

Skeletons and the Parallel Programming Challenge

Murray Cole



Overview

- The Parallel Programming Challenge
- Skeletons to the Rescue?
- Current work in Edinburgh

Parallel Programming Challenge

Mainstream Parallelism

- Parallelism is now a mainstream reality
 - Chip manufacturers' roadmaps now look to **increase core count rather than clock rate** (clocks may *slow down* to save energy)
 - GPGPU devices offer massive on-die parallelism (with SIMD-like constraints)
 - Soon even on-chip manycore will take on aspects of “distributed” parallelism (eg Intel's Single Chip Cloud Computer)
- We may have **three or four layers** of parallelism

Haven't we heard this before?

- HPC parallelism has sought a solution for many years, but has ended up “making do” with MPI, OpenMP (and now OpenCL/CUDA).
 - These are expert-labour-intensive, awkward to interface and produce code which is not very performance portable.
- This time we are in the mainstream. This makes it a big deal!
 - Without a productive solution, we will not be able to use the available resources effectively.
 - **Intel and Microsoft may go bust....**

Skeletons to the Rescue?

Skeletons to the Rescue?

- Key observation: **Many parallel applications involve customised instances of generic algorithmic patterns. Let's abstract and package these.**
 - Farm, Pipe, D&C, Stencil, DSLs...
- Separate *software productivity layer* (instantiation and composition of skeletons) from *expert performance programming layer* (skeleton implementation, exploiting knowledge of constrained computational structure and target architecture).

Skeletons to the Rescue?

- This approach becomes even more appealing in the era of multilayer parallelism:
 - Application programmer is happy not to have to write the **coordination glue** (in different models!)
 - Expert programmer is happy that the application programmer has been prevented from writing the coordination glue and **overspecifying** the implementation.
- If we can demonstrate that this **works** and is **widely applicable**, then ***we win a very big prize.***

Status Report

- Skeletons research has been active for 20+ years. Are we having any **impact**? Is the wider world starting to think the same way?
- How can we achieve **greater** impact?

Is anybody listening?

- In a broad sense, yes
 - MapReduce
 - Intel TBB / Microsoft Task Parallel Library
 - MPI collectives? OpenMP loop directives?
 - DSLs like StreamIt?
 - Mattson et al book on Patterns, Our Pattern Library ...
- But these cover a rather small set of patterns.
Is that it?

Achieving Greater Impact

- Look at **Parallel Benchmarks**?
 - Splash, NAS PB, SPEC OMP, SPEC MPI, Parsec, Lonestar, Mediabench
 - Non-trivial to convince, since we rewrite the source, but greater credibility if we achieve it?
 - Do these exhibit skeletal structure? Lots of farms, bag-of-tasks, stencils, some simple D&C, some pipelines, lots of irregularity.
- Demonstrate **multi-layer performance portability**

Current Work in Edinburgh

Current Work in Edinburgh

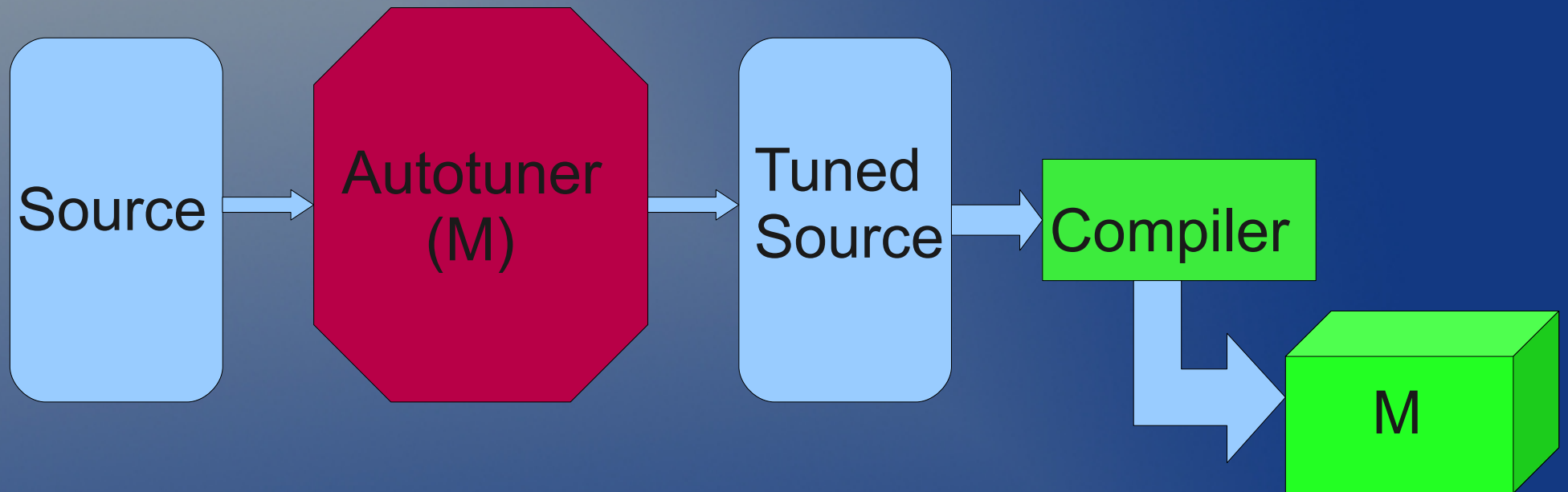
- We are trying to exploit the “**skeletons as providers of structural information**” angle, to demonstrate skeleton-enabled performance optimisations.
- We plan to combine this work with that of our *machine-learning-led autotuning* group, to improve transparent performance portability.
- Initial case studies: a **worklist** skeleton (on NUMA transactional memory) and a **stencil** skeleton (exploiting OpenCL for GPUs).

Forget about parallelism

Autotuning – Basic Idea

- Assumption: If a program will run for a very long time, or very many times, it is worth spending a long time optimising it.
- Given
 - Source program S , including a number of **tuning knobs** (eg tiling controls, block sizes, loop re-orderings, alternative algorithms ...);
 - A target machine M and compiler C
- Find settings for each knob which optimise the performance of S when compiled by C for M .

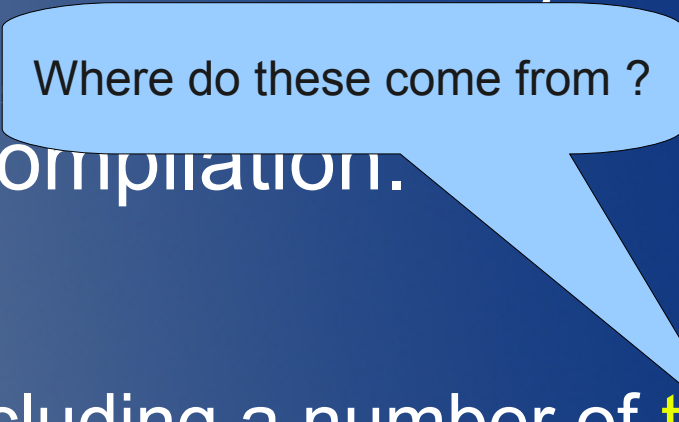
Autotuning – Basic Idea



Autotuning – Basic Idea

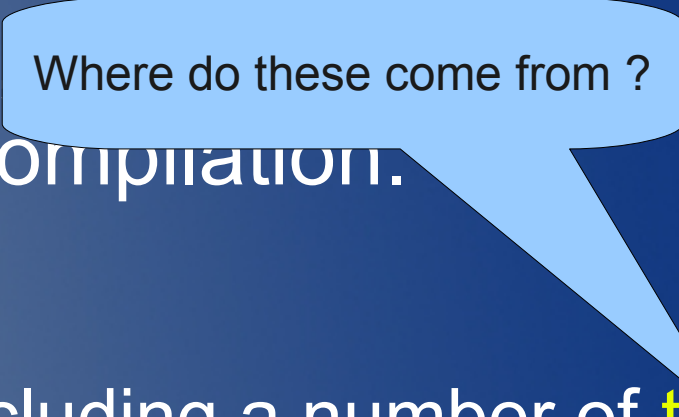
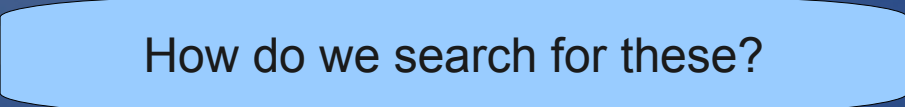
- Principle: If a program will run for a very long time, or very many times, it is worth spending a long time optimising it.
- Given
 - Source program S , including a number of **tuning knobs** (eg tiling controls, block sizes, loop re-orderings, alternative algorithms ...);
 - A target machine M and compiler C
- Find **settings** for each knob which optimise the performance of S when compiled by C for M .

Autotuning – Basic Idea

- Principle: If a program will run for a very long time, or very many times, it is worth spending a long time optimising its compilation.

Where do these come from ?
- Given
 - Source program S , including a number of **tuning knobs** (eg tiling controls, block sizes, loop reorderings, alternative algorithms ...);
 - A target machine M and compiler C
- Find **settings** for each knob which optimise the performance of S when compiled by C to M .

Autotuning – Basic Idea

- Principle: If a program will run for a very long time, or very many times, it is worth spending a long time optimising its compilation.

- Given
 - Source program S, including a number of **tuning knobs** (eg tiling controls, block sizes, loop re-ordering, ...);

 - A target machine M and compiler C
- Find **settings** for each knob which optimise the performance of S when compiled by C to M.

Simplistic Autotuning

- Application programmer (or for a library, the expert library programmer) explicitly indicates the tuning knobs.
- Enumerate, compile **and run** all points in the program space implied by the tuning knobs, and pick the best.
- Repeat every time the architecture changes.
- This is what libraries like ATLAS and FFTW do.
- But what if the search space gets too big?
(some knobs may be numeric)

Avoiding the Search Space

- One approach is for the programmer to embed heuristics which capture the right decisions **explicitly within the source code**:

```
if (someSize > THRESHOLD) {  
    techniqueA;  
} else {  
    techniqueB;  
}
```

- This is difficult, particularly if we need to capture **relationships between tuning knobs**, and heuristics are probably **machine-specific**.

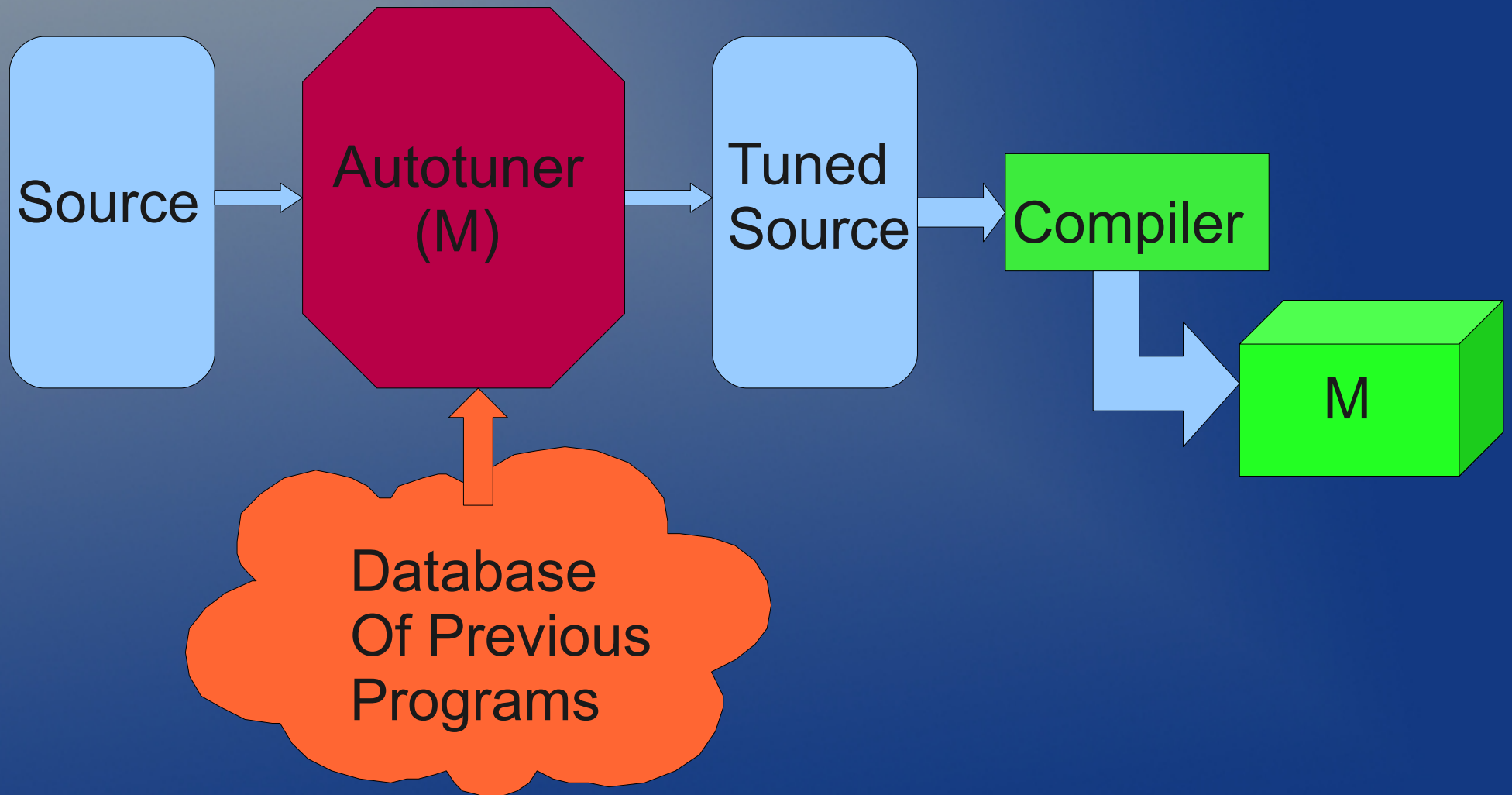
Pruning the Search Space

- An alternative is to try a “Machine Learning” approach, in which we try to **learn** (ie statistically correlate) the correct tuning decisions for a given C and M

```
if (CONDITION TO BE LEARNED) {  
    techniqueA;  
} else {  
    techniqueB;  
}
```

- Principle: source S will respond well to knob settings which produced good results for other previous programs which are “**similar**” to S.

Autotuning – Basic Idea



Pruning the Search Space

- An alternative is to try a “Machine Learning” approach, in which we try to learn the correct tuning decisions

```
if (CONDITION TO BE LEARNED) {  
    techniqueA;  
} else {  
    techniqueB;  
}
```

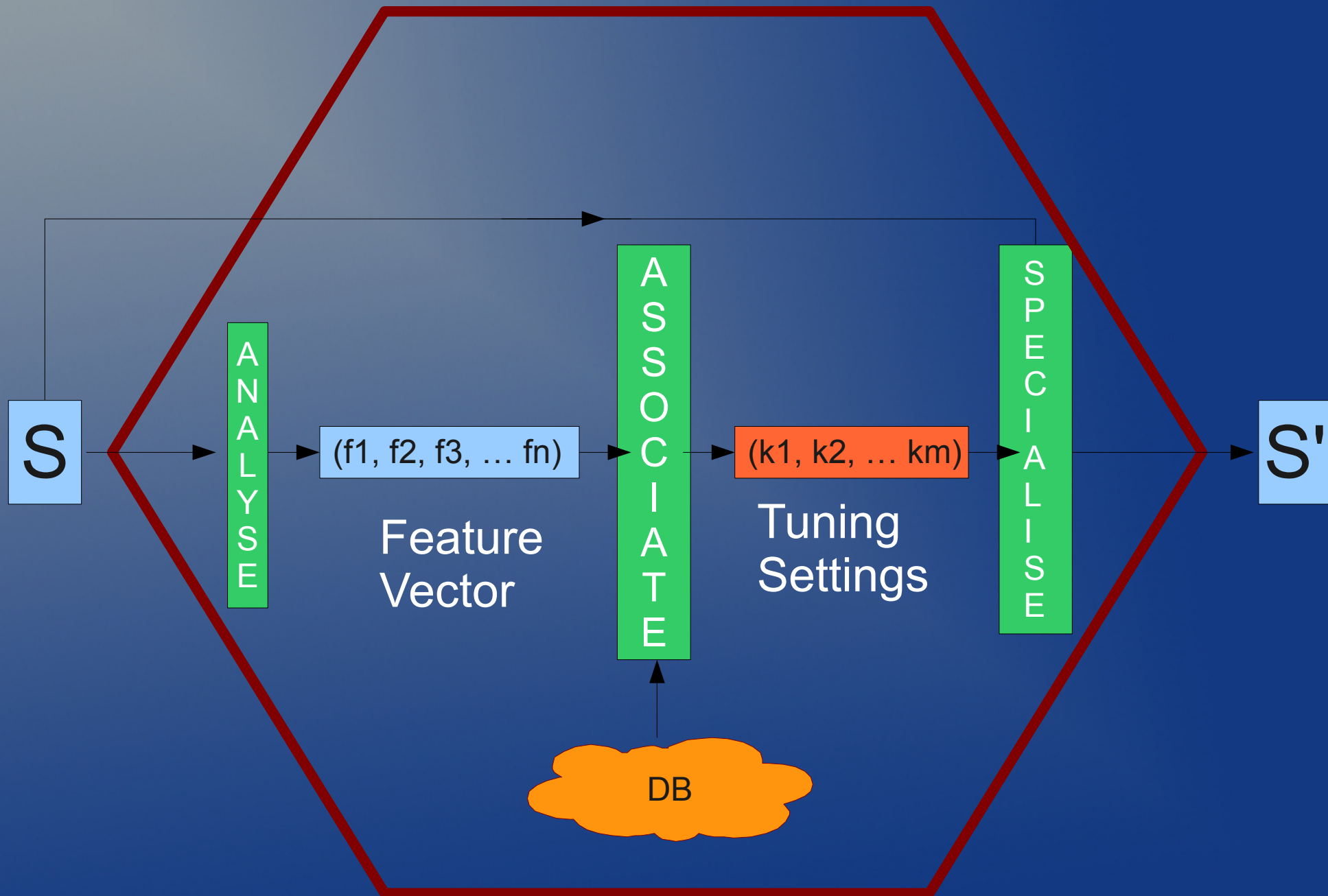
How do we capture similarity?

- Premise: program *S* will respond well to knob settings which produced good results for other previous programs which are “similar” to *S*.

Features

- In Machine Learning terms, we have a **classification** task, requiring us to partition the set of programs (possibly + input/size) by responsiveness to tuning knob settings.
- We choose a set of “**features**” whose values will act as abstract representations of programs.
- Typically we will use a mixture of **static** and **dynamic** features, eg basic block size, branch complexity, data sizes, loop counts, cache behaviour....
- Finding a good feature set is hard.

Machine Learning Autotuner



The Box of Tricks

- Various techniques, all of the form
 - Learning phase: take a collection of programs and compile and run these at length on the target machine, gathering statistics relating features and tuning settings to quality of outcome. **(Slow, but one-off and automated)**
 - Application phase: take a new program P, deduce its feature vector, classify it against learned data and select tuning settings. **(Fast!)**
- New machine? Learn again.....automated!

Success Stories

- Rapidly Selecting Good Compiler Optimizations using Performance Counters, Cavazos, O'Boyle et al, CGO 2007.
 - Sequential C programs from SPEC
 - Knobs: gcc flags
 - Features: various hardware performance counters (ie dynamic), cache hits, loop counts, branch predictions
- 17% improvement over “highest” opt. setting

Success Stories

- Mapping Parallelism to Multi-cores: a Machine Learning Based Approach, Wang and O'Boyle, PPOPP09.
 - OpenMP programs (parallel for) targeting Xeon/Cell multicore
 - Knobs: loop scheduling policy, #threads
 - Features: (Static) instruction type counts, Dynamic) profile counters as above
 - 37% improvement over OpenMP default

Success Stories

- A Case for Machine Learning to Optimize Multicore Performance. Ganapathi et al, HotPar09.
 - Hand annotated stencil codes on multicore
 - Knobs: #threads, blocking, prefetching
 - Features: The usual suspects....
 - “up to” 18% improvement (run time) over expert

Now consider parallelism

Autotuning Parallel Programs

- If we were to consider Machine Learning autotuning of general parallel programs there would be two big issues:
 - How do we find appropriate **tuning knobs**?
 - How do we find a relevant **features**?

Skeletons to the Rescue (we hope)

Skeletons and Autotuning

- How do we find appropriate **tuning knobs**?
 - This becomes the expert programmer's task. The tuning knobs are embedded in the implementation of the skeleton.
- How do we find relevant **features**?
 - This is still hard. The constrained nature of skeletons may make it easier, but the fact that we are now dealing with classes of program may make it harder.

Case Study: A **Worklist** Skeleton

- Derived from transactional memory, irregular parallelism oriented benchmarks STAMP and Lonestar, (by PhD student Fabricio Goes)
 - A bag of tasks (the worklist).
 - An irregular, dynamic graph of data points.
 - Execute tasks in parallel (any order), possibly generating new tasks, until all done
 - Task may update point and its neighbours.
 - Suitable for Transactional Memory: tasks may but typically don't conflict, need to be careful

Case Study: A **Worklist** Skeleton

- Tuning knobs (multicore implementation)
 - **Privatized Worklists** with stealing (or not) (reduce contention, reduce abort ratio?)
 - **Helper threads** to enable prefetching (more productive use of cores once natural parallelism is exhausted)
 - **Transactional granularity** (how many tasks per transaction?)
 - **Abort policy** (can choose whether to retry with a different task)

Case Study: A **Worklist** Skeleton

- Search space exploration
 - 16 core SMP, various STM systems
 - Four applications from the STAMP set
 - Distributed work pool is a good idea in general
 - Other optimizations vary in their effectiveness (both alone and in combination) from app to app
 - Next challenge will be to **learn** which features can determine the right choice

Case Study: A **Stencil** Skeleton

- (PDRA Chris Fensch)
- Applications in Simulation, Image Processing ...
 - Multi-dimensional cartesian data space
 - Each point hosts the same typed fields
 - Use a “stencil” defining a fixed neighbourhood of “close” points which will contribute to local computations
 - Iteratively, and in lockstep, apply stencil ops at every point in the space
 - Terminate after some number of iterations, or upon reaching some condition, determined by combining state at each point.

Case Study: A **Stencil** Skeleton

- Initial goal: allow targeting of both multicore and GPU architectures
 - Using OpenCL as the implementation medium allows us to target both models: the skeleton hides the memory management code which complicates OpenCL
 - Using OpenCL for the app programmer's interface (but only sequential pointwise code) allowed the OpenCL compiler to generate good SSE aware object code

Case Study: A **Stencil** Skeleton

- Next challenge: **tuning knobs**
 - Data layout (“array of structs”, “struct of arrays”, other communication/cache/GPU friendly layouts)
 - Tiling factors (how to distribute and traverse the implied iteration space)
 - Layers of parallelism (use/don't use multiple nodes, multiple cores, GPU)
 - Numbers of processes, threads

Summary

- Any technology which makes a contribution to the provision of **productive** parallelism which is **transparently performance portable** across **multiple layers** can make a big impact.
- Skeletons, or at least skeleton principles, may be such a technology, but we need to push forward now, **demonstrating applicability** to real problems, or at least credible benchmarks.
- Slogan:
“abstraction + specialisation = performance”