Maximizing Speedup through Self-Tuning of Processor Allocation

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

Presenter: Myeongcheol Kim
Contents

- Introduction
- Experimental Environment
- Algorithms for Self-tuning
- Performance
- Conclusion
Introduction

- Does job’s speedup increase monotonically with the number of processors?
  - No, many parallel applications achieve maximum speedup at some intermediate allocation.
Introduction

• 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

• **Goal: dynamically determined # of processors**
  • To maximize job’s speedup
  • Unused processors are released back to the system
Introduction

• 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

• This process is called self-tuning
Experimental Environment

• Kendall Square Research KSR-2 COMA shared memory multiprocessor
  • OS: OSF/1 (UNIX variant)

• H/W monitoring unit available on each node of the KSR-2 to perform runtime measurements of application efficiency
  • Event monitor
    • Provides cache misses, processor stall time, etc. via read-only registers

• 10 parallel applications
  • Hand-coded applications
  • Complier-parallelized sequential program
    • Only consider iterative parallel applications
Runtime Measurements

• A number of different runtime 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
Runtime Measurements

• 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
Runtime Measurements

- Efficiency = 1 – (loss of efficiency)
  - Loss of efficiency = system overhead + idleness + processor stall

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

- Speedup = # of processors * efficiency

\[ S(p) = p \times E(p) \]
Algorithms for Self-tuning

- Method of Golden Sections (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

Figure 1: Two iterations of the method of golden sections: (a) Initial state \( a_0, b_0, c_0, d_0 \); (b) \( S(b_0) > S(c_0) \) \( \Rightarrow \) \( a_1 = a_0, b_1 = d_1 - 0.618(d_1 - a_1), c_1 = b_0, d_1 = c_0 \); (c) \( S(b_1) < S(c_1) \) \( \Rightarrow \) \( a_2 = b_1, b_2 = c_1, c_2 = a_2 + 0.618(d_2 - a_2), d_2 = d_1 \).
Algorithms for Self-tuning

• Assumption
  • Speedup is a single variable function
    • $S(p) : I \rightarrow R$, where the domain is $[1, P]$
    • $S(p)$ can be calculated using equation 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 $[S(P), P]$
  • Start search in this domain
Algorithms for Self-tuning

• 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
Algorithms for Self-tuning

• Validity check for three assumptions
  • For non-unimodal speedup functions, heuristic-based extended MGS search procedure will correctly locate the global maximum
  • Speedup values seen at the beginning of execution are good representations of the job’s behavior in the indefinite future
  • The speedup values of successive iterations are directly comparable
Algorithms for Self-tuning

- **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
Performance

- Self-tuning imposes very little overhead
- Basic self-tuning can significantly improve performance over no-tuning
Performance

- Change-driven self-tuning can significantly improve performance over basic self-tuning
Performance

- 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

• 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 Self-tuning

- 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
• 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