Adaptive, Efficient, Parallel Execution of Parallel Programs

Srinath Sridharan
Gagan Gupta
Gurindar S. Sohi
Efficient Parallel Execution

- Critical for energy and time efficiency
- A challenging problem
  ‣ Dynamic/complex interplay between program and system
- Requires regulating parallelism
  ‣ To match \textit{hardware resource capabilities}
  ‣ Mismatch can lead to under-/over-subscription
Canonical Parallel Machines

- Dedicated parallel server in your own space
- Few programmers
- Had full knowledge of machine resources
- Machine resources stable
- Limited set of programs with known resource demands
- Potentially very large numbers of processors
Canonical Programming Models

- Developed for canonical parallel machines
- E.g., Pthreads, OpenMP, MPI, TBB, Cilk, etc.
- Regulate parallelism **statically**
  - Fix parallelism at development or launch time
- Requires intimate knowledge of the HW/SW configuration of the system
Ubiquitous Parallelism

• Parallel hardware everywhere
• Lots of programmers
• Machine resources heterogeneous and dynamically changing
• Don’t know resources available for program
• Multiple software layers further complicate
• Potentially significant value for more “efficient” parallel execution
Dynamic Heterogeneity

Rapidly changing resource capabilities

Sources:
1. HW faults
2. Cache locality
3. Thermal
4. Power/Energy
5. Multiprogramming
6. Efficiency metrics
Future Scenario

• Program in canonical parallel language
• Run efficiently on ubiquitous parallel hardware
  - Small 10’s of processors/cores
  - Little knowledge of hardware/software layers
• Control one’s own fate for efficiency
  - Don’t rely on OS or other software
Solution Mechanics

Mechanism: control the exposure of application’s parallelism

Need: dynamically and continuously regulate parallelism based upon dynamic resource needs and availability

Desired: completely “under the hood” while using known programming models
Approach: Varuna

• Dynamic, continuous, rapidly and transparent parallelism adaptation

• Optimize for diverse efficiency metrics
  ‣ Execution time and resource consumption

• No application/OS source code changes

• Always outperformed state-of-the-art approaches
How do we realize such a strategy?

1. Detect changes in operating conditions
2. Determine optimum degree of parallelism (DoP) for a performance objective
3. Control program’s parallel execution to match DoP
One Approach: Exhaustive Search

- High overheads
- Operating conditions may change during search
- Can we do better than exhaustive search?
Ideal Approach

- Instantaneously supply optimum degree of parallelism without resorting to search
- Requires oracle view of how dynamic serialization and dynamic acceleration vary with parallelism
Our Approach: Model Side Effects

- Model relationship among dynamic serialization, dynamic acceleration, performance, and DoP
- Gain insights into how side effects vary with parallelism for a given operating condition
- Helps rapidly determine optimum parallelism without performing exhaustive search

How do we model such a relationship?
Our Approach: Model Side Effects

- As a simple **scalability model** based on Amdahl’s law

- Use scalability model to **rapidly estimate** how side effects vary with parallelism

- Use formulae, derived from model, to **instantaneously determine** optimum degree of parallelism
  
  ‣ **MAX** (throughput)

  ‣ **MIN**(resource consumption cost): CPU consumption-execution time product
Modeling Side Effects

Amdahl’s law:

\[ S(P) = \frac{t_p + t_s}{\frac{t_p}{P} + t_s} \]

Assumes parallel region is perfectly parallelizable

Reality: Parallel region can incur side effects and is a function of P

\[ \sigma(P) = P \]

Dynamic serialization

Dynamic acceleration

\[ t_{qc}(P) = t_q(P) - t_c(P) \]
Modeling Side Effects

New parallel region speedup: \( \sigma(P) = \frac{t_p}{\frac{1}{P} + t_{qc}(P)} \)

\( \sigma(P) = \frac{1}{\frac{1}{P} + q_c(P)} \)

\( q_c(P) \) provides insights into how an application’s parallel scalability is influenced by the current operating condition
Example Illustration

\[ \sigma(P) = \frac{1}{\frac{1}{P} + q_c(P)} \]
MAX (Throughput)

\[ \sigma(P) = \frac{1}{\frac{1}{P} + qc(P)} \]

\[ d\frac{1}{\sigma(P)} = -\frac{1}{P^2} + \frac{dq(P)}{dP} = 0 \]

- \( dq(P)/dP \): Rate of change of \( qc(P) \)

- Eqn. applied when \( dq(P)/dP > 0 \), as in case of ReverseIndex

- If —ve or zero, add more resources!
Determining $dq(P)/dP$

- Naive approach:
  - Sweep through all values of $qc(P)$
  - Not effective when conditions change frequently

- Our approach: **Linear regression** on select values of $qc(P)$
  - May lead to errors, but outperforms state-of-the-art
Determining $\frac{dq_c(P)}{dP}$

- Compute $q_c(P)$ values by measuring instantaneous speedups at $P=1$, $P=2$, $P=N/2$, $P=N$

$$q_c(P) = \frac{1}{\sigma(P)} - \frac{1}{P}$$

- Least square method to compute $\frac{dq_c(P)}{dP}$
Measuring Instantaneous Speedup

\[ \sigma(P) = \frac{\text{Perf}(P)}{\text{Perf}(1)} \]

- Instructions per second (IPS) as proxy for performance, Perf

- Assumptions:
  - No spin locks in user level code
  - Do not count OS instructions
MIN (consumption)

\[
\sigma(P) = \frac{1}{\frac{1}{P} + q_c(P)} \quad \Rightarrow \quad \frac{d\sigma(P)}{dP} = P \times \frac{dq_c(P)}{dP} + q_c(P) = 0
\]

\[
P_{opt-c} = -\frac{q_c(P)}{\frac{dq_c(P)}{dP}}
\]
Controlling Parallel Execution

• Ability to pause/resume/migrate/introduce computations

• Task-based programming model possesses necessary foundations to realize this ability
  ‣ Relieves programmers from managing execution
  ‣ Decoupled from HW contexts, can be transparently extended to have their own notion of context
  ‣ Low cost and finer granularity permits programmers to create as many as they want aiding in better load balancing
Controlling Parallel Execution

- Multithreaded programs are not equipped with this capability (e.g., Pthreads)
  - Programmers create and manage parallelism statically

- Consequence: Difficult to control execution without programmer involvement

- Several existing parallel programs are multithreaded
  - Need a way to transparently control their execution
Virtual Tasks (Vtasks)

Abstract HW contexts into Fiber-like entities to which app’s computations are mapped

Vtasks are dynamically mapped on HW contexts

Maintain state of computation using contexts
Virtual Tasks (Vtasks)

- Programming model independent
- E.g., can use threads or tasks
- Varuna transparently maps threads/tasks to vtasks
- Such strategy allows transparent control without programmer involvement
Ensuring Forward Progress

• Vtasks are co-operative tasks
  ‣ Arbitrarily pausing/resuming computations can hamper forward progress

• A vtask’s progress can be affected when it is waiting on a
  ‣ Mutex lock held by a blocked vtask
  ‣ Signal by a blocked vtask (e.g., consumer computation waiting for data from producer computation in a producer-consumer style program)
Handling Blocked Mutexes

• Pause a vtask only when it reaches a safe point
  ‣ Control point which is in user mode and does not hold any user level locks

• User mode status necessary to avoid kernel spin locks
  ‣ One approach: Monitor switches between user and kernel modes
    ‣ Requires changes in OS interfaces or predictive mechanisms based on monitoring privilege instructions [Uhlig, et., al]

• Simpler approach: Synchronization points as safe points
  ‣ Before or after mutex locks, conditional wait or signal, after reaching barrier, etc.
Handling Blocked Mutexes

- **Vtask V1**
  - Acquire lock, L1
  - Critical Section
  - Acquire lock, L2
  - Suspended V1 due to oversubscription

- **Vtask V2**
  - Acquire lock, L1
  - Waiting for V1 to release L1
  - Cannot make forward progress as V1 is suspended while holding L1
Handling Blocked Mutexes

- Each vtask maintains count of mutexes it has acquired
- Not suspend a vtask until the count becomes zero
Handling Blocked Mutexes

• Many multithreaded programs have fewer safe points since they seldom synchronize

• More safe points can be created by spawning more vtasks
  ‣ Task-based programs naturally achieve this

• Observation: Many multithreaded programs take \textit{Num\_threads} as command line argument and create independent work based on that
  ‣ Creating more vtasks as simple as altering command line arg.
Detect Operating Condition Changes

- **Our approach:** Use HW performance counters to periodically monitor a gamut of metrics for changes
  - e.g., Instruction fetch rate, LLC miss rate, CPU utilization, bandwidth utilization, energy consumption, etc.
- Reconfigure parallelism if a change is detected
Varuna’s System Overview

Application

Varuna

Analytical Engine ➔ Parallelism Manager ➔ Analytical Engine

Operating System

Hardware
• Continuously monitor operating conditions

• Determine optimum parallelism to expose for a given performance objective

Analytical Engine

Establish relationship between parallelism, performance and side effects

Determine optimum parallelism for given performance objective

Signal Parallelism Manager

Passively monitor program performance for changes
Parallelism Manager

Parallel computations

Standard parallel APIs (e.g., Pthreads)

Vtask Generator

Vtask pool

Resource Mapper

Worker threads

Input from Analytical Engine
Evaluation

• Three 64-bit X86 machines: 24-context Xeon, 16-context Opteron, and 8-context Core-i7

• 24 different Pthreads, TBB, and Prometheus benchmarks
  › No source code modifications!

• Two different environments
  › Isolated and Multiprogrammed

• Evaluated for execution time, energy, and resource consumption cost
Evaluation

• Benefits of applying vtasks to unmodified multithreaded and task programs

• Benefits of applying adaptive optimization

• Effectiveness in dynamically changing conditions

• Agility in responding to changes
## Vtask Count on Xeon

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Pthreads</th>
<th>Varuna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barneshut</td>
<td>24</td>
<td>100,000</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>24</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Bzip2</td>
<td>26</td>
<td>24</td>
</tr>
<tr>
<td>Canneal</td>
<td>24</td>
<td>96</td>
</tr>
<tr>
<td>Dedup</td>
<td>74</td>
<td>24</td>
</tr>
<tr>
<td>Fluidanimate</td>
<td>24</td>
<td>384</td>
</tr>
<tr>
<td>Hash Join</td>
<td>24</td>
<td>1024</td>
</tr>
<tr>
<td>Histogram</td>
<td>24</td>
<td>1024</td>
</tr>
<tr>
<td>RE</td>
<td>24</td>
<td>100,000</td>
</tr>
<tr>
<td>ReverseIndex</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Stream</td>
<td>24</td>
<td>1024</td>
</tr>
<tr>
<td>Swaptions</td>
<td>24</td>
<td>384</td>
</tr>
<tr>
<td>Word Count</td>
<td>24</td>
<td>256</td>
</tr>
<tr>
<td>X264</td>
<td>26</td>
<td>24</td>
</tr>
</tbody>
</table>
Isolated: Pthreads Execution Time on Xeon

Lazy vtask creation [Mohr, et., al]
Isolated: MAX (throughput)

12% average reduction in exec. time
22% average reduction in energy consumption
Isolated: MIN(consumption)

Picked optimum parallelism of one

12% degradation compared to V_PT_T
Isolated: $\text{MIN(consumption)}$

80% reduction in resource consumption cost

![Bar chart showing relative consumption cost for various benchmarks.](chart.png)
Multiprogrammed: TBB Execution Time on Xeon

- Avg. 33% reduction in execution time
- 30% and 20% better than FDT and Parcae
Varuna’s adaptation strategy
Conclusion

• Future multicores will exhibit dynamically operating conditions

• Varuna helps to adapt a program’s parallelism

• Quick, efficient and requires no source code changes!

• Improves time-efficiency, energy-efficiency and resource consumption cost by 36%, 32%, and 80% respectively.
## Varuna vs Others

<table>
<thead>
<tr>
<th>Proposal</th>
<th>Continuous adaptation</th>
<th>Principled approach</th>
<th>No source code changes</th>
<th>Resource agnostic</th>
<th>Speed</th>
<th>Diverse metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parcae/DoPE</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Slow</td>
<td>No/Yes</td>
</tr>
<tr>
<td>FDT</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Fast</td>
<td>No</td>
</tr>
<tr>
<td>Lithe</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Slow</td>
<td>No</td>
</tr>
<tr>
<td>Varuna</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Rapid</td>
<td>Yes</td>
</tr>
</tbody>
</table>
• Rapid & accurate parallelism determination using above observations

• Will show for one efficiency metric: Min (Execution time)

• Min (consumption) in paper. Many more possible!
Periodic Diversification

• Model assumes unimodal distribution
  › Can get stuck in a local optimum

• Solution: Periodically diversify search
  › Inspired from Tabu search

• We randomly alter points 2 and 3
Determining $dq_c(P)/dP$

- Naive approach:
  - Sweep through all values of $q_c(P)$
  - Not effective when conditions change frequently

- Our approach: **Linear regression** on select values of $q_c(P)$
  - May lead to errors, but outperforms state-of-the-art
**Determining dqc(P)/dP**

- Compute qc(P) by measuring instantaneous speedups at $P = 1$, $P = 2$, $P = N/2$, $P = N$

$$qc(P) = \frac{1}{\sigma(P)} - \frac{1}{P}$$

- **Least square method** to compute $dqc(P)/dP$
Limitations of the Model

- Model assumes monotonic distribution of $qc(P)$
  - May arrive at non-optimum solution if violated
- Have not experimentally seen such scenarios
- Solution: Employ a hill climbing search
  - Incorporates Tabu search to overcome local optima problem
  - Currently, user-driven
- Future work: Hybrid approach