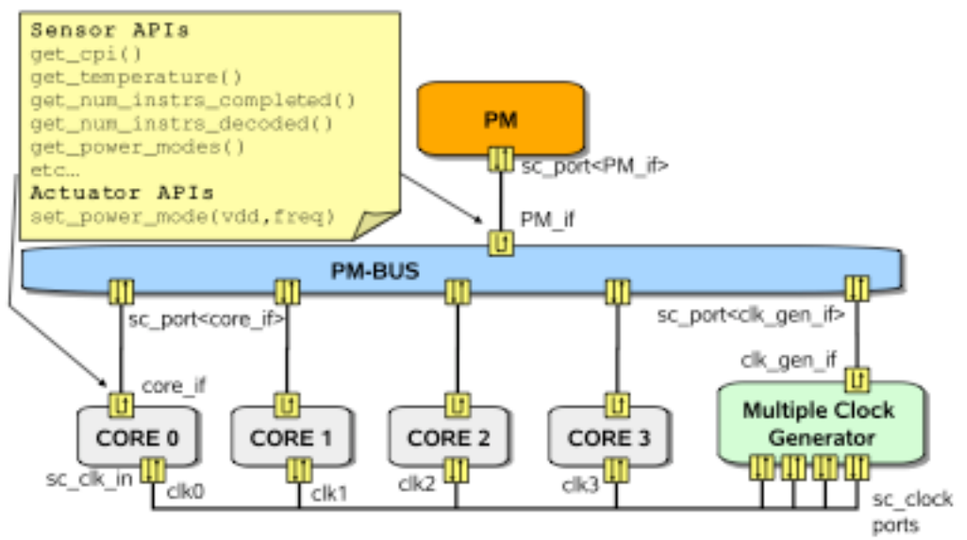



Fig. 1.: Internal organization of core model.



# The algorithm

dynamic voltage and frequency scaling (DVFS)

*MaxBIPS Algorithm*

C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: maximizing performance for a given power budget". In Proceedings of the 39th Annual International Symposium on Microarchitecure (MICRO'06), IEEE, pages 347-358, December 2006

The MaxBIPS algorithm relies on the fact that when a given core switches from power mode A (VddA, freqA) in observation window N to power mode B (VddB, freqB) in observation window N+1, the future performance and power is predictable using simple formulas, as shown below. This assumes that the workload characteristics do not change significantly from one window to the next.

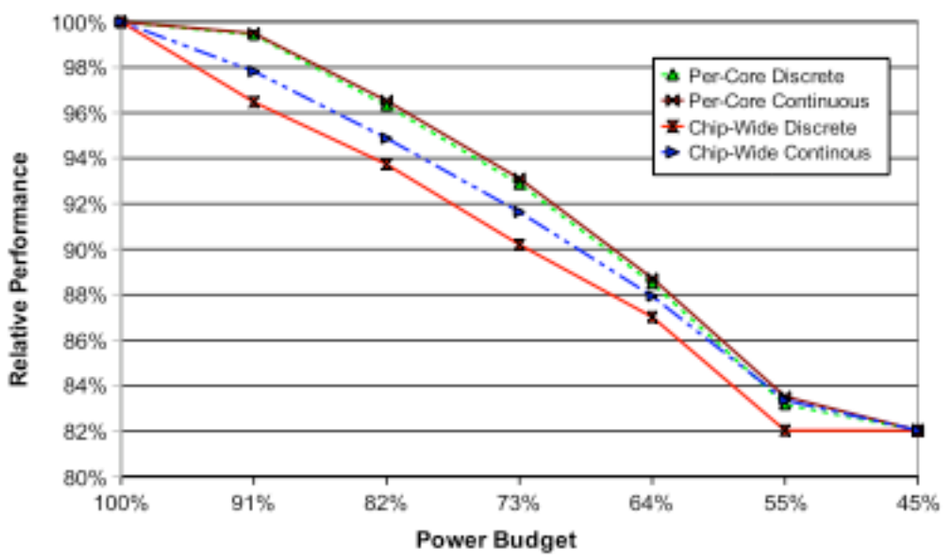
| Observation Window | $N$ | $N+1$ |
|---|---|---|
| Mode | $(v, f)$ | $(v', f')$ |
| Performance | $I$ | $I * (f'/f)$ |
| Dynamic Power | $P$ | $P * (v'/v)^2 * (f'/f)$ |
| Static Power | $L$ | $L * (v'/v)^3$ (approx.) |

Based on these formulas, the MaxBIPS algorithm computes the estimated power and performance for all possible tuples $[\text{core}_i, \text{mode}_j]$ and selects the mode that maximizes performance while not exceeding the power budget. The worst-case complexity of the algorithm is $O(M^N)$, with $M$ the number of modes and $N$ the number of cores. In practice many $[core, mode]$ tuples can be pruned out during the search as soon as the total power exceeds the current budget, and both $N$ and $M$ are limited for practical reasons; as a result the algorithm can afford a simple search on all tuples for the optimal solution.

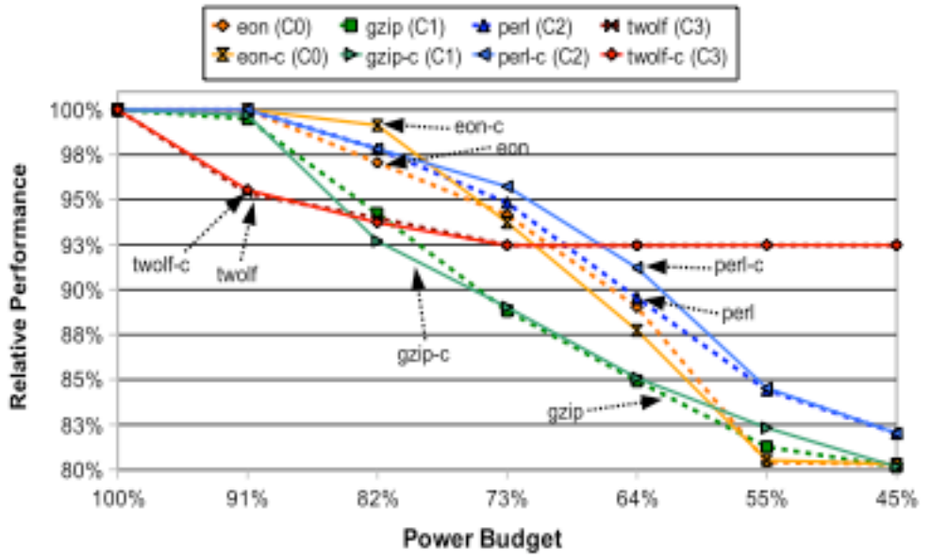
discrete power mode algorithm

compared with continuos model

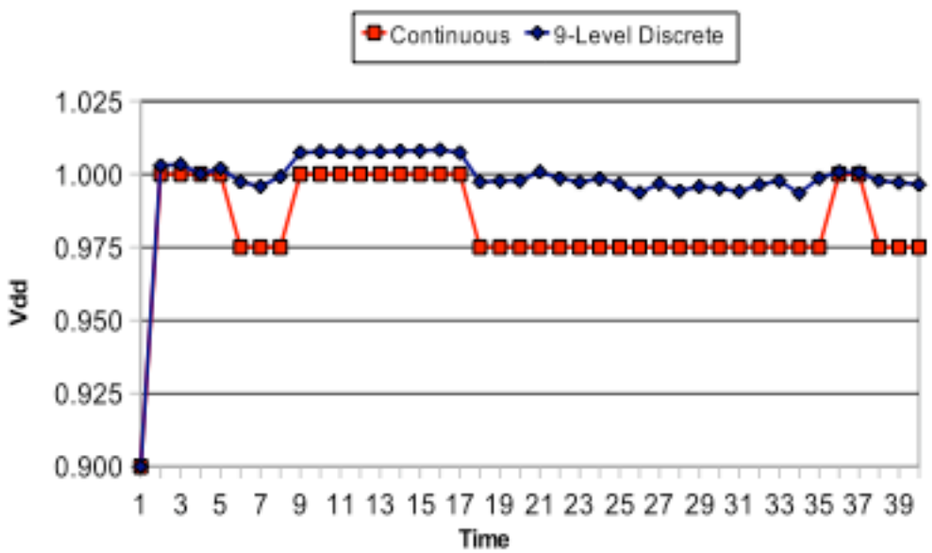
# Experimental (simulation) results



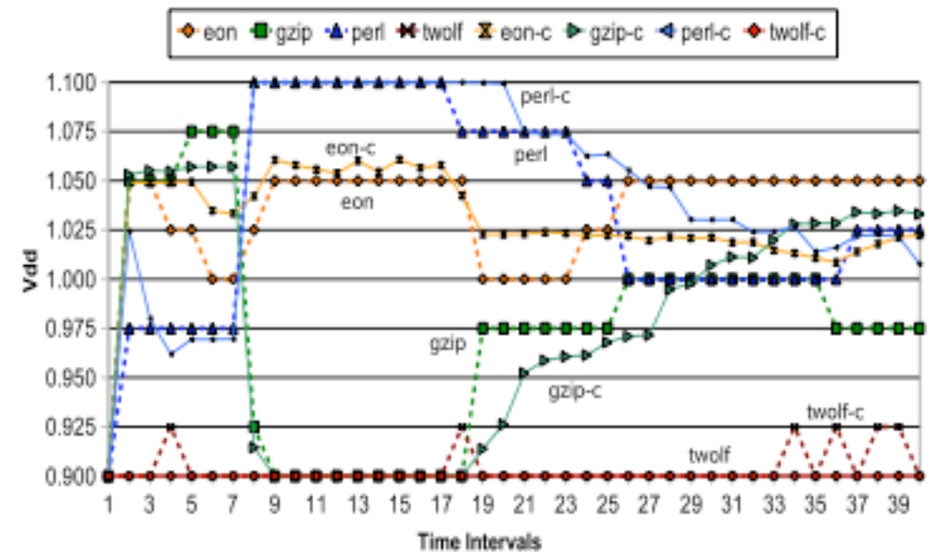

(a) Overall chip performance for decreasing power budgets.




(b) Individual core performance under per-core DVFS.




(c) Chip-wide DVFS transitions for 73% power budget.




(d) Per-core DVFS transitions for 73% power budget.

# In perspective ...

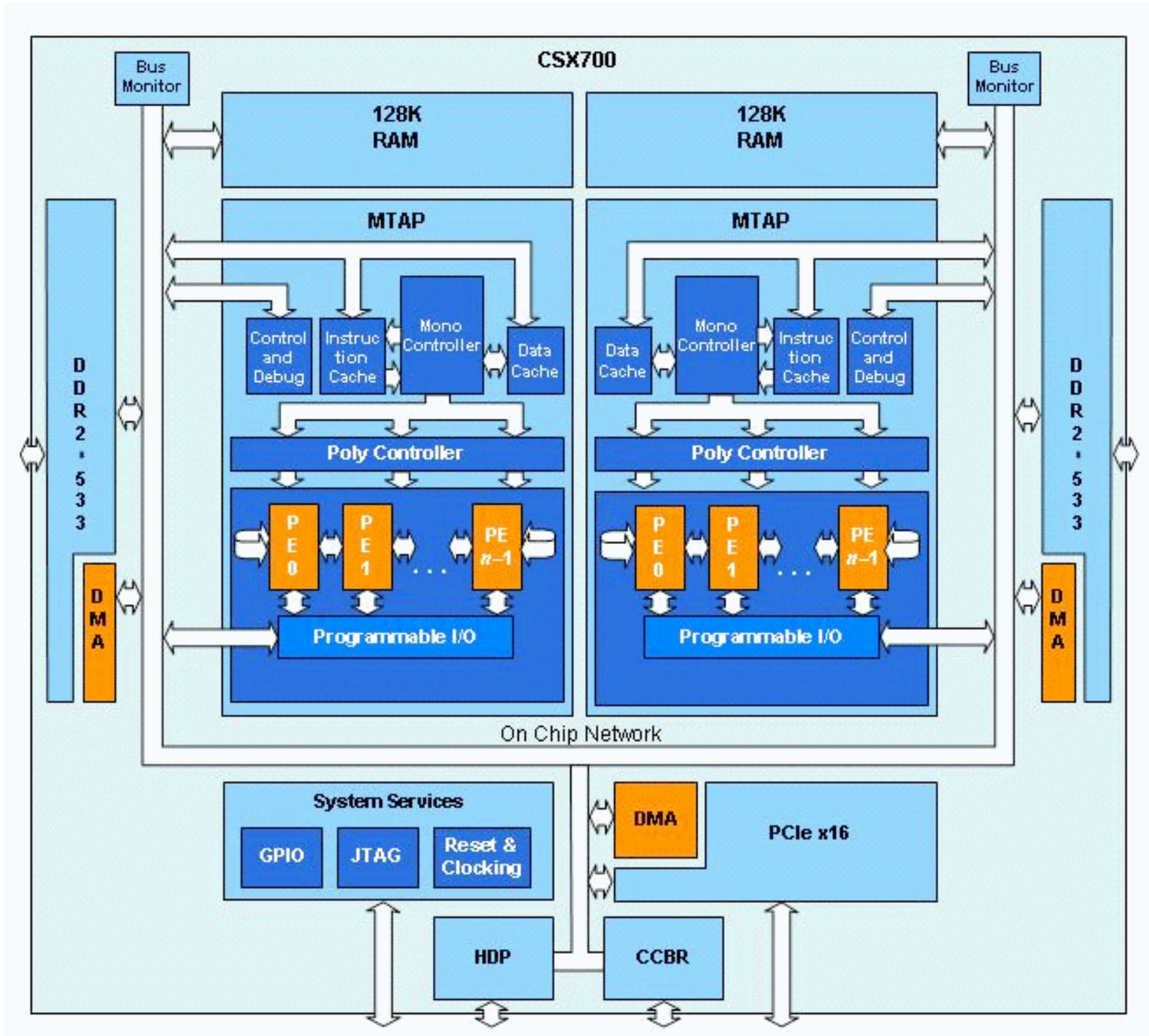
- CSX 700 (by Clearspeed technologies)
- 192 cores
- 96 GFLOPS (IEEE single/ double precision)
- 10 WATTS

**Adaptive power management features**

The CSX700 includes numerous adaptive power management features. The clock speed can be changed dynamically under software control in real-time while an application is in mid-flight, allowing the application to change the performance and power consumption of the device. The CSX700 includes integrated sensors for chip temperature and power consumption, allowing a CSX700-based system to dynamically manage a running application to stay within set power or temperature envelopes.


**Figure 1: The CSX700 192-core embedded processor**

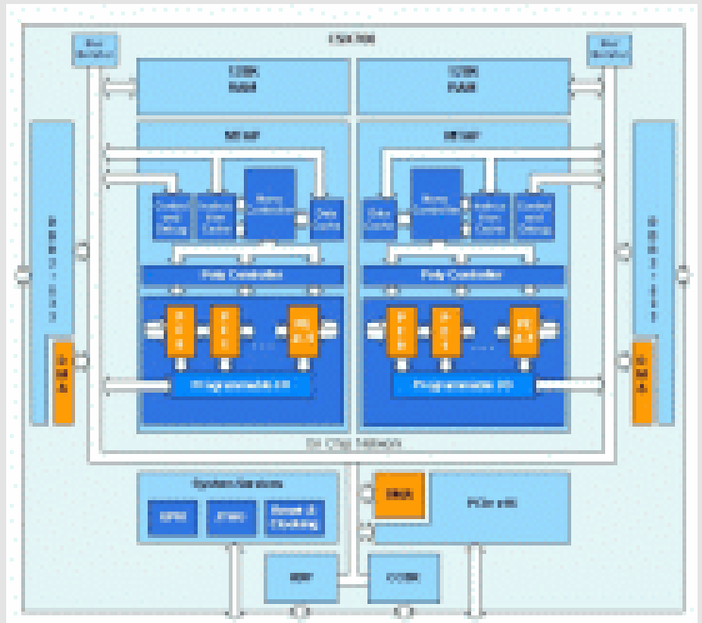
# Architecture




The CSX700 SoC design

CLOSE

# Architecture



The CSX700 SoC design
Click image to enlarge

**Performance**

- 250MHz core clock frequency
- 96 GFLOPS single or double precision
- 75 GFLOPS sustained double precision DGEMM
- 9W typical power dissipation
- 192 Gbytes/s internal memory bandwidth
- 2 x 4 Gbytes/s external memory bandwidth
- 4 Gbytes/s chip-to-chip bandwidth

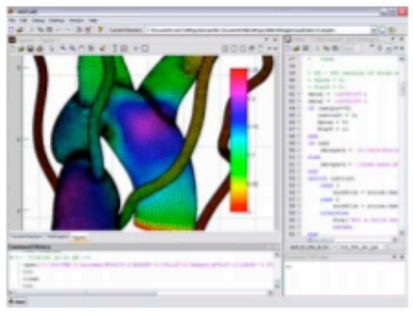
**Features**

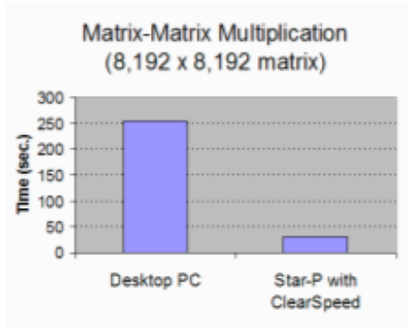
192 high-performance processing elements, each with:

- Dual 32 & 64-bit FPU
- 6 Kbytes high bandwidth memory
- PCIe x16 host interface
- 2 x 64-bit DDR2 DRAM interface with ECC support
- 256 Kbytes on-chip scratchpad memory
- On-chip instruction and data caches
- ECC protection on all on and off-chip memory
- ClearConnect NoC provides on-chip and inter-chip data network
- Host debug port
- 64-bit virtual, 48-bit physical addressing
- On-chip DMA controller

# Usage ...

- Accelerate MATLAB® code
- 10-100X faster computation
- 10-100X larger data sets
- No re-programming in C/C++/Fortran/MPI






Data Parallel Scaling:

## Star-P® and ClearSpeed™ Acceleration for Multi-Core Clusters

Star-P is a new programming paradigm that enables desktop tools – such as the Python, MATLAB® from The MathWorks, R and others – to be run on accelerated multi-core systems for the development, testing and deployment of parallel algorithms. Servers and/or clusters that are equipped with ClearSpeed Advance™ accelerators can now be programmed using the Star-P parallel programming platform. Applications with underlying math functions such as large matrix multiplies benefit from ClearSpeed's floating point acceleration, and with Star-P, additional functionalities and applications can run in parallel on scalar systems.

### Target Markets

Financial Services, Universities and National Laboratories, Life Sciences, Healthcare, and Manufacturing enterprises have 100's and even 1000's of engineers, scientists and analysts that use desktop tools, such as MATLAB to develop mathematical and statistical models for image processing, signal processing, financial quantitative analysis, and other modeling applications. IT and HPC executives buy large scale systems and accelerators based on applications that need the capacity and performance of these systems.

### The Star-P Solution: Adding Software Capabilities for ClearSpeed Advance Accelerators

ClearSpeed acceleration can be realized through plug and play with standard math libraries and tools such as MATLAB, or by programming with ClearSpeed's software development tools. Star-P enhances this dynamic by offering a parallel programming platform, combining accelerator technology, that the large majority of engineers, scientists and analysts within high performance computing (HPC) can use to build custom applications without having to know all of the technical details of low level programming techniques required in C, C++, Fortran, Message Passing interface(MPI) and others.

# Usage (2)

## ClearSpeed Demonstrates Massively-Dense Computing with TeraFLOP Performance and Enhanced Software Tools for Easier Programming of Accelerated Multi-Core Systems.



Demonstrating the potential of heterogeneous multi-core systems, a ClearSpeed accelerated dual core Intel[R]-based cluster recently delivered over one TeraFLOP
of LINPACK performance while occupying just 16u of rack space. With each ClearSpeed Advance e620 accelerator delivering 80 GigaFLOPS
peak double precision performance and over one GigaFLOP
per watt of sustained LINPACK performance, the entire cluster had a maximum power consumption of less than 7KW and completed the benchmark in just 14 minutes, half the time required by the non-accelerated system. The energy used to achieve this TeraFLOP performance was approximately 1.5KWh, costing a mere 15 cents assuming a cost of 10 cents per KWh. With the latest quad core
Intel processors, the same performance and energy profile could be compressed into just 10 rack units and cost less than $150,000. To underscore what a difference a decade makes, in 1997 Intel impressed the industry with the world's first-ever terascale system known as ASCI Red
that was delivered to Sandia National Laboratories
. With 4,510 compute nodes the then state-of-the-art and record-breaking system occupied 2,500 square feet, consumed 850KW and cost $55 million.

"The massively-dense, energy-efficient systems that can be built with ClearSpeed accelerators are essential to providing the orders of magnitude greater compute power required to tackle the next level of grand challenge problems, such as modeling protein interactions or simulating turbulent airflow for aero engine design," said Ben Bennett, ClearSpeed general manager. "Even more important than the accelerators themselves, making the software development environment easy and familiar is the single most critical element in realizing the performance of accelerated multi-core systems. ClearSpeed has established a significant lead in this area."