Maximizing Speedup through Self-Tuning of Processor Allocation

Thu D. Nguyen, Raj Vaswani, and John Zahorjan
University of Washington

Presenter: 정 연
contents

+ Introduction
+ Experimental environment
+ Algorithms for Self-tuning
+ Performance
+ Conclusion
Introduction

Many parallel applications achieve maximum speedup at some intermediate allocation

- Job’s speedup does not increase monotonically with # of processors

Goal: dynamically determine # of processors

- Unused processors are released back to the system

Why dynamic?

- It may not be possible a priori to determine the best allocation
- No static allocation may be optimal for the entire execution lifetime of a job
How to?

- Dynamically measures job efficiencies at different allocations
- Uses these measurements to calculate speedups
- Automatically adjusts a job’s processor allocation to maximize its speedup
- Relies on appropriate HW support to measure efficiencies
- Method of golden sections (MGS) optimization technique
  - MGS finds the maximum of a unimodal function over a finite interval by iteratively using computed function values to narrow the interval in which the maximum may occur
- Iterative parallel applications
  - Majority of the execution is driven by a sequential loop
  - Measurements taken for a particular iteration are good predictors of near future behavior

This process is called self-tuning
Experimental environment

- Kendall Square Research KSR-2 COMA shared memory multiprocessor
  - KSR KAP preprocessor
  - KSR PRESTO runtime system

- HW monitoring unit available on each node of the KSR-2 to perform runtime measurements of application efficiency
  - Event monitor
    - Via read-only registers

- 10 parallel applications
  - Hand-coded applications
  - Complier-parallelized applications
Runtime measurements

- A number of different metrics
  - Execution time
  - Efficiency
    - Directly related to speedup
- Loss of efficiency in shared memory systems due to
  - Parallelization overhead
    - Typically small
  - System overhead
  - Idleness
  - Communication
    - Occurs when required data does not reside in local cache
    - Directly related to the processor stall

2010-12-20
Per-noise event monitors on the KSR-2 maintain three critical HW counters
- Elapsed wall-clock time
- Elapsed user-mode execution time
- Accumulated processor stall time

Measuring system overhead and processor stall time
- Reading these three register at the beginning and end of each iteration

Measuring idleness
- Instrument all PRESTO and CThreads synchronization code to track elapsed idle time using the wall-clock counter
  - Assume all application synchronization takes place through calls to the PRESTO and CThreads libraries rather than through direct manipulation of shared variables
Efficiency = 1 - loss of efficiency
  - Efficiency = 1 - system overhead - idleness - processor stall

Speedup = # of processors * efficiency
Algorithms for Self-tuning

A basic Self-Tuning algorithm

+ MGB
  - Searches for the maximum of a unimodal function over a finite interval
    - By iteratively using computed function values to narrow the interval in which the maximum may occur
Algorithms for Self-tuning(2)

- Algorithm begins by executing one iteration using all P available processors
  - Search domain reduces to \([S(P), P]\)

- Non-unimodal speedup functions?
  - Most speedup functions are unimodal over substantial ranges of processors
  - Simple greedy heuristic to deal with non-unimodal speedup functions
    - Upon encountering non-unimodal case, simply continue the search in the largest subinterval for which the measured speedups are conformal with a unimodal function, and which contains the largest speedup found so far
    - This worked well in practice
Refining the basic Self-Tuning approach

+ **Validity check for three assumptions**
  1) For non-unimodal speedup functions, heuristic-based extended MGS search procedure will correctly locate the global maximum
  2) Speedup is not a function of time
  3) The speedup values of successive iterations are directly comparable

+ **Change-driven self-tuning Algorithm**
  o Continuously monitors job efficiency and re-initiates the search procedure whenever it notices a significant change in efficiency

+ **Time-driven self-tuning Algorithm**
  o Includes change-driven self-tuning
  o Will also rerun the search procedure periodically regardless of change in job efficiency
  o Considering the possibility that job efficiency changes in the middle of a change-driven self-tuning search
Performance

- Self-tuning imposes very little overhead
- Basic self-tuning can significantly improve performance over no-tuning
Change-driven self-tuning can significantly improve performance over basic self-tuning.

Time-driven self-tuning is not useful for the applications studied here.

The performance benefit of self-tuning can be limited by the cost of probes.
Multi-phase Self-tuning

- The iterations of some applications are composed of multiple parallel phases
  - Phases: specific piece of code
    - A parallel loop in a compiler-parallelized program
    - A subroutine in a hand-coded parallel program

- Assume that on each entry to and exit from a phase, the runtime system is provided with the unique ID of the phase
  - Find a processor allocation vector \((p_1, p_2, \ldots, p_N)\) that maximizes performance when there are \(N\) phases in an iteration
Multi-phase Self-tuning(2)

**Independent multi-phase self-tuning (IMPST)**
- Merely apply self-tuning to each phase independently
- Simple
- Problem: performance of each phase depends only on its own allocation and not on the allocations for any other phases
  - Allocation in phase n dictates a problem partitioning
  - Partitioning affects the amount of communication required in phase n+1

**Inter-dependent multi-phase self-tuning (DMPST)**
- Simulated annealing and a heuristic-based approach
- Randomized search technique
  - Choosing an initial candidate allocation vector
  - Selecting a new candidate vector (apply random multiplier)
  - Evaluating and accepting new candidate vectors until steady state
  - Terminating the search
Multi-phase Self-tuning (3)

Multi-phase techniques are able to achieve performance not realizable by any fixed allocation.

+ Inter-dependent self-tuning yields better performance than independent self-tuning.
Conclusion

- Maximizing application speedup through runtime, self-selection of an appropriate number of processors on which to run
  - Based on ability to measure program inefficiencies
  - Peak speedups are data or time dependent

- Simple search procedures can automatically select appropriate numbers of processors
  - Relieves the user of the burden of determining the precise number of processors to use for each input data set
  - Potential to outperform any static allocation
    - Dynamic
Q&A