“Industrial” Auto-tuning using CrayATF

Adrian Tate
Technical Lead of Scientific Libraries
Senior Software Engineer, Cray Inc.
iWAPT, Tokyo, Oct 2009

History of Cray Supercomputers

1976

Cray-1

1982

Cray-XMP

1985

Cray-2

1988

Cray-YMP

1991

Cray-C90

1993

Cray-T3D

1994

Cray-T90

1995

Cray-T3E

2001

Cray-SV1

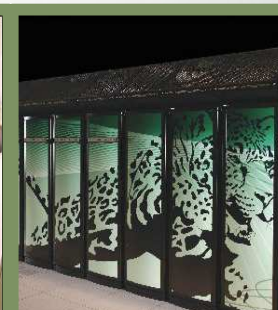
2003

Cray-X1

2005

Cray-XT3

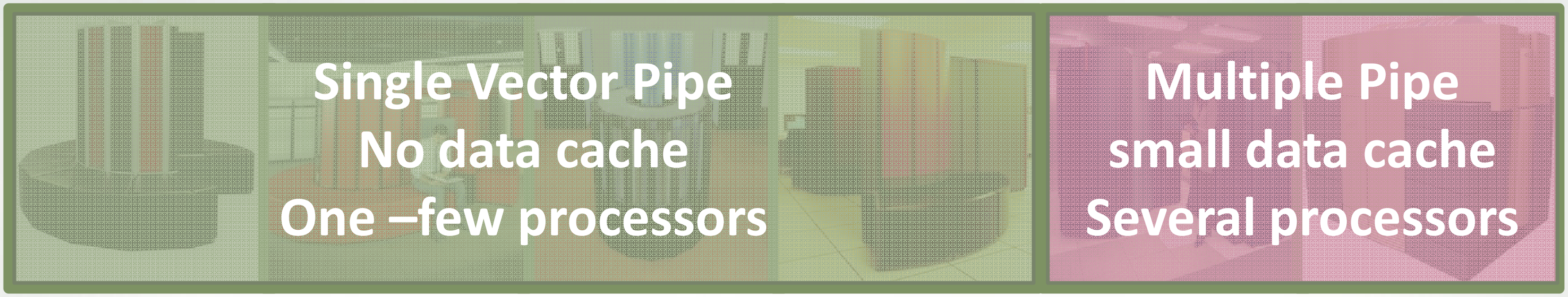
2008



Cray-XT5

hardware trends

1976 | 1982 | 1985 | 1988 | 1991 | 1993



Cray-1 | Cray-XMP | Cray-2 | Cray-YMP | Cray-C90 | Cray-T90

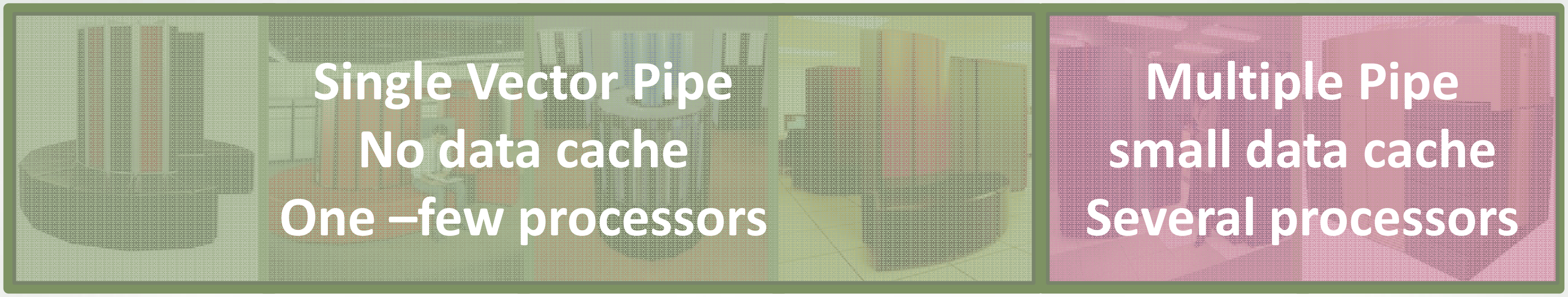
1994 | 1995 | 2001 | 2003 | 2005 | 2008

Cray-T3D | Cray-T3E | Cray-SV1 | Cray-X1 | Cray-XT3 | Cray-XT5

hardware trends

BLAS1 LINPACK BLAS2 BLAS3 LAPACK

1976 1982 1985 1988 1991 1993



Cray-1 Cray-XMP Cray-2 Cray-YMP Cray-C90 Cray-T90

ScaLAPACK PETSc ATLAS FFTW Trilinos

1994 1995 2001 2003 2005 2008

Cray-T3D Cray-T3E Cray-SV1 Cray-X1 Cray-XT3 Cray-XT5

What is the role of vendor in libraries tuning today?

- Clearly, not to make the problem worse
- Improve performance of PETSc and Trilinos on Cray MPPs
  - tuning sparse matrix vector multiply in general fashion
- Tune HPL benchmark for largest machines (massive runtime)
  - O (N^3) factorization driven by multiple parameters
- Tune Dense linear algebra (BLAS3 mainly)
  - BLAS3
- Apply the above only to the Cray hardware
  - Allows the search space to be manipulated to our advantage
- Tune eigensolvers in a general purpose way

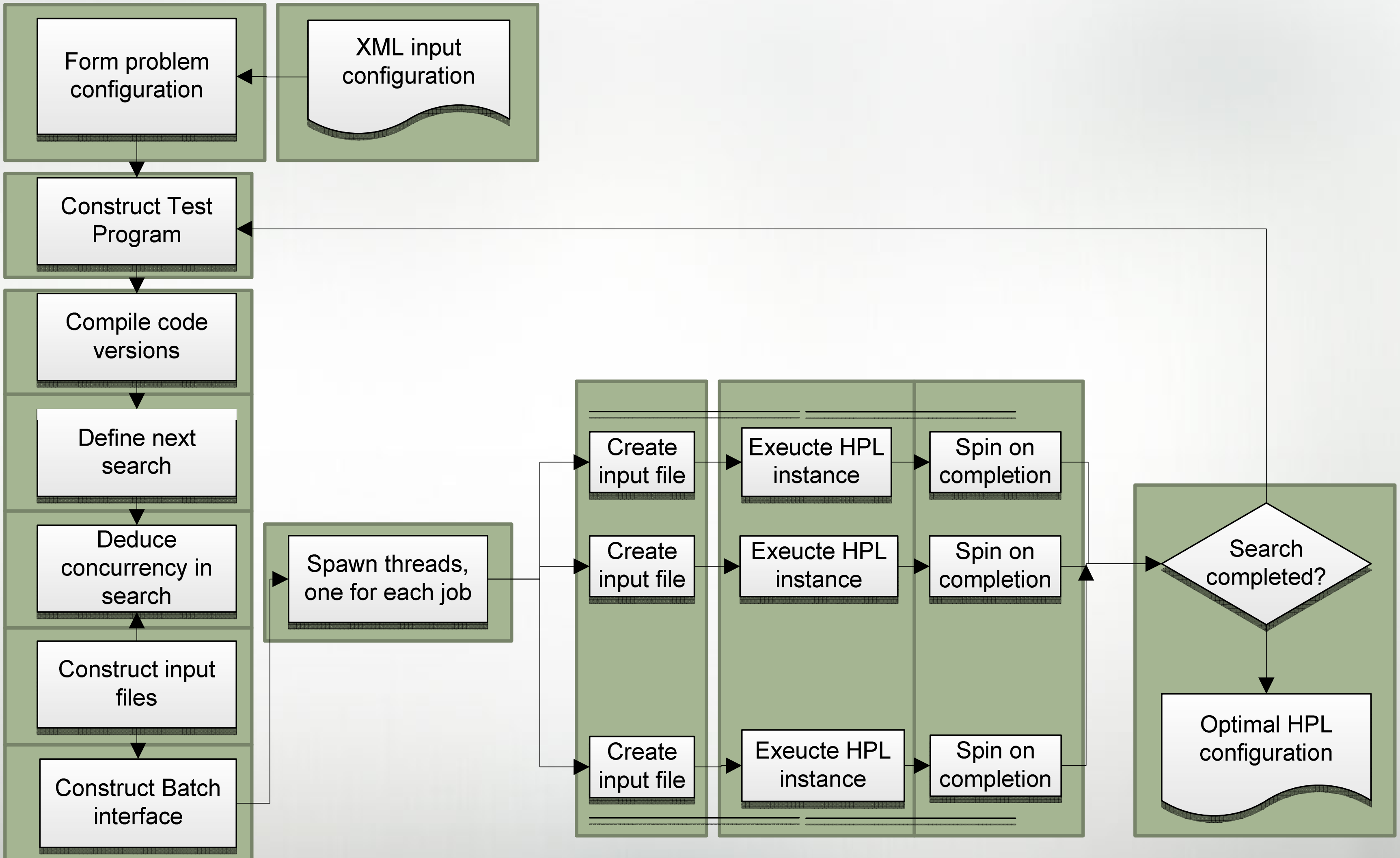
- It is pretty obvious that hand-tuning alone cannot achieve this

  Can we construct a generalized AT framework to do all the above?

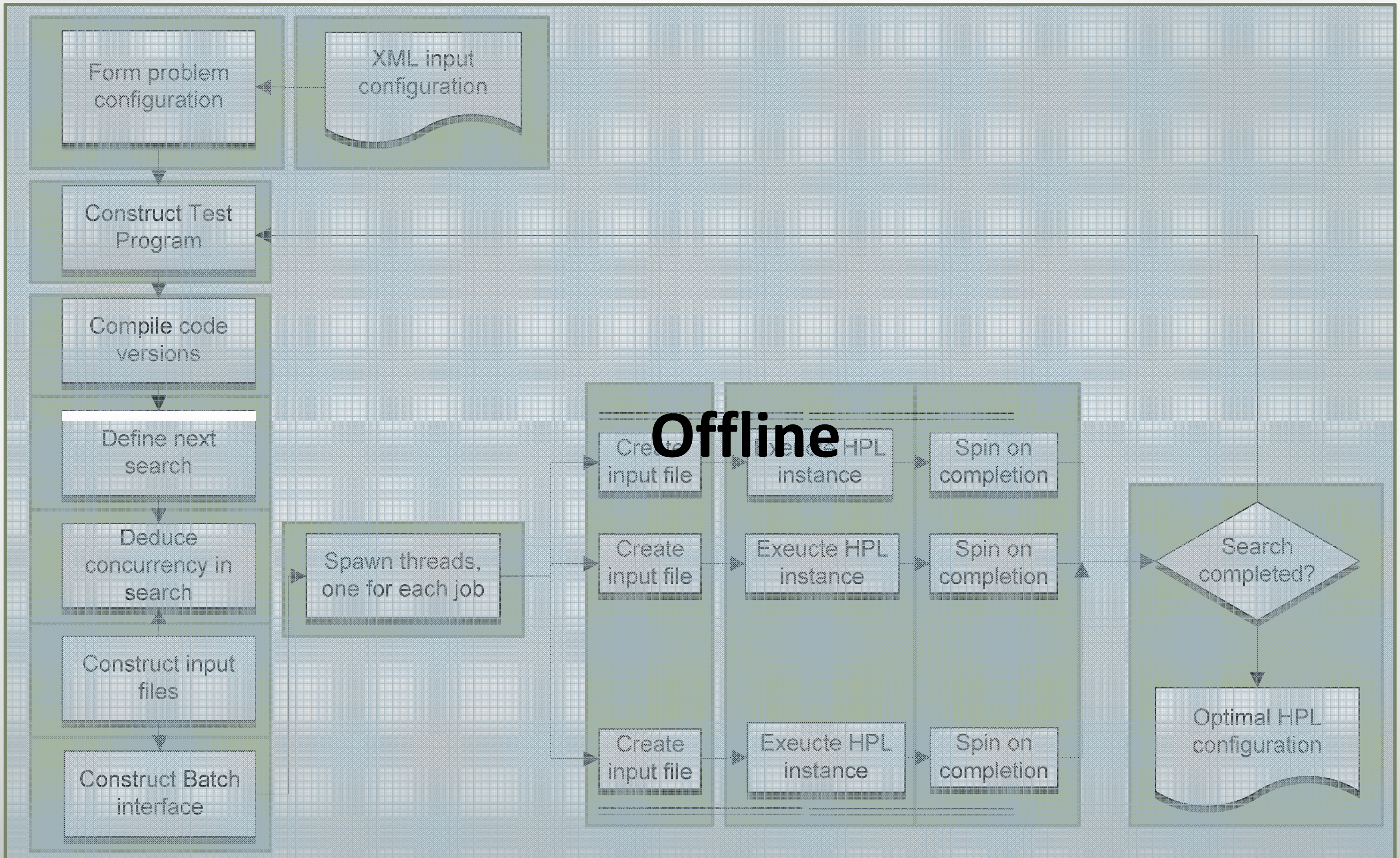
Look at 3 examples : HPL

- HPL (High Performance Linpack)
  - O(N^3) factorization and solve
  - Parameter tuning is now paramount
  - Has 13 parameters (+ 7 more in Cray version)
  - some parameters have very large dimensionality
  - Search space is very large indeed (more later)
  - Has become a massive problem due to excessive runtime

Desired flow for HPL Auto-tuning



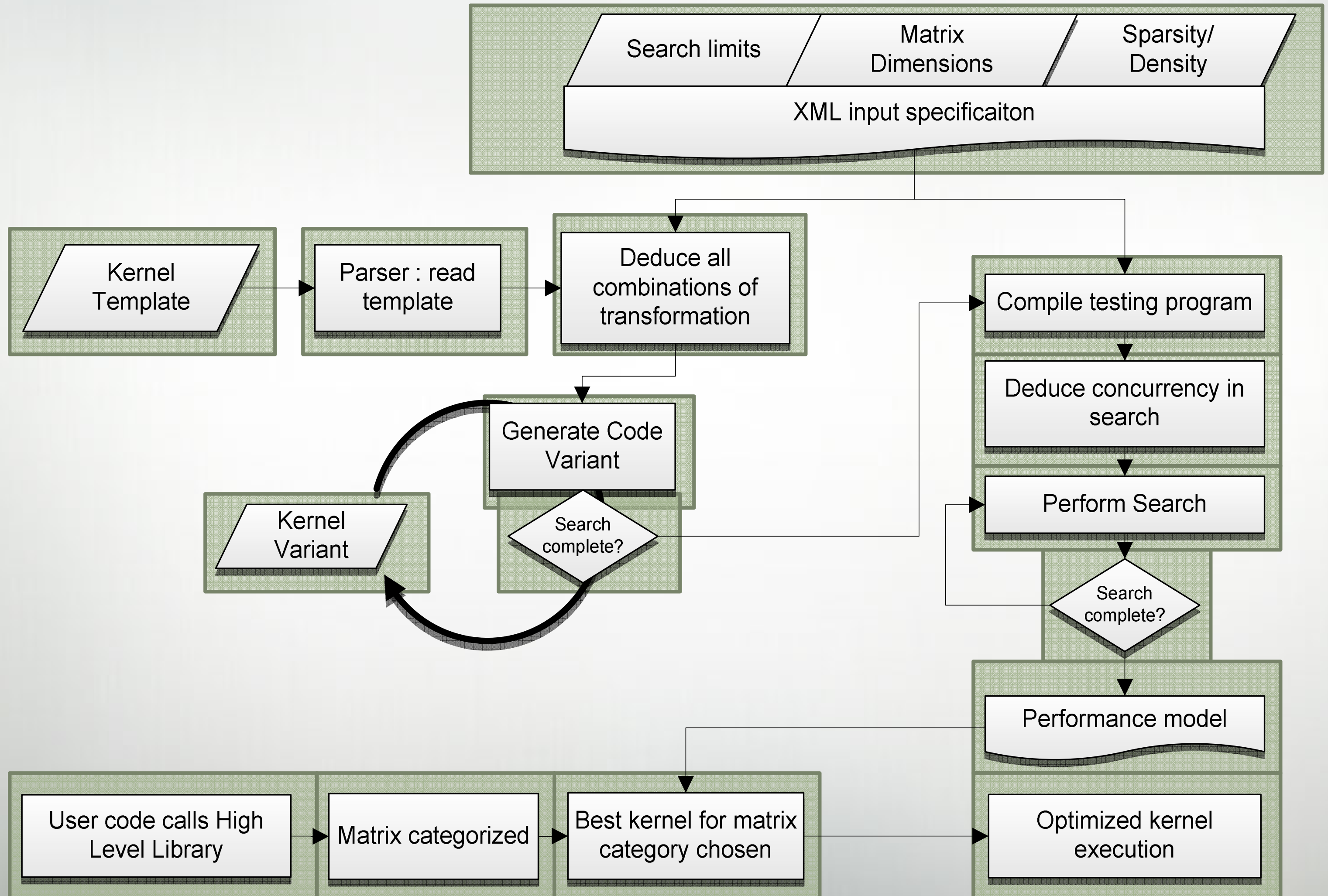
Desired flow for HPL Auto-tuning



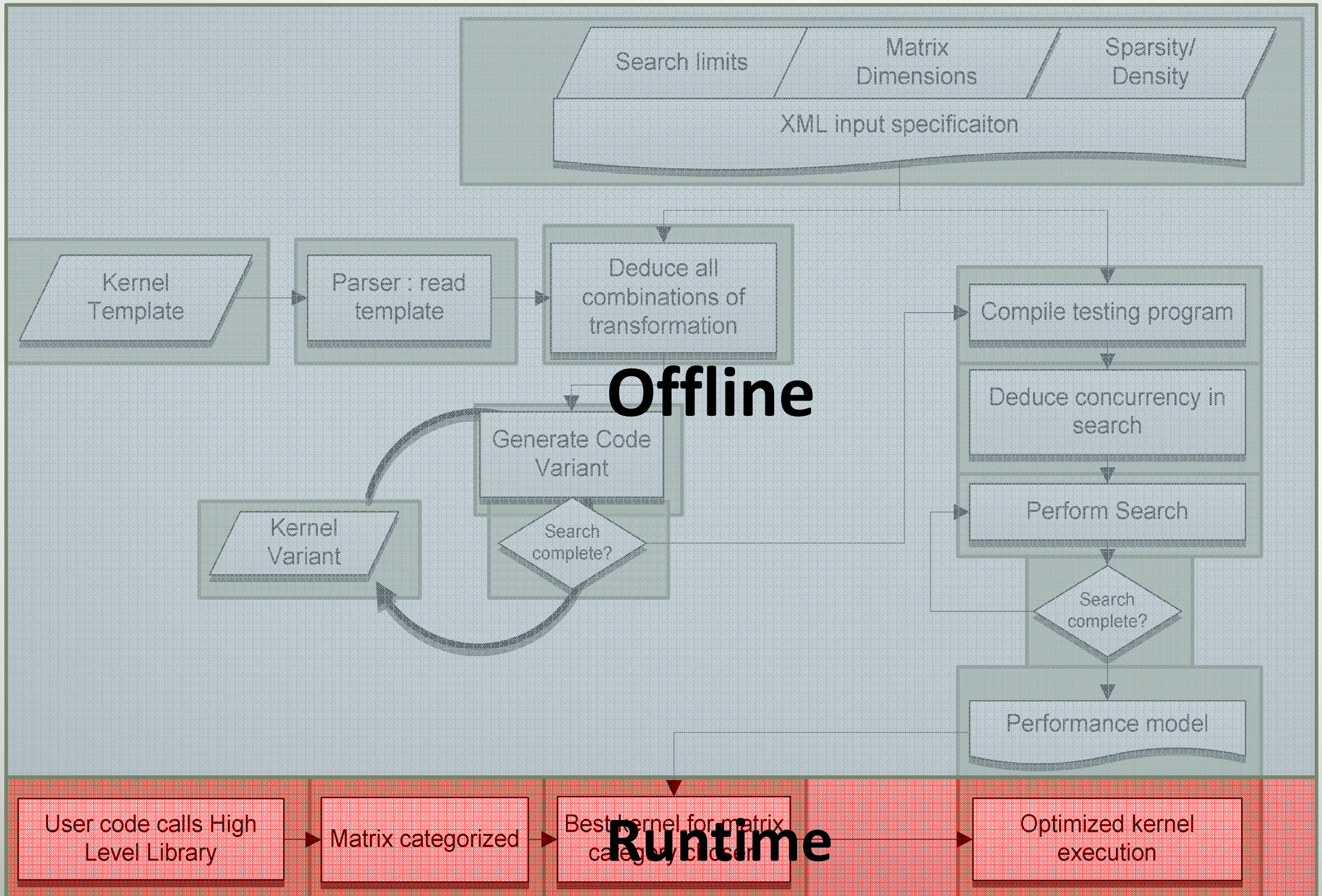
Look at 3 examples : Sparse Linear Algebra

- Sparse Linear Algebra (mainly sparse matrix-vector product)
  - (for CSR) Irregular memory access
  - Memory bandwidth bound kernel
  - Wildly dependent on matrix characteristics
    - Has never had a general purpose tuned code for this reason

Desired Sparse Linear Algebra flow



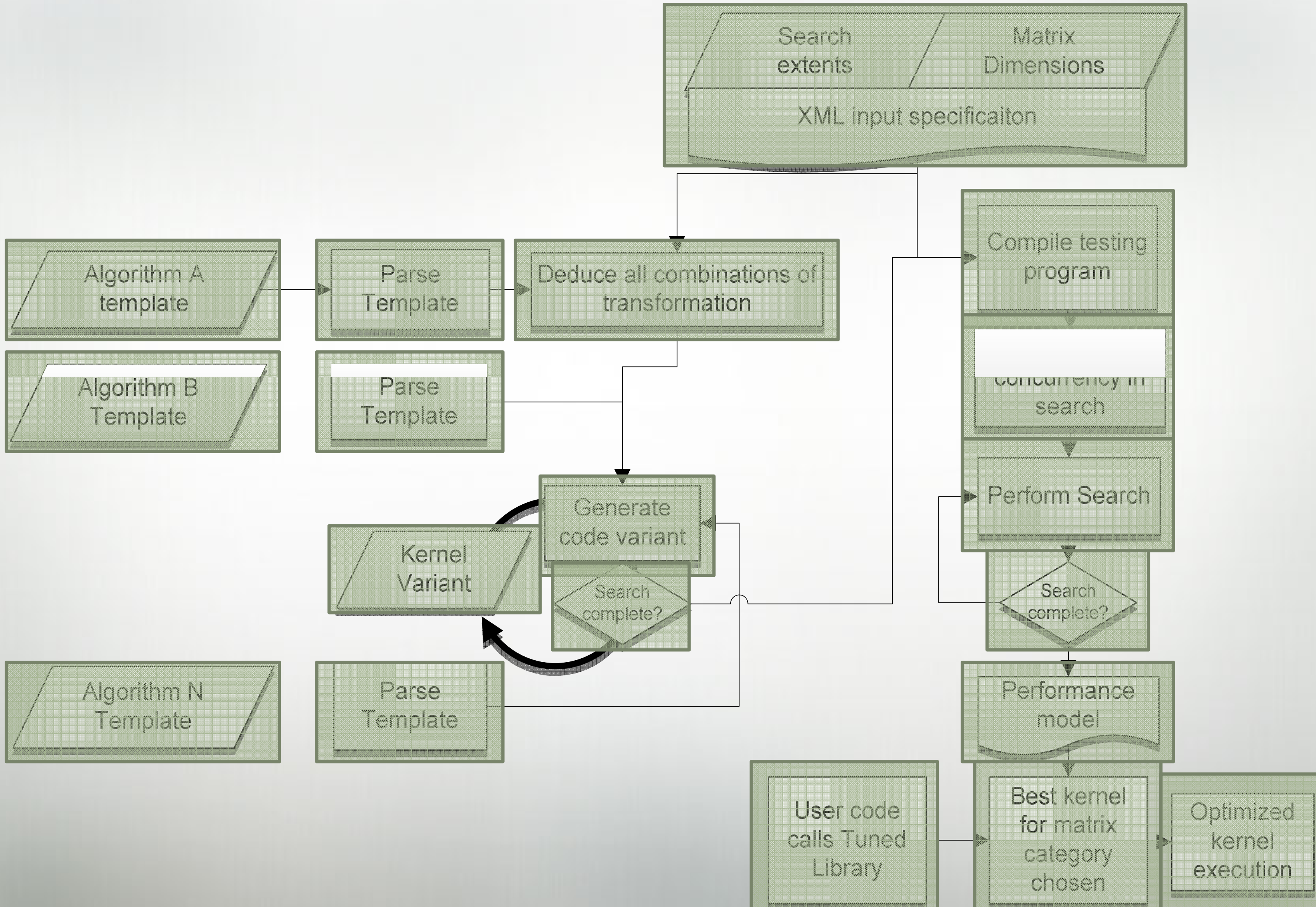
Desired Sparse Linear Algebra flow



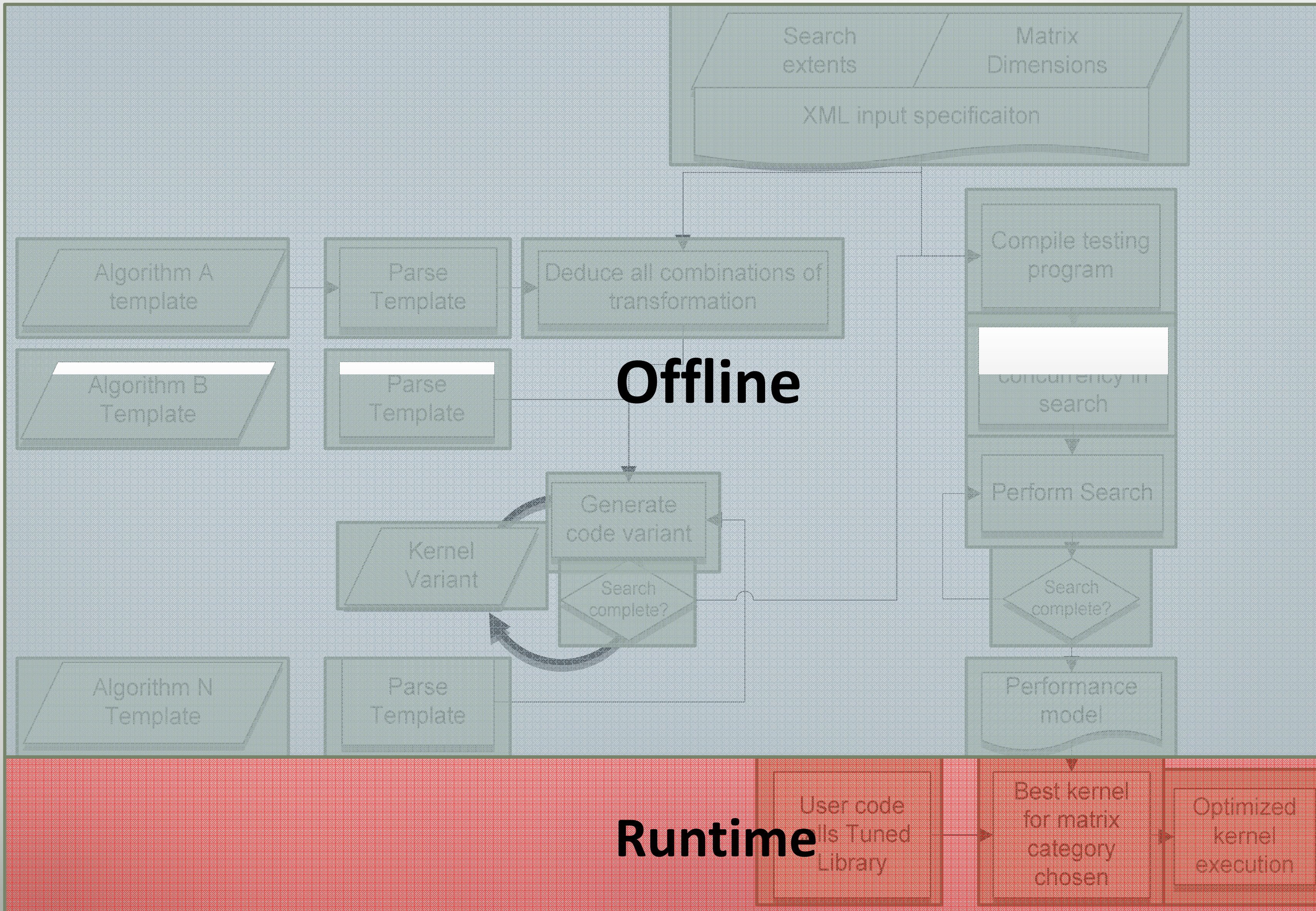
Example 3 : Dense Linear Algebra

- Mostly serial O(N^3) BLAS3 optimizations
- Loop transformations
- Multiple algorithmic effects

Desired Dense Linear Algebra Flow



Desired Dense Linear Algebra Flow



"Industrial" Auto-tuning Model

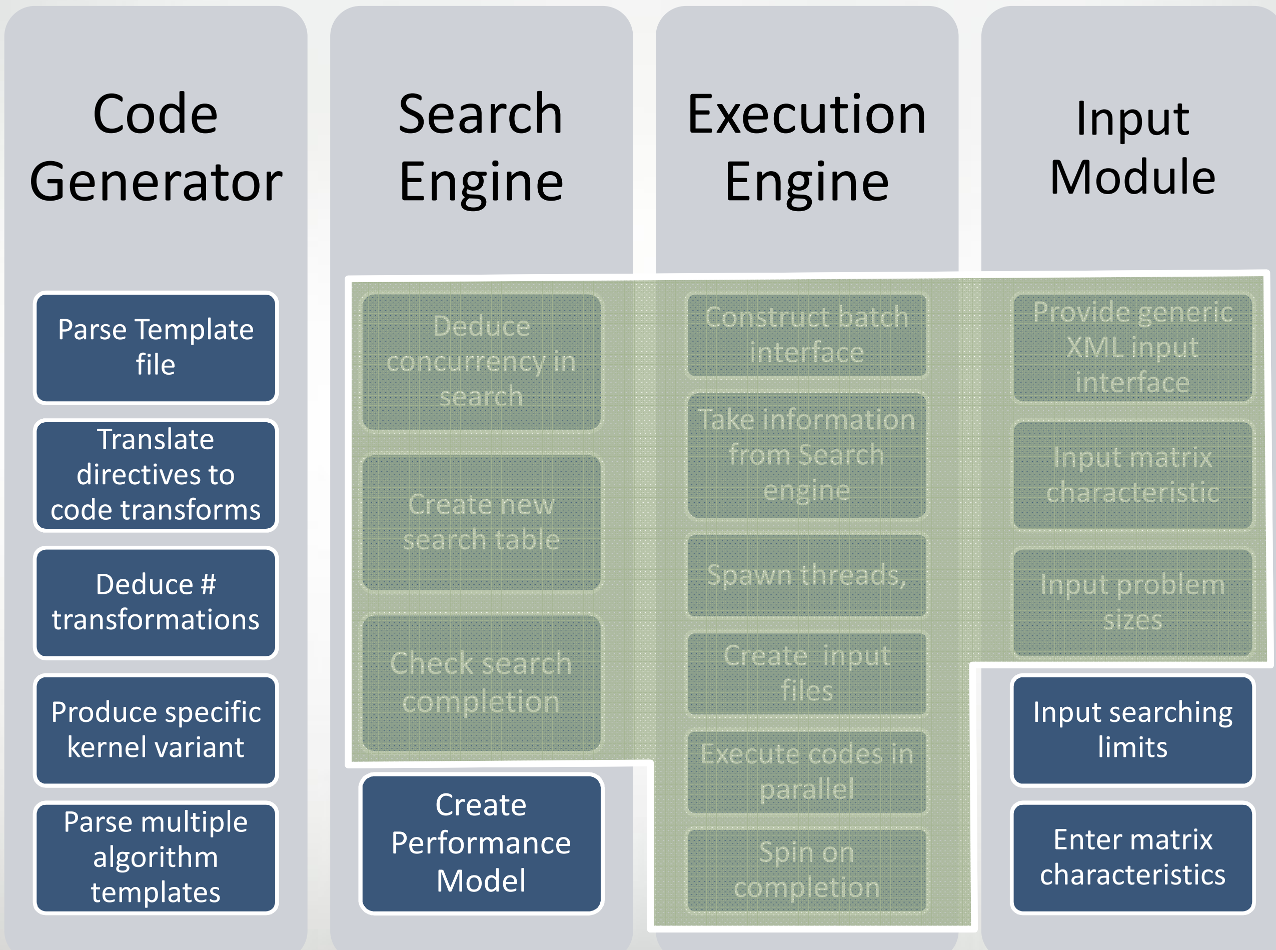
- Search space is made "manageable" because of
  - Restriction to one processor type
  - Knowledge of target problem sizes / characteristics
- Search space is attainable because of infinite resource
- Freedom only to make incremental changes (e.g. no new data-structures)
- Hence, to make an auto-tuner that works in the real world
  - Enormous Offline Testing infrastructure
    - We have unlimited resources available for the offline testing!
  - Performance model as output from offline autotuning
    - We can assume the same architecture for each distribution!
  - Adaptive libraries that take the performance model as input

- The above define our "industrial" autotuning model
- CrayATF is the framework built on this model

Overlay and group the functionality just described

Code Generator	Search Engine	Execution Engine	Input Module
Parse Template file	Deduce concurrency in search	Construct batch interface	Provide generic XML input interface
Translate directives to code transforms	Create new search table	Take information from Search engine	Input matrix characteristic
Deduce # transformations	Check search completion	Spawn threads,	Input problem sizes
Produce specific kernel variant	Create Performance Model	Create input files	Input searching limits
Parse multiple algorithm templates		Execute codes in parallel	Enter matrix characteristics
		Spin on completion	

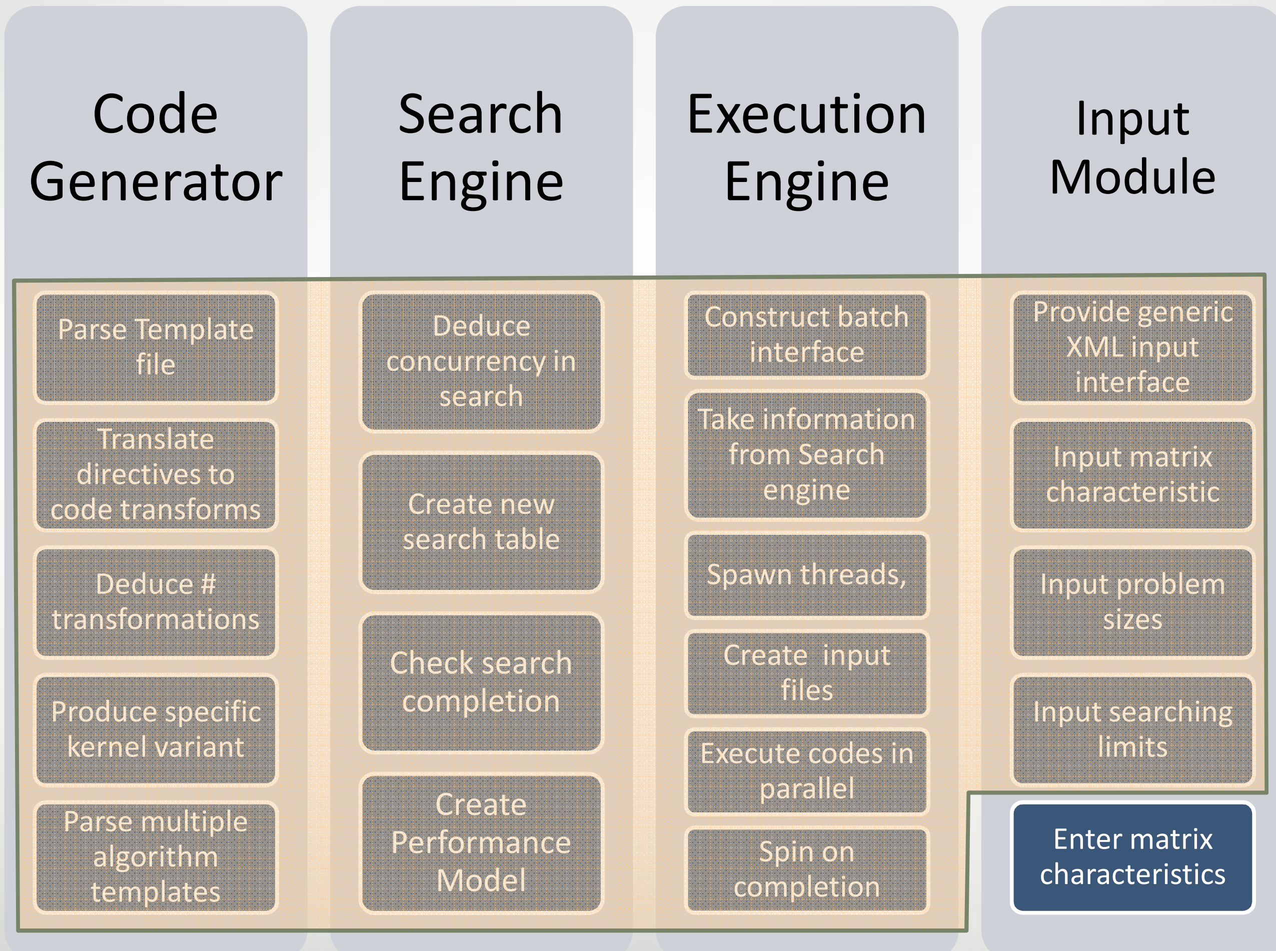
HPL as components



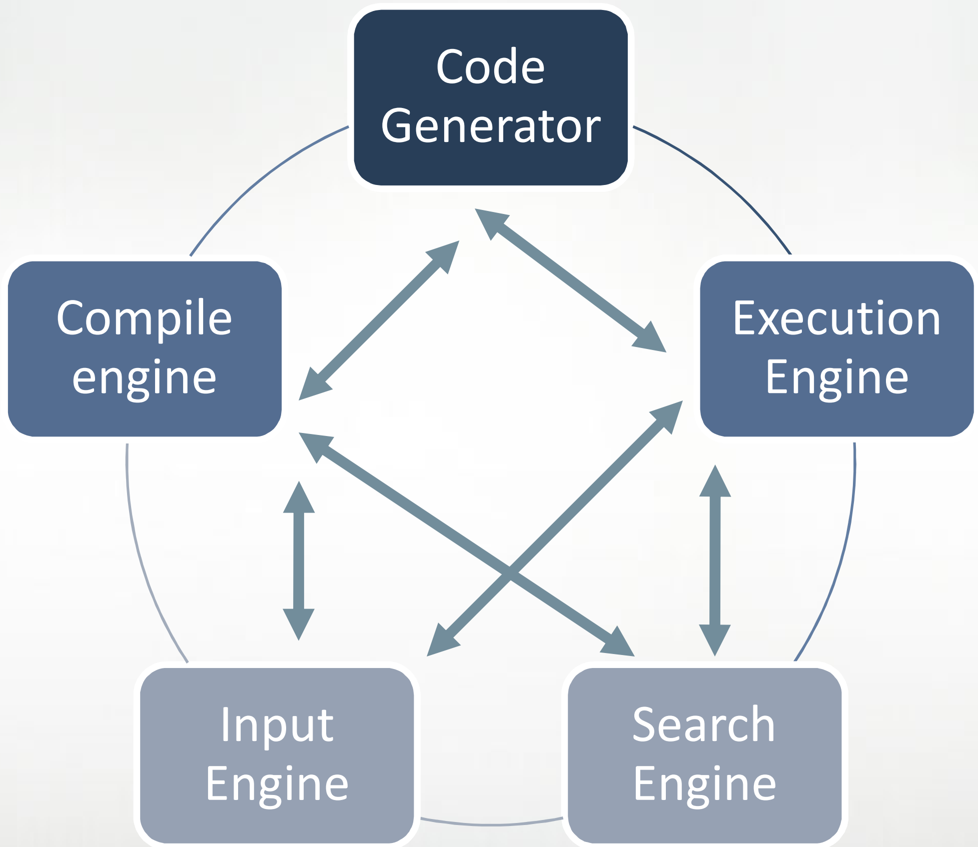
Sparse Linear Algebra as components

Code Generator	Search Engine	Execution Engine	Input Module
Parse Template file	Deduce concurrency in search	Construct batch interface	Provide generic XML input interface
Translate directives to code transforms	Create new search table	Take information from Search engine	Input matrix characteristic
Deduce # transformations	Check search completion	Spawn threads,	Input problem sizes
Produce specific kernel variant	Create Performance Model	Create input files	Input searching limits
Parse multiple algorithm templates		Execute codes in parallel	Enter matrix characteristics
		Spin on completion	

Dense Linear Algebra as components

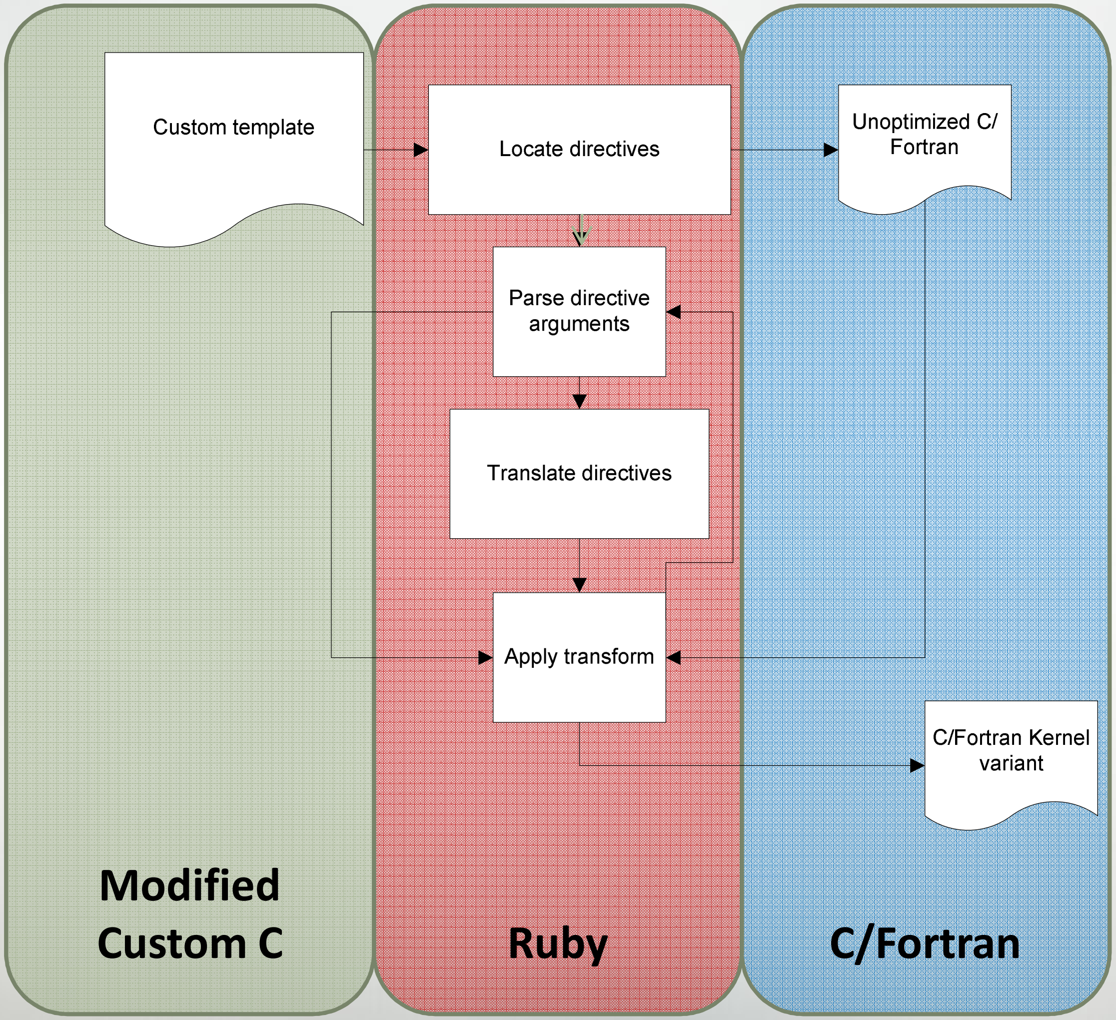


CrayATF – a Generic and Modular Framework

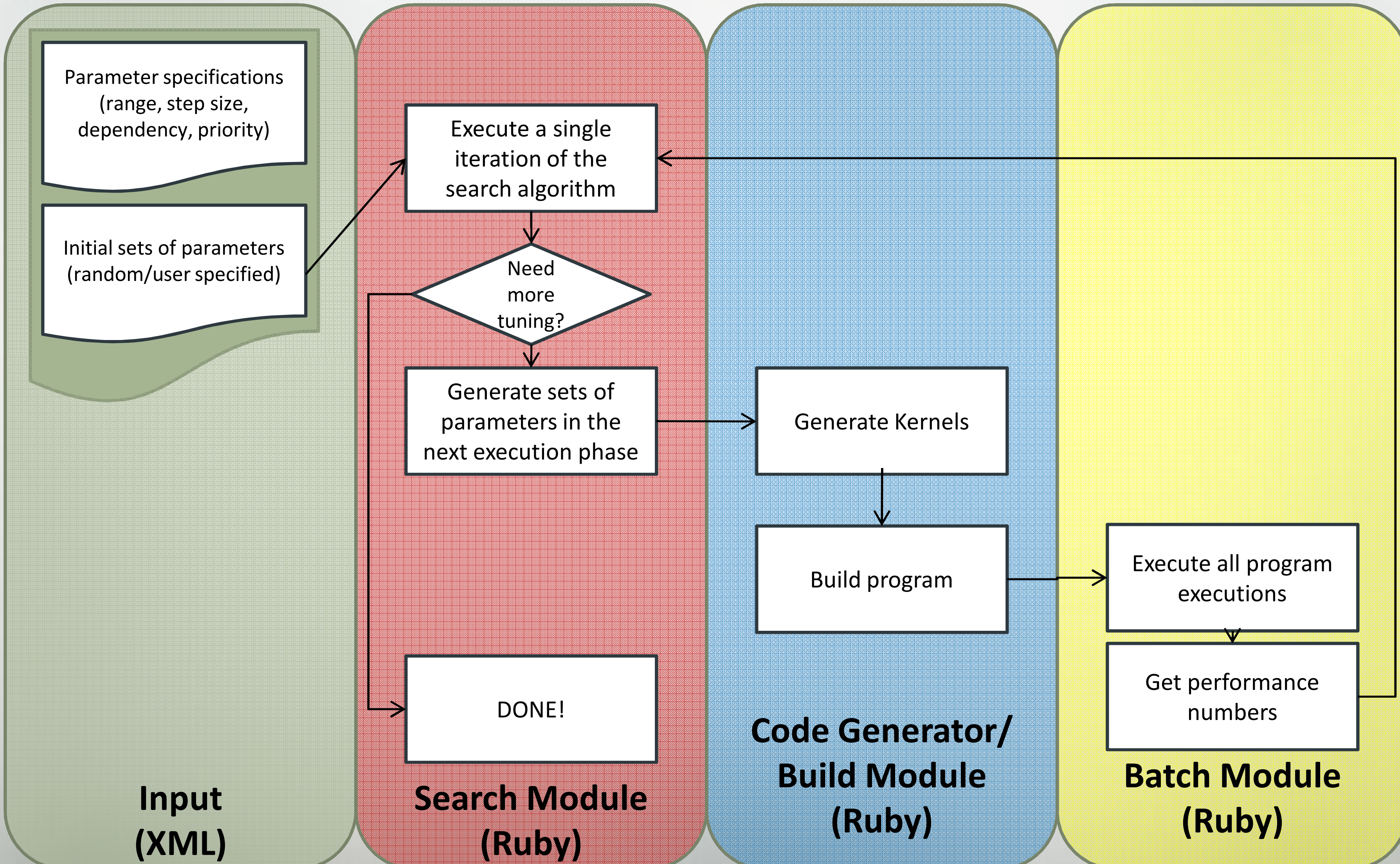


Most importantly – this is
a) extensible
b) replaceable

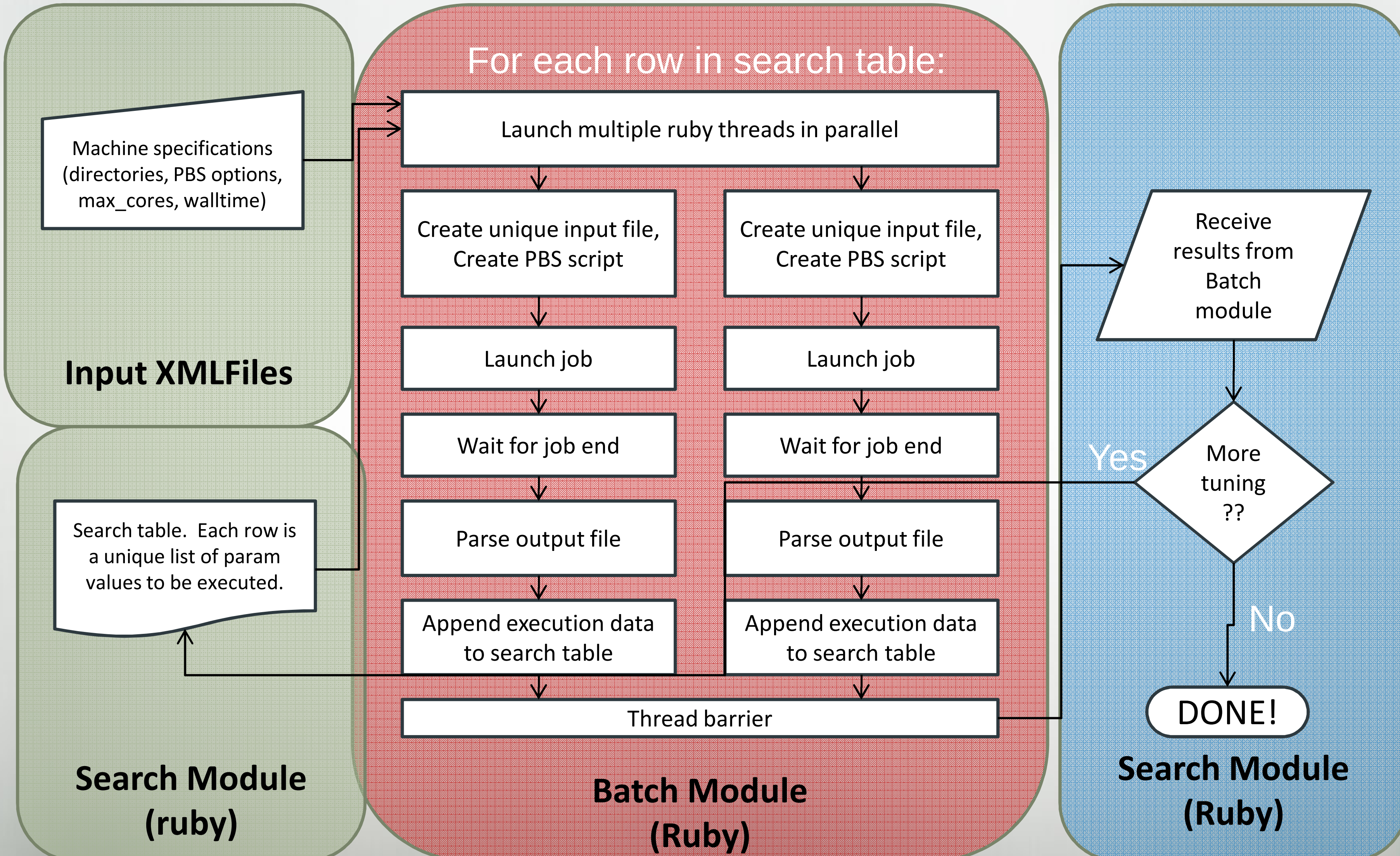
Code Generator



Search Module



Batch Module



Why Ruby ?

- Ruby is the language used for almost all ATF development
- Scripting ability
  - E.g. One-line text replacement of a single file

```
subs.keys.each {|x| filestring.gsub!(x, subs[x]) }
```

- System programming ease:
  - E.g. On Cray XT systems, find all the jobs I have in the queue, and delete them :

```
out = Array.new(`qstat -u #{`whoami`.to_s}`.to_s.to_a)
5.upto(out.length-1){|line|
  system("qdel #{out[line].split('.')[0].to_i}")
}
```

Why Ruby?

- Extremely simple and lightweight threading
  - Threadpool implemented in 40 lines of code includes routines to:
    - Initialize the pool
    - Launch threads
    - Destroy threads
    - Exception handling
- Super-soft typed
  - For non-numerical work, we do not want to be concerned with
    - Datatype conversion
    - Accuracy
    - Performance (!)
  - Allows functional code to be developed very quickly
- Integration with XML for extremely powerful configuration/input methods

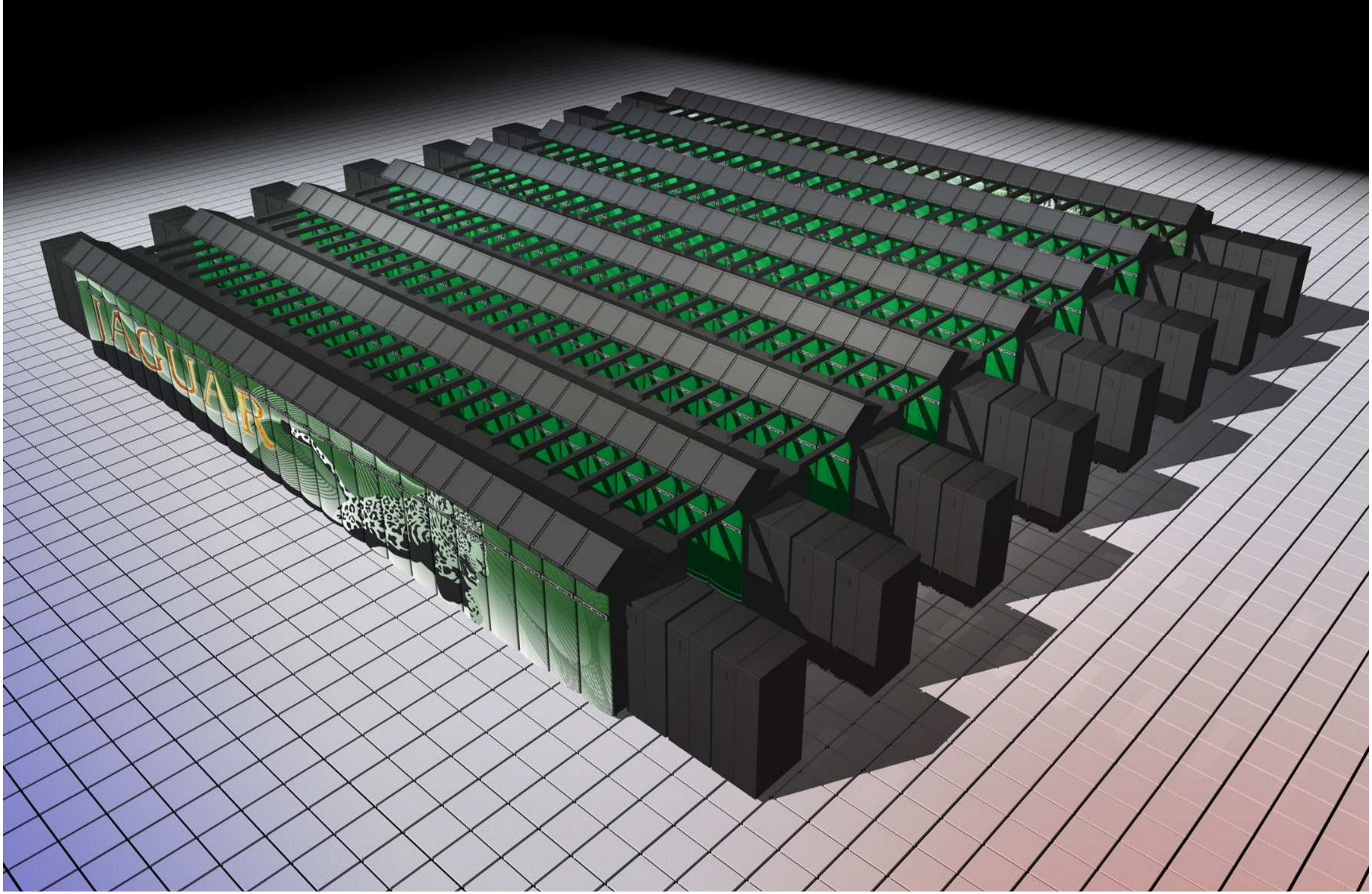
ATF in use – HPL

- High Performance Linpack benchmark
  - Used for top500 rankings
- Traditional tuning approach for HPL :

1. Choose N to fill local memory (reduce comms cost)
2. heavily tune serial dgemm (parallel dgemm dominates)
3. find a good enough parameter combination (trial and error)

- This has been successful in the past, but
- #1 is hard to do when the machine grows so large
- #3 has never been taken very seriously in practice
  - But does have good auto-tuning treatment - Hollingsworth et al

Then Cray delivered a small machine to ORNL

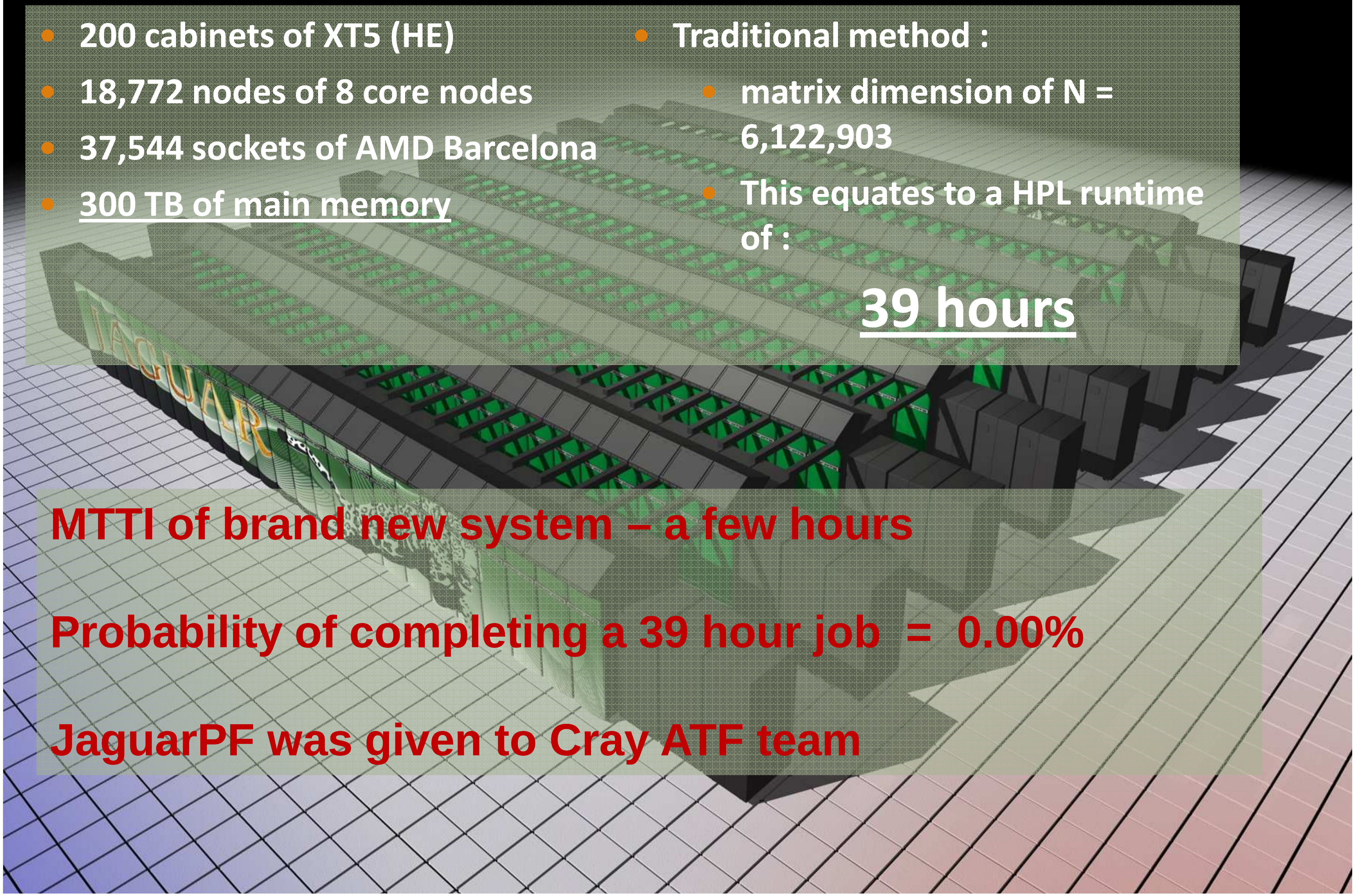


Then Cray delivered a small machine to ORNL

- 200 cabinets of XT5 (HE)
- 18,772 nodes of 8 core nodes
- 37,544 sockets of AMD Barcelona
- <u>300 TB of main memory</u>

- Traditional method :
  - matrix dimension of N = 6,122,903
  - This equates to a HPL runtime of :

<u>**39 hours**</u>



MTTI of brand new system – a few hours

Probability of completing a 39 hour job = 0.00%

JaguarPF was given to Cray ATF team

HPL parameter tuning

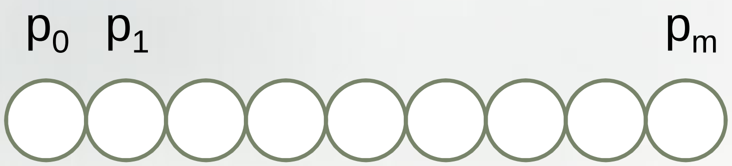
- 17 HPL parameters + Cray's additional parameters + Programming model options
- An example of sensitivity of a single parameter :

NB	bcast	pmap	pfact	nbmin	ndiv	rfact	depth	swap	thresh	transL	transU	EQUIL	align	P	N	time	%peak
160	**1**	0	1	1	3	2	2	0	100	1	0	1	2	140	3429286	483.38	**51.70%**
160	**3**	0	1	1	3	2	2	0	100	1	0	1	2	140	3429286	313.47	**79.71%**

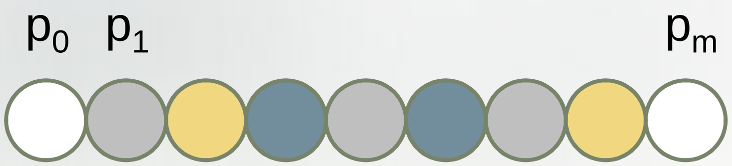
- Exhaustive search space is measured in tens of years runtime
- Typically, studies reduce the search by reducing scale
- However, early progress of ATF-HPL showed that :
- parameter information from small scale does not translate to full scale

“Grouped and Attributed Orthogonal Search”

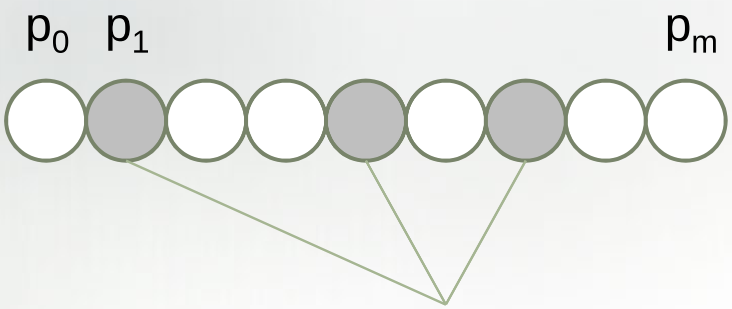
- We can’t consider most search algorithms
  - Within only 5% of optimal would be a disaster for top500 list
- Use Grouped, Attributed, Orthogonal search



1. **Define list of parameters to be studied**

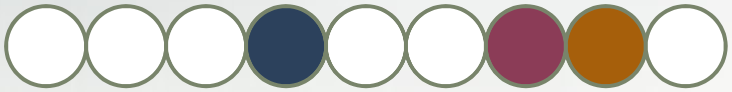


2) Define Groupings between parameters

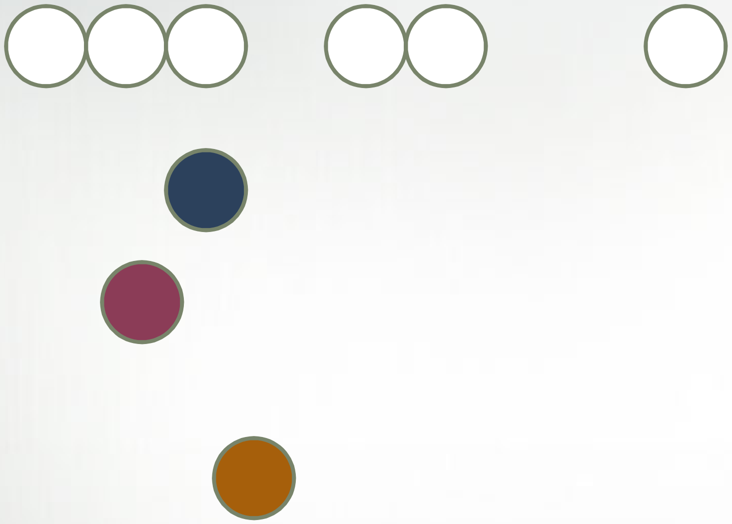


Attributes for this group
•Requires full scale
•Requires small memory
•Can tolerate early completion
needs to be varied wildly

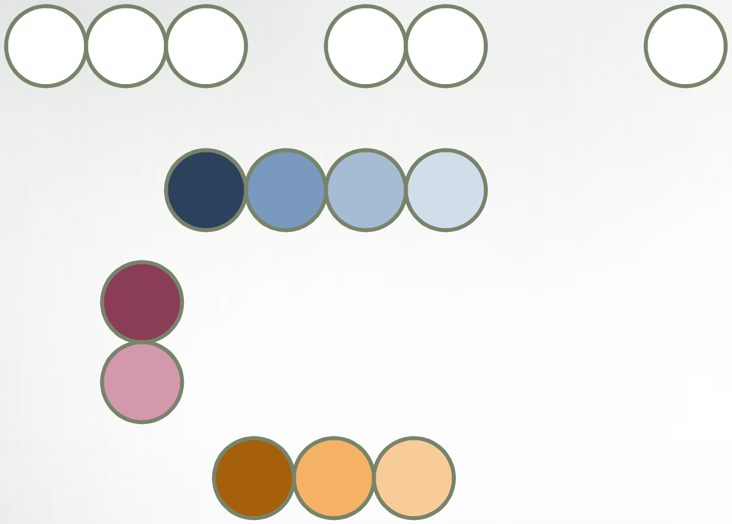
3) Define attributes for each group based on attributes for each parameter



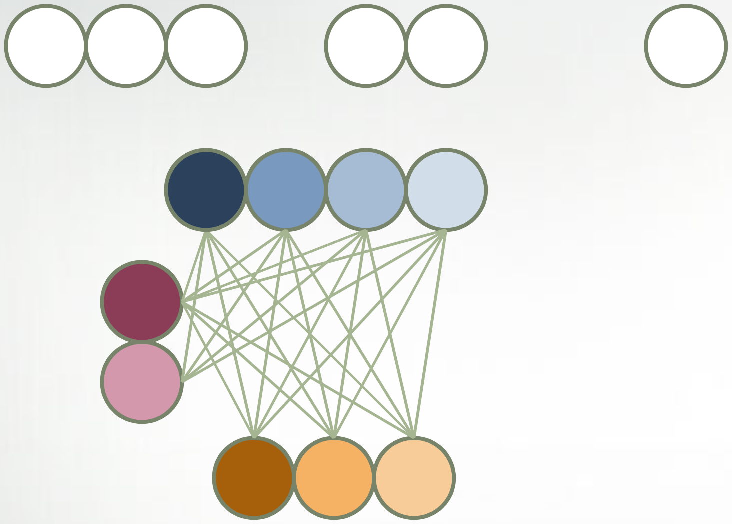
4) **Loop over each group**



4) Loop over each group

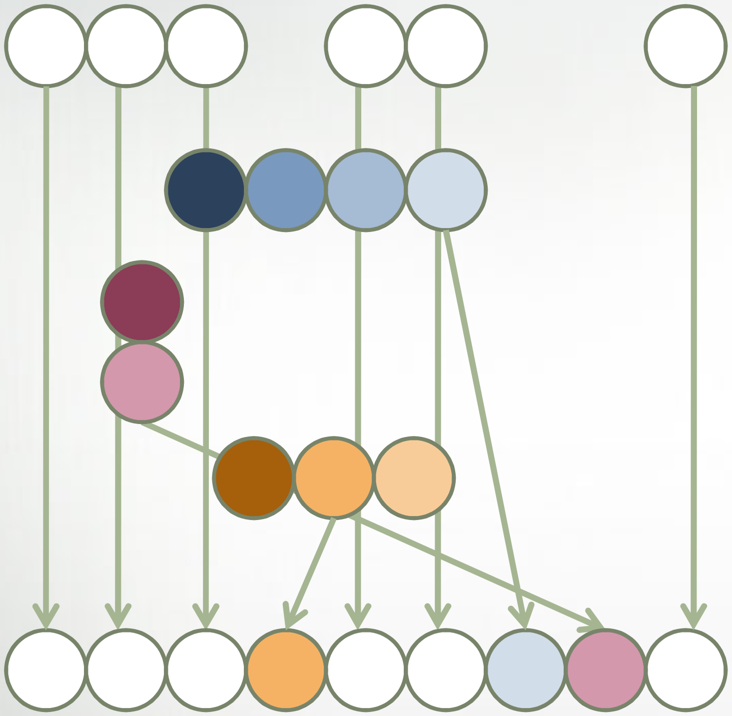


5) Expand length of each parameter



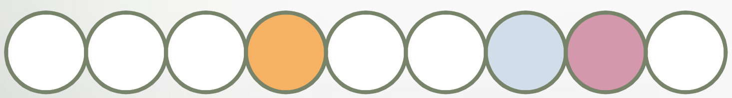
5) Perform Search within group

(holding all other parameters steady)

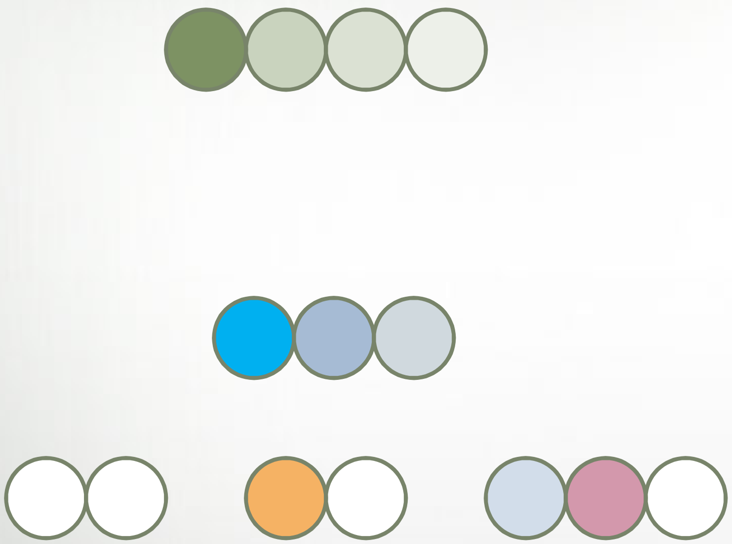


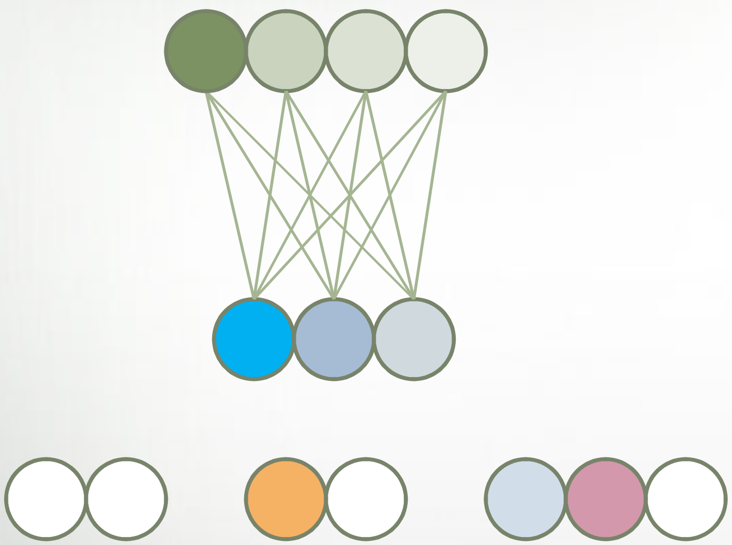
5) Take the best performing result & carry the best parameter values to the next search

5) Define next search group (keeping best from last search)



CRAY
THE SUPERCOMPUTER COMPANY





In practice

- GOAS gets very close to optimal
- At expense of large search space
- At expense of huge amount of man-power

- Knowledge of hardware and algorithm allows very sensible selection of groups
  - Reduces the search space by knowledge

- Although it is not elegant, GAOS cannot be beaten in our tests

ATF in use - Nov 2008 top500 list

Rank	Site	Vendor	Cores	RMax	RPeak	Nmax	Power
1	DOE/NNSA/LANL	IBM	129600	1105000	1456700	2329599	2483.47
2	Oak Ridge National Laboratory	Cray Inc.	150152	1059000	1381400	4712799	6950.6
3	NASA/Ames Research Center/NAS	SGI	51200	487005	608829	2300760	2090
4	DOE/NNSA/LLNL	IBM	212992	478200	596378	2456063	2329.6
5	Argonne National Laboratory	IBM	163840	450300	557056	2580479	1260
6	Texas Advanced Computing Center	Sun	62976	433200	579379	0	2000
7	NERSC/LBNL	Cray Inc.	38642	266300	355506	1612399	1150
8	Oak Ridge National Laboratory	Cray Inc.	30976	205000	260200	2466816	1580.71
9	NNSA/Sandia National Laboratories	Cray Inc.	38208	204200	284000	2500000	2506
10	Shanghai Supercomputer Center	Dawning	30720	180600	233472	0	0

Cray Adaptive Sparse Kernel (CASK)

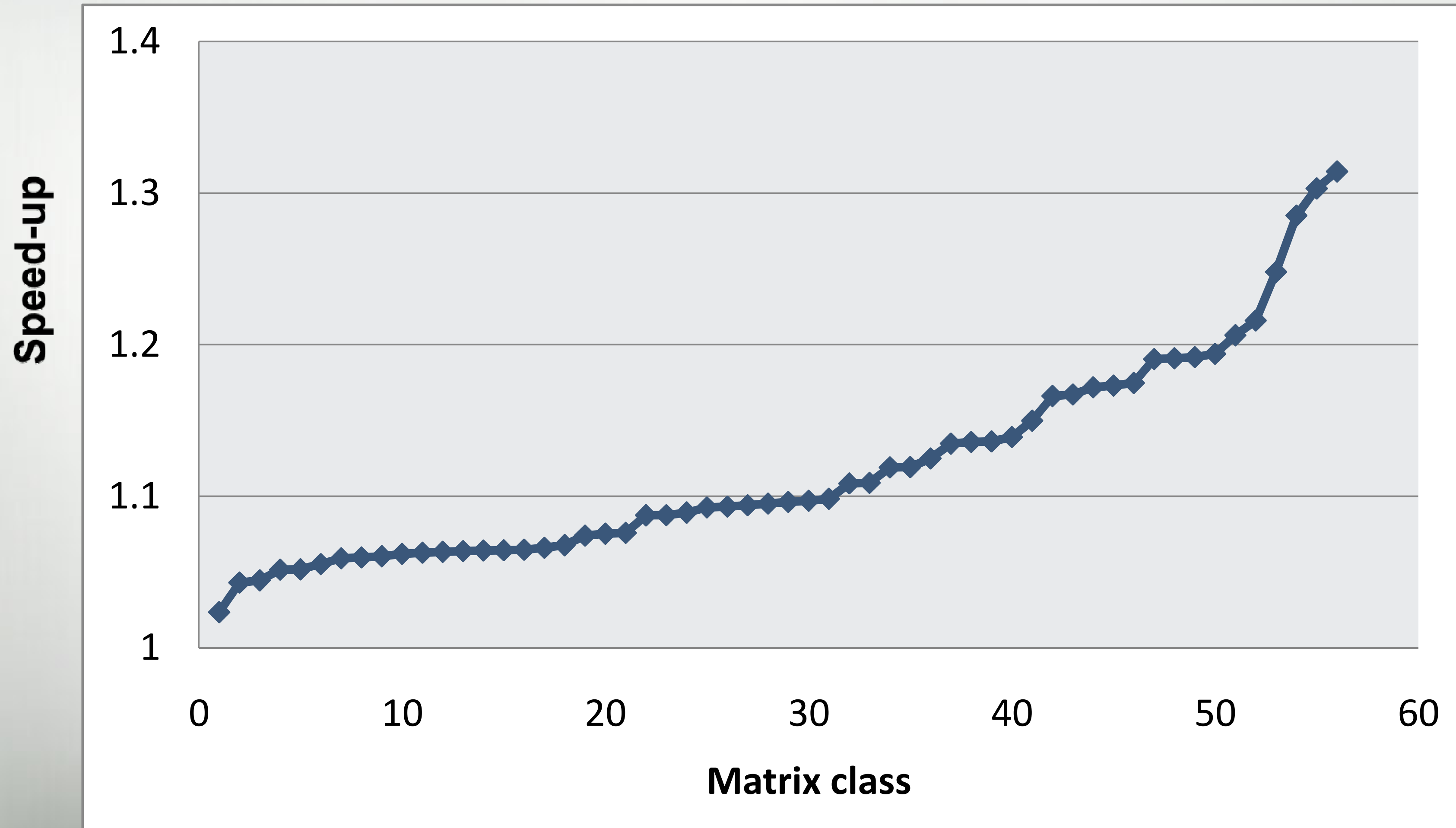
- Cray Adaptive Sparse Kernels – The Crown Jewel of CrayATF
- The CASK Process

    1. Offline – produce all code variants for tuning strategy
    2. Offline – define target matrix classifications
    3. Offline – produce performance model for given matrix class
    4. Runtime – analyze matrix and deduce classification
    5. Runtime – assign tuned kernel to user code

- CASK silently sits beneath PETSc on Cray systems
    - Trilinos support coming soon
- CASK ATF flow looks very like the flow shown earlier
- **CASK released with PETSc 3.0 in February 2009**
    - Generic and blocked CSR format

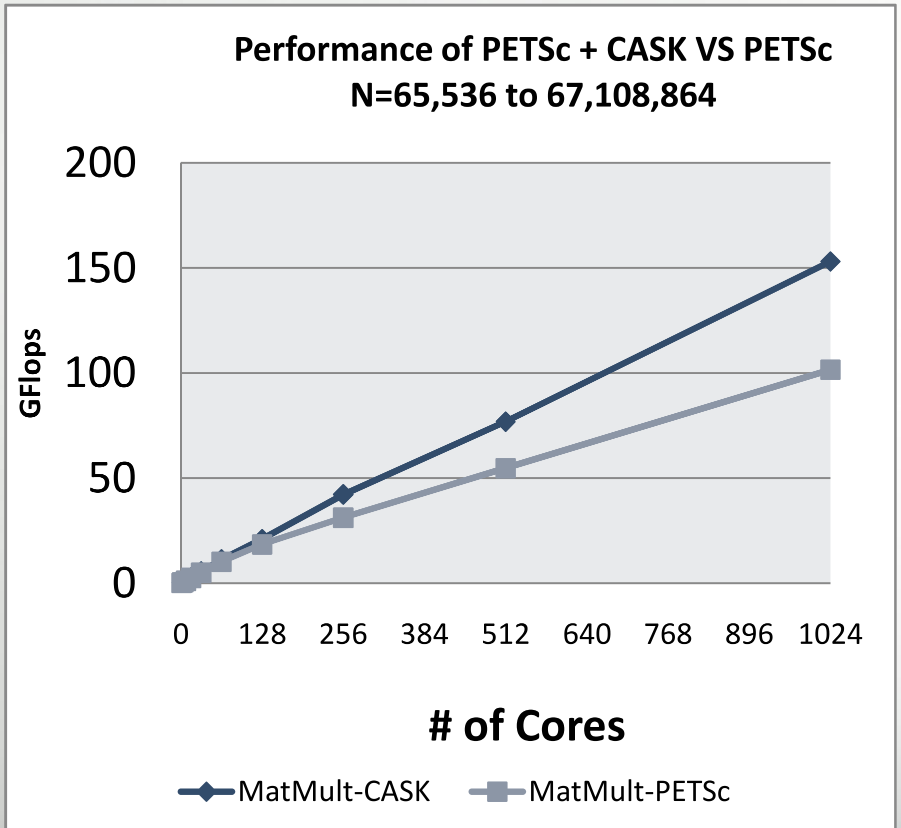
CASK and PETSc: Single Node XT5

Speed-up of PETSc + CASK versus PETSc
Speedup on Parallel SpMV on 8 cores
60 different matrix classifications

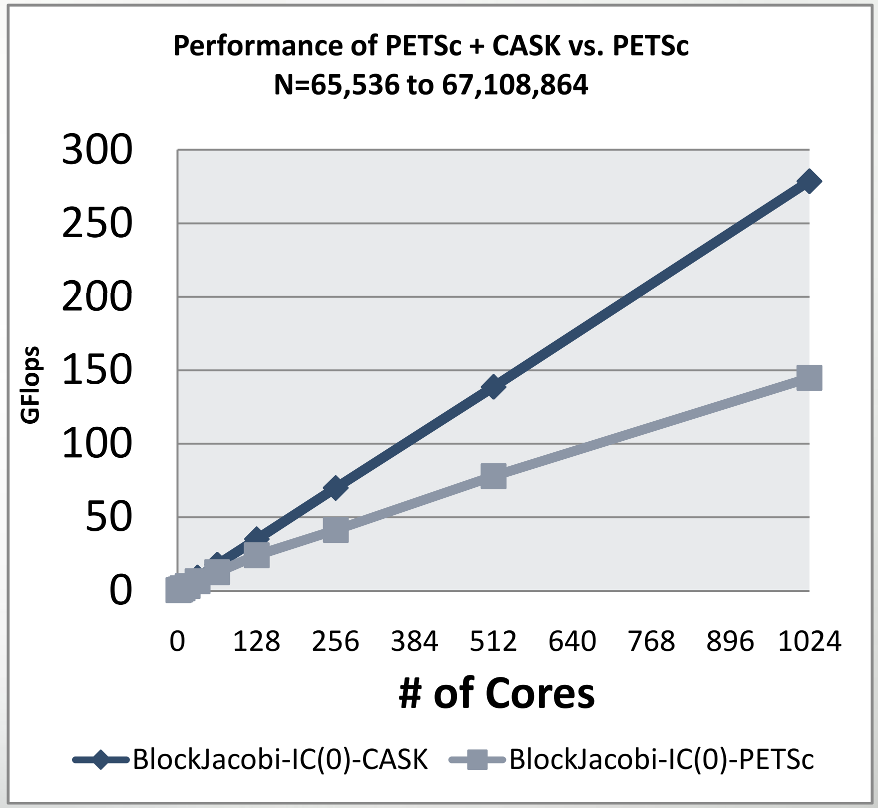


CASK Scalability (XT4)

SpMV performance only



Full solver with incomplete Cholesky local preconditioning



Lessons

- When you build an infrastructure for “industrial” purposes
  - Search spaces should be manipulated via your knowledge of hardware
  - At least 50% of the effort is pure software engineering
  - Languages like Ruby and Python make things realistic
  - Should not get too attached to what is “auto-tuning”
    - Whatever works for our problems is what we need to do
    - We do not care about definitions
    - Search algorithms are only interesting if they help us achieve our goals
    - It seems that there are emerging several distinct sub-classes of auto-tuning
- We found new uses for auto-tuning in the process :
  - Sanity/stability testing of new hardware
  - Excellent regression test for libraries

Clearing up

- ATF is not a generalized auto-tuner for scientific applications
  - It is practical design tailored for vendor tuning of libraries
- We did not make auto-tuning easy
  - In our case, it required one of the best teams in the industry to be 100% devoted for many many months
- CrayATF is in its infancy, not nearing completion

Next steps

- Cray will provide hybrid next generation XT system with GPUs in 2010
- Cray will provide a HPC Programming Environment for hybrid system

- On this system, the tuning approach is HIGHLY parameterized
  - Which algorithm is best?
  - Number of blocks per matrix?
  - How much matrix to GPU/CPU?
  - How to schedule transfer to GPU?
  - Number of threads per block?
  - Shape of threads per block?
  - What type of memory to use?

- ATF for GPUs will be main approach into library GPU tuning

Thankyou

- Q&A