

Maximizing Speedup through Self-Tuning of Processor Allocation

Presenter : Hyuk-Jing Jeong, Inyong Lee

Contents

- Introduction
- Experimental Environments
- Self-Tuning Algorithms
- Multi-Phase Self-Tuning Algorithms
- Conclusion

Introduction

How to maximize the speedup of a parallel Job?

- More processors do not lead to more speedup
 - Optimal processor number for a job (P_{opt}) will be
$$1 \leq P_{opt} \leq P(\text{Max number of processors})$$
- No static processor allocation may be optimal
 - A job's speedup may vary as its execution evolves over time
- Author's Solution : Dynamically determine the number of processors → Self-tuning

Self-Tuning Overview

Self-Tuning iteratively performs 3 steps

1. Dynamically measure **job efficiencies** ($E(p)$) at different processor allocations
2. Calculate **speedup** ($S(p)$)
3. Adjust a job's processor allocation to maximize speedup

Experimental Environment

Experimental environment

- KSR-2 COMA shared memory multiprocessor (supercomputer)
- KSR-2 provides a tool named event monitor which measures runtime information such as,
 - Elapsed execution time
 - Accumulated user-mode execution time
 - Accumulated idle time
 - Accumulated processor stall time
- 10 parallel applications
- Manually instrumented the apps to measure runtime information using the event monitor

Measuring Job Efficiency & Calculating Speedup

- Max job efficiency : 1
- 4 well-known factors that reduce job efficiency
 - ~~Parallelization overhead~~ (typically small)
 - System overhead
 - Idleness
 - Communication (\approx processor stall)

- Measureable factors
- WT(p) : Elapsed execution time
- UT(p) : User-mode execution time
- IT(p) : Idle time
- PST(p) : Processor stall time

$$\text{Efficiency : } E(p) = 1 - \underbrace{\frac{WT(p) - UT(p)}{WT(p)}}_{\text{System overhead}} - \underbrace{\frac{IT(p)}{WT(p)}}_{\text{Idleness}} - \underbrace{\frac{PST(p)}{WT(p)}}_{\text{Processor stall}}$$

(processor p)

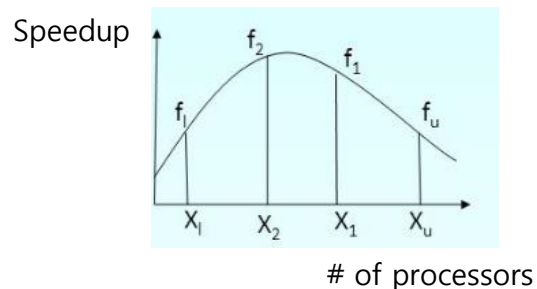
$$\text{Speedup : } S(p) = p \times E(p)$$

(processor p)

Self-Tuning Algorithm

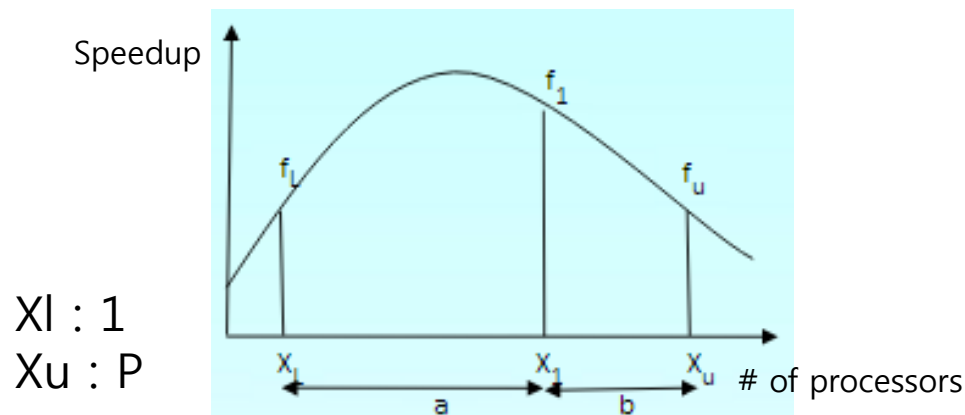
Basic Self-Tuning Algorithm (1)

- Assumption
 - Speedup is a single variable function, $S(p) : I \rightarrow R$ with domain $[1,P]$
 - $S(p)$ can be calculated by measuring $E(p)$ for any one iteration
- Single variable optimization problem
 - Adjustable factor : # of processors
 - Goal : Maximize Speedup
- Observation
 - Most speedup functions are unimodal



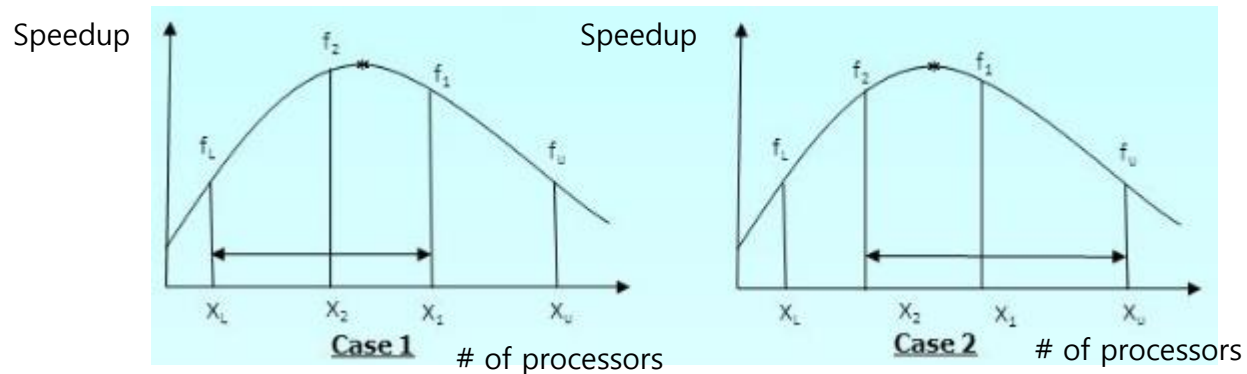
Basic Self-Tuning Algorithm (2)

- MGS (Method of Golden Section)
 - Iterative method to find out **the extremum of a unimodal function**
- At first, we measure $S(1)$, and $S(P)$. (P : maximum # of processors)
- And then, we measure an intermediate point (f_1) which divides the interval $[1 \sim P]$ by golden ratio (0.618)
 - It is known that the ratio will efficiently find out the extremum
- Now we have **3 data** (f_L , f_1 , f_u) and max point will be in $[X_L \sim X_u]$



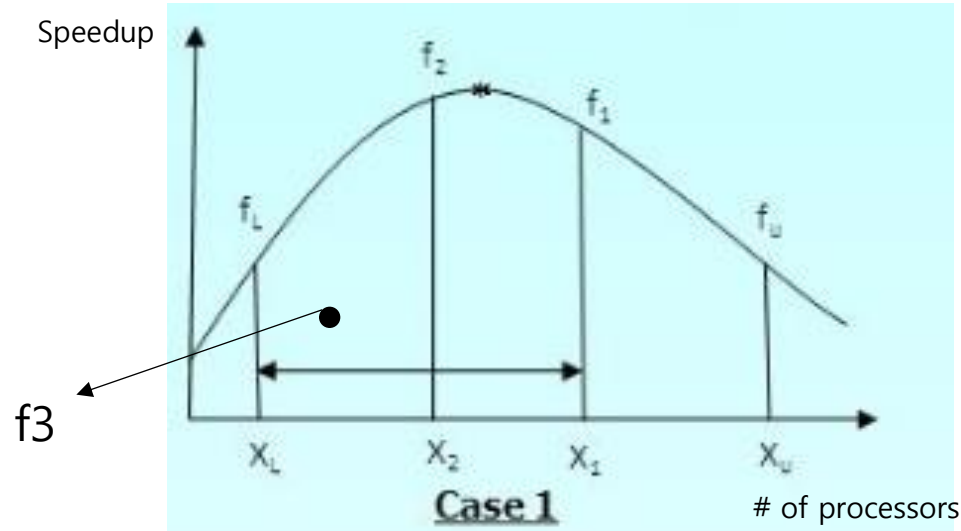
Basic Self-Tuning Algorithm (3)

- Next, we measure f_2 located on the symmetry of f_1
 - Case 1 ($f_2 > f_1$)
 - max point will be in $[X_L \sim X_1]$, point for next iteration : (f_L, f_2, f_1)
 - Case 2 ($f_2 < f_1$)
 - max point will be in $[X_2 \sim X_u]$, point for next iteration : (f_2, f_1, f_u)



Basic Self-Tuning Algorithm (4)

- A heuristic algorithm for non-unimodal case
 - If we encounter a new data (f_3) which indicates the function is **not unimodal**, then we continue with the largest sub-interval conformal with a unimodal function [$f_3 \sim f_1$]



Basic Self-Tuning Algorithm Assumption

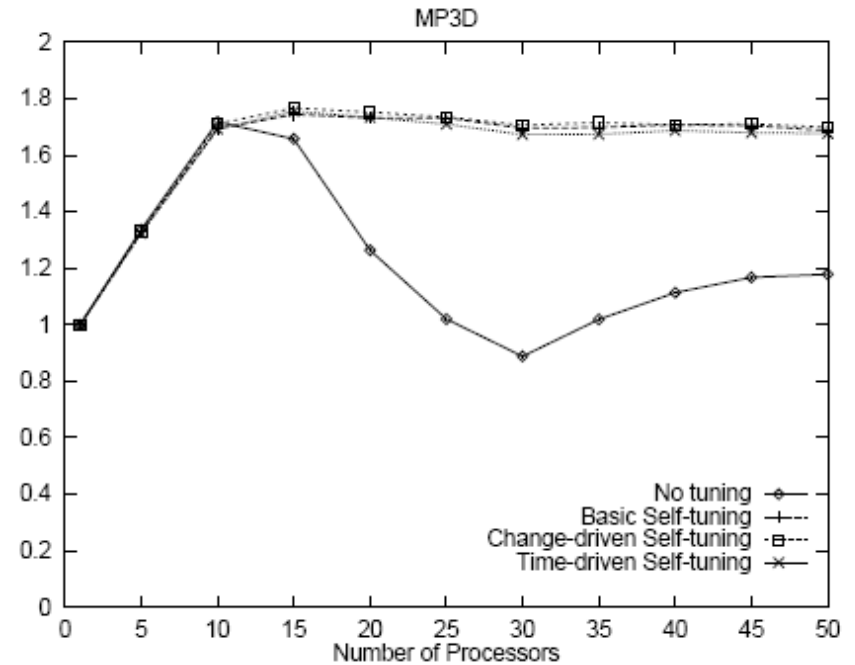
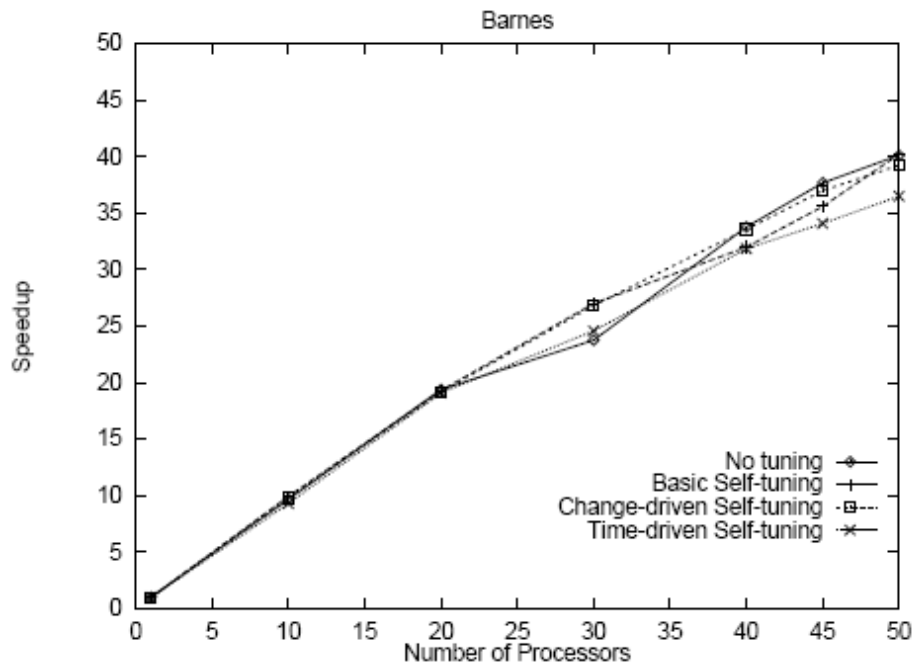
- Non-unimodal speedup function Case
 - Heuristic-based extended MGS search procedure will correctly locate global maximum
- Speedup assumption
 - The basic procedure assumes about job's latter behavior with speedup values at the beginning of execution
 - But, speedup is independent with time. So allocation found may not be so appropriate in later
- Speedup values of successive iterations will vary to some degree

Refined Self-tuning approach

- Change-driven self-tuning
 - Continuously monitors job efficiency and re-initiates the search procedure whenever it notices a **significant change** in efficiency
 - Accords to a predefined threshold value
- Time-driven self-tuning
 - Useful when job efficiency changes in the middle of self-tuning search but stabilizes before the search completes
 - Includes the change-driven approach and **rerun the search procedure periodically**

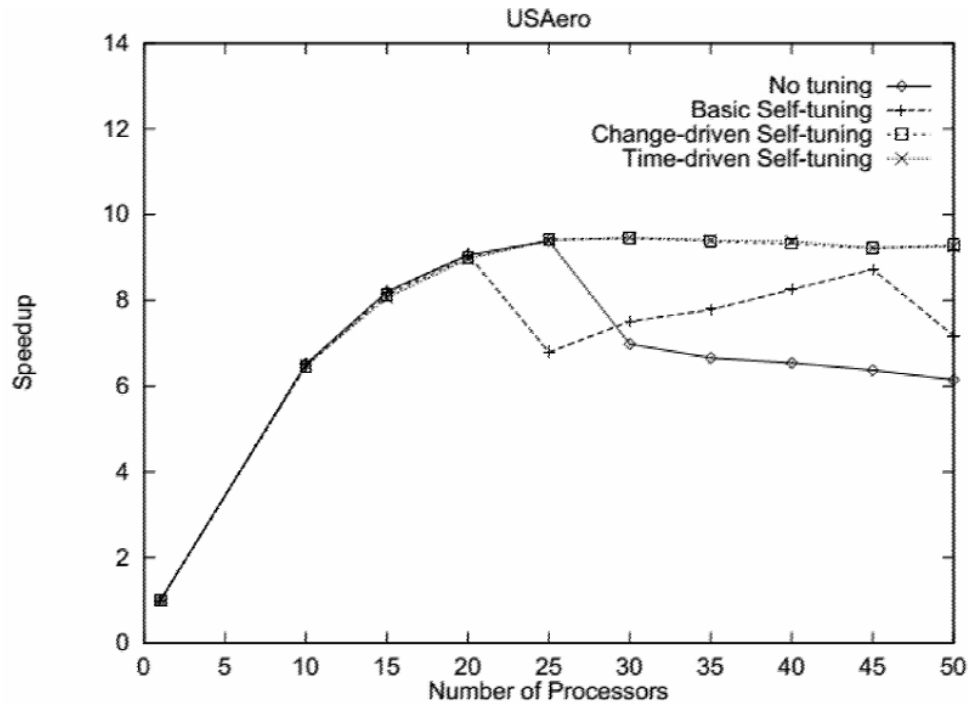
Performance

Performance



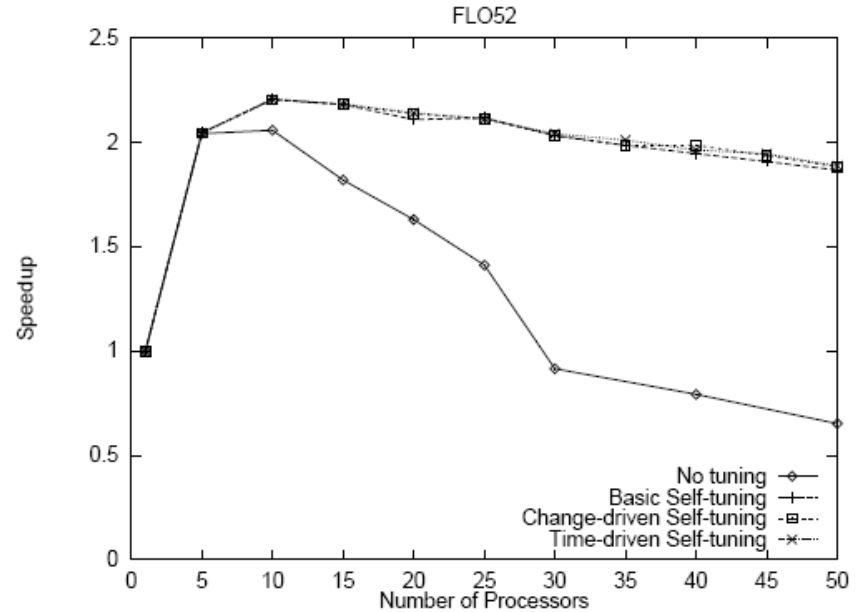
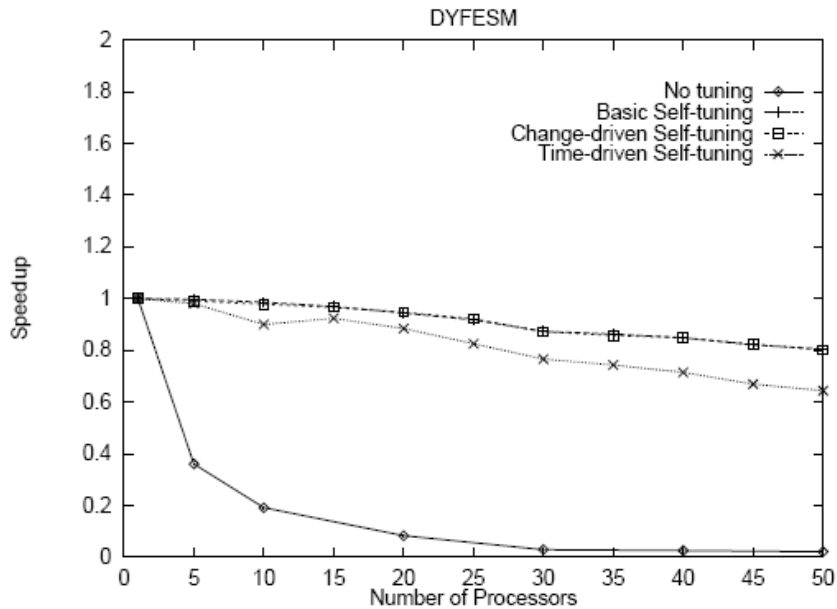
- Self-tuning imposes very little overhead
- Basic self-tuning is significantly better than no-tuning

Performance



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

Performance



- Time-driven self-tuning is not useful for the programs here
- The performance benefit of self-tuning can be limited by the cost of probes

Multi-phase Self-tuning

Problem of refined self-tuning approach

- The iterations of some applications are composed of **multiple** parallel phases
- Phase
 - A specific piece of code
 - e.g. parallel loop in a compiler-parallelized program
 - Each phase may be executed or not depending on the outcome of conditional expressions and sequential loop

Problem of refined self-tuning approach

- Extension of problem
 - Assume that on each try 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, \dots, p_n) that maximizes performance when there are n phases in an iteration

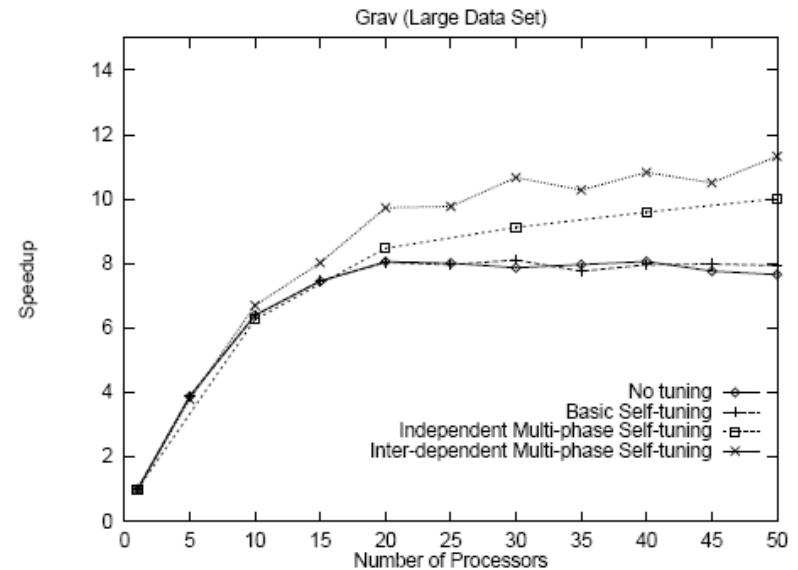
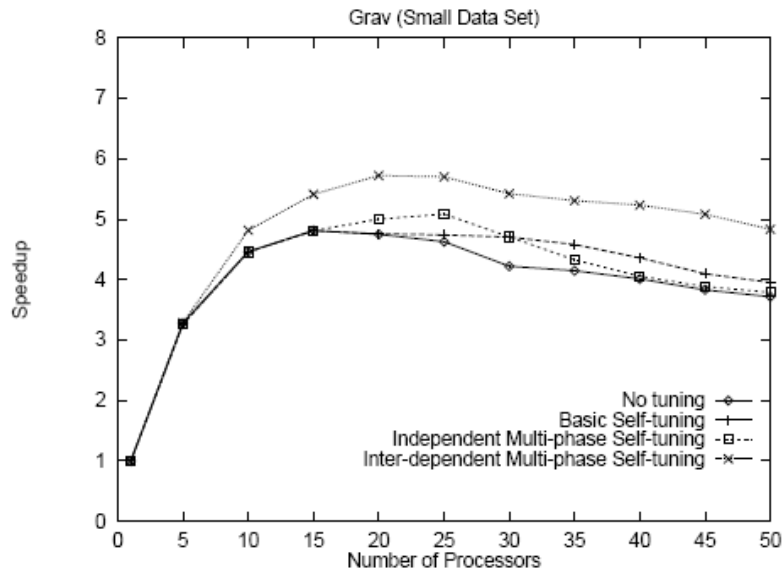
Multi-phase Self-tuning (1)

- Independent multi-phase self-tuning (IMPST)
 - Merely **apply basic self-tuning to each phase** independently
 - Simple
 - Problem : Performance of each phase ALSO depends on the allocations for other phases.

Multi-phase Self-tuning (2)

- 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 (random multiplied)
 - 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 any other

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 from the burden of determining the precise number of processors to use for each input data set
 - Potential to outperform any static allocation