MAXIMIZING SPEEDUP THROUGH SELF-TUNING OF PROCESSOR ALLOCATION

Thu D. Nguyen, Raj Vaswani, and John Zahorjan
University of Washington
Presenter: 조현범
Contents

- Introduction
- Experimental environment
- Algorithms for Self-tuning
- Performance
- Conclusion
Many parallel applications achieve maximum speedup at some intermediate allocation

- Not monotonic with # of processors
**Goal**: dynamically determine # of processors

- Unused processors are released back to the system
- 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
- Method of golden sections (MGS) optimization technique
- Only consider 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
A number of different metrics
  - Execution time
  - Efficiency
    - Directly related to speedup

Loss of efficiency in shared memory systems due to
  - Parallelization overhead
  - System overhead
  - Idleness
  - Communication
    - Occurs when required data does not reside in local cache
    - Directly related to the processor stall
Three critical HW counters for measuring system overhead and processor stall time

- Elapsed wall-clock time
- Elapsed user-mode execution time
- Accumulated 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

\[ E(p) = 1 - \frac{WT(p) - UT(p)}{WT(p)} - \frac{IT(P)}{WT(p)} - \frac{PST(p)}{WT(p)} \] (1)

Speedup = # of processors * efficiency

\[ S(p) = p \times E(p) \]
**MGS**

- 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
Assumption
  - Speedup is a single variable function, $S(p)$
  - $S(p)$ can be calculated using equation 2 for any $p$, $1 \leq p \leq P$, by measuring $E(p)$ for any one iteration
  - Single variable optimization

Reducing search interval
  - Search domain reduces from $[1, P]$ 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.
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
- Continuously monitors job efficiency and re-initiates the search procedure whenever it notices a significant change in efficiency

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

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 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