B. Tudor and Y.M. Teo, A Practical Approach for Performance Analysis of Shared Memory Programs, Proceedings of 25th IEEE International Parallel & Distributed Processing Symposium, Anchorage, USA, May 16-20, 2011.

# A Practical Approach for Performance Analysis of Shared-Memory Programs

Bogdan Marius Tudor and Yong Meng Teo
Department of Computer Science
National University of Singapore
Computing 1, 13 Computing Drive,
Singapore 117417
[bogdanma,teoym]@comp.nus.edu.sg

***Abstract*—Parallel programming has transcended from HPC into mainstream, enabled by a growing number of programming models, languages and methodologies, as well as the availability of multicore systems. However, performance analysis of parallel programs is still difficult, especially for large and complex programs, or applications developed using different programming models. This paper proposes a simple analytical model for studying the speedup of shared-memory programs on multicore systems. The proposed model derives the speedup and speedup loss from data dependency and memory overhead for various configurations of threads, cores and memory access policies in UMA and NUMA systems. The model is practical because it uses only generally available and non-intrusive inputs derived from the trace of the operating system run-queue and hardware events counters. Using six OpenMP HPC dwarfs from the NPB benchmark, our model differs from measurement results on average by 9% for UMA and 11% on NUMA. Our analysis shows that speedup loss is dominated by memory contention, especially for larger problem sizes. For the worst performing structured grid dwarf on UMA, memory contention accounts for up to 99% of the speedup loss. Based on this insight, we apply our model to determine the optimal number of cores that alleviates memory contention, maximizing speedup and reducing execution time.**

***Keywords*-analytical model, speedup performance, speedup loss, data dependency, memory contention.**

## I. Introduction

With the advent of multicore systems, parallel computing is becoming mainstream. To accommodate the hardware changes, a large number of programming languages (e.g. Cilk, Fortress [1]), models and methodologies (e.g. OpenMP, Intel Thread Building Blocks and pthreads library for shared-memory programs, UPC [2] and Co-array Fortran [3] for distributed shared-memory) have been proposed. Together with the growing importance of programming models, performance analysis of parallel programs is important for several reasons. Firstly, at the design stage, performance analysis allows the parallel program developer to quantify the parallelism of applications and its execution speedup. Secondly, before procuring parallel machines, performance evaluation enables the users to estimate the program runtime performance on different machine configurations. Thirdly, non-intrusive performance analysis methods can be applied concurrently to the execution program to auto-tune its performance. However, the large number of parallel programming options such as programming models, languages, problem size and memory systems leads to significant challenges in understanding the performance loss associated with each choice.

Current performance analysis approaches can be compared based on three key design trade-offs: accuracy, intrusiveness, and ease of use. Traditional methods for performance analysis include software instrumentation methods and trace-driven analysis [4], [5], [6]. However, while they have good accuracy, these approaches are often intrusive. This leads to difficulty in generalizing them across programming languages and models. Recently, a shift in analysis methods recognize that the performance of large parallel programs depends on a multidimentional space of options and configuration parameters, including programming models, number of threads and processor cores, problem size, memory architecture, thread-to-core placement among others. Therefore, the ease of applying the model across this parameter space is becoming a crucial design criteria for performance analysis methods [7], [8]. Methods that rely on empirical data, such as regression based-approaches, neural networks and machine learning [7], [9], [10], [11] typically produce good accuracy but they require significant modeling effort or large volume of training data. On the other hand, analytical models [12], [13], [14], [15] are easy to apply, but often the simplifying assumptions reduce accuracy below what is useful for practical purposes.

In this paper, we propose a practical model for characterizing the parallel performance of shared-memory programs. The model divides the lifetime of a program into *useful work*, overhead due to *memory contention* and inactivity induced by *data dependency*. Our analytical model determines the speedup of a program across a range of configuration parameters such as number of threads, number of cores and memory access policies on both Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA) systems. Speedup loss due to *data dependency* is determined using a trace of the operating system run-queue, and the speedup loss resulting from memory contention is based on modeling the number of processor cycles used by the program. Our approach is practical because it does not incur instrumentation overhead. Moreover, it is independent of the underlying programming languages and models, and does not require source or binary code inspection. Validation of our model against measurements using six

OpenMP HPC dwarfs from the NPB [16] benchmark shows that our model results differ by 9-11% from measurement on both UMA and NUMA memory systems.

The contributions of this paper are twofold. First, we propose a simple and practical analytical performance model for evaluating the speedup of shared-memory programs. Key to our approach is the modeling of speedup loss due to data-dependency and memory contentions. Furthermore, we model the memory scalability in both UMA and NUMA systems. Second, we applying our model to study the performance impact of the memory wall. We observe that memory contention significantly dominates in large programs, and accounts for up to 99% of speedup loss. Based on this insight, we extend our model to determine the optimal number of cores that maximizes speedup.

This paper is organized as follows. Section II introduces our proposed speedup model by presenting definitions, assumptions, modeling details as well as limitations of our approach. In section III, we present our validation results against measurements on six HPC dwarfs from NPB OpenMP suite, and apply the model to study the impact of the memory wall on speedup. We contrast our approach with related work in section IV. Section V concludes the paper and discusses future work.

## II. Speedup Model

In this section, we present an overview of our proposed speedup model and discuss in detail the modeling of speedup loss due to data dependency and memory contention.

### A. Model Overview and Definitions

The objective of our model is to derive the parallel speedup and to express the speedup loss due to (i) data dependency in the program and (ii) resource contention among the processor cores.

A shared-memory program that exploits the parallelism of multicore systems is partitioned into a number of threads. The number of threads is determined either at program creation time or as a runtime parameter. Once the partitioning is set, the lifetime of a program consists of periods of time when threads are active executing computation and issuing/waiting for memory requests, or inactive because they are stopped at synchronization points or due to insufficient work.

When a program is partitioned into $m$ threads, only a subset are active at any given time, due to data dependencies. Let $A(m,n,t)$ denote the number of active threads running on $n$ cores at time $t$, and $A(m,n)$ denote the number of active threads, averaged over the entire execution time of the program, $T$. Thus,

$$A(m,n) = \frac{1}{T}\int_0^T A(m,n,t)dt$$

When there are enough cores to execute all threads concurrently, $n \geq m$, the maximum number of active threads in the program is reached, which is defined as $A(m,\infty)$. Thus, $A(m,\infty) = A(m, n \geq m)$. However, if the number of cores $n < m$, then the number of active threads is constrained by $n$.

At any time moment the active threads are in one of the following states: executing useful work or stalled due to memory contention. In general, a significant runtime overhead in multicore systems is caused by contention among shared resources. State-of-the-art processor cores are deeply-pipelined and superscalar, and thus are able to execute multiple integer and floating point instructions, and issue multiple memory requests in the same cycle. However, they share the caches and memory controllers. Therefore, when multiple cores issue memory requests at the same time, there is a contention among requests from different cores, leading to a difference in the performance of each core, compared with the uncontented case. Cores appear active even when they suffer from memory contention, but instead of performing useful work, processor cores are stalled waiting for the operands to be fetched from memory. We acknowledge that there are many sources of runtime overhead, but in this paper we focus on the runtime overhead due to memory contention.

The total number of cycles spent by a multithreaded program can be divided into *stall cycles* (cycles when no instruction is completed) and *work cycles* (cycles when at least one instruction is completed). Even without memory contention among the cores, the number of stall cycles is not zero, since there are branch mispredictions, pipeline stalls within cores, latencies of cache and memory and even contention among the memory request issued by the same core. We consider therefore that not all stall cycles are parallel overhead, but only the stall cycles caused by contention among different cores.

Let $U$ denote the useful work of the program, expressed as time units, and let $M(n)$ be the overhead caused by memory contention among $n$ cores, expressed as time units. Naturally, $M(1) = 0$, because there is no contention among cores when executing on a single core. The total amount of time required to execute the program on one core, $T(1)$, is:

$$T(1) = U$$

On $n$ cores, if there are $m$ threads out of which $A(m,n)$ are active, assuming that the useful work of the program does not change with $n$, then the execution time $T(n)$ is:

$$T(n) = \frac{U + M(n)}{A(m,n)}$$

The parallel speedup is therefore:

$$S(m,n) = \frac{T(1)}{T(n)} = A(m,n)\frac{U}{U + M(n)}$$

Let $\omega(n) = \frac{M(n)}{U}$, the ratio of average number of active threads due to memory contention to the average number of active threads due to useful work. Thus, the speedup performance for a shared-memory program with $m$ threads on $n$ cores as:

$$S(m,n) = \frac{A(m,n)}{1 + \omega(n)} \tag{1}$$

Equation 1 exposes a useful insight on the parallelism of a shared-memory program. For a program partitioned into $m$

threads and executed on $n$ cores, $A(m,n)$ measures the total amount of parallelized work, and the denominator expresses how much of that work is useful work. Furthermore, it follows from the definition of $\omega(n)$ that the average number of threads performing useful work is $\frac{A(m,n)}{1+\omega(n)}$.

Eager et. al [13] have derived a expression similar to equation 1, using the average program parallelism and the speedup, but without considering runtime overhead. In general, our speedup model is consistent with many studies on inherent parallelism [13], [14], [17], [18]. However, these studies relate the speedup only to the average parallelism of the program and express the performance loss due to constraining the number of cores. These simplified models, while useful, are impractical for understanding current multicore systems, where memory contention can have a significant impact on speedup performance. Another limitation of existing studies based on inherent parallelism is that they do not provide a method of deriving it. Simply measuring the active number of threads is not sufficient, since some of the threads may be active but stalled due to memory contention. In contrast, our model defines the speedup loss due to data dependency and memory contention. Furthermore, we provide a practical approach of deriving $A(m,n)$ and $\omega(n)$.

**Definition 1 (Speedup Loss Due to Data Dependency).** *Speedup loss due to data dependency in the program, $D(m)$, is defined as the difference between available threads, $m$, and average number of active threads given no constrain on the number of cores, $A(m,\infty)$:*

$$D(m) = m - A(m,\infty) \tag{2}$$

**Definition 2 (Speedup Loss Due to Memory Contention).** *Speedup loss due to memory contention, $R(m,n)$ is defined as the difference of the number of active threads and the number of threads doing useful work*:

$$R(m,n) = A(m,n) - \frac{A(m,n)}{1+\omega(n)} = A(m,n)\frac{\omega(n)}{1+\omega(n)} \tag{3}$$

### B. Speedup Loss Model

In this section we present a model to determine $A(m,n)$ and $\omega(n)$. For a shared-memory program partitioned into $m$ threads, we can quantify $S(m,n)$, $A(m,n)$, $D(m)$ and $\omega(n)$, for any value of $n$, using a baseline run on $b$ cores, where $m > b$, and a hierarchical model of a multicore machine based on the number of processor cycles required to execute the program.

*Data Dependency*

The data dependency of the program is the average number of threads which are inactive throughout the execution time of the program. The reasons why some threads may not be active include synchronization operations, load imbalances among threads and insufficient work to keep all threads busy. In general, we treat all sources of thread inactivity as data dependency. We acknowledge that there are reasons unrelated to the parallelism that may cause threads to become inactive, such as system calls. As we target applications with heavy memory contention and data dependency, we focus on parallel programs where system calls do not represent a significant source of performance loss.

We derive a model for the data dependency of the program that determines the average number of active threads of a program, $A(m,n)$, based on an execution of the program partitioned into $m$ threads on $b$ cores, where $m > b$. Based on $A(m,n)$, we then determine $D(m)$.

The insight behind our approach is that when the number of threads is greater than the number of cores, the sum of the number of executing threads and the number of threads in the run-queue represents the number of active threads of the program. Our key idea is to conduct one measurement run, called a *baseline run* executing the program on a number of processor cores smaller than the number of threads. Since there are more threads than cores, some of the threads will queue for service in the run-queue. Based on the profile of number of active threads over time and on the service time of the threads, we infer the time required to execute the threads when there are enough cores to execute all the threads concurrently, $n \geq m$, without considering the memory overhead. We then determine $A(m,n)$ as time weighted average.

However, oversubscribing the cores has three effects on performance [19]:

(i) The total execution time of the program is different compared to when there are enough cores for all the threads, because there are not enough cores to execute all threads concurrently.
(ii) It may cause load imbalances between threads.
(iii) It may cause significant context switching which may increase the kernel service time of the program as well as affect the efficiency of the caching.

Our model accounts for the first two effects, and we give an experimental analysis of the third.

We run the program partitioned into $m$ threads on $b$ cores, where $m > b$. We determine the time required to finish the program, given enough cores to execute all threads concurrently, $m \leq n$, which we denote as critical path time $T_{cp}$. The average number of active threads is then determined as a time weighted average of the parallelism of each region of program.

When $m > n$, some threads may have to queue for service in the run-queue. Therefore, at time moment $t$, $x(m,t)$ denotes the number of threads that are executing and $q(m,t)$ denotes the number of threads that are queueing. Given enough cores to run all the threads in parallel, $m \leq n$, then the number of active threads is:

$$A(m,\infty,t) = x(m,t) + q(m,t)$$

Based on the time required to execute the baseline run, the model determines the critical path time, $T_{cp}$.

During the baseline run, let $\tau$ denote the service time received by the program from time $t$ to time $t+\Delta T$. If $\Delta T$ is very small, there is no change in the number of active threads from $t$ to $t+\Delta T$, and we have:

$$\tau = x(m,t) \cdot \Delta T = \sum_{j=1}^{m} \tau_j$$

where $\tau_j$ is the service time required by thread $j$. The execution time when $m \leq n$, is $\Delta T_{cp}$, and is equivalent to the maximum service time received by one thread:

$$\Delta T_{cp} = max\{\tau_j\}$$

The average number of active threads during this interval is:

$$A(m,\infty,t) = \frac{\tau}{\Delta T_{cp}} = \frac{\sum_{j=1}^{m} \tau_j}{max\{\tau_j\}}$$

When $m \leq n$ cores, the total execution time of the program, $T_{cp}$, is the sum of the minimum time to execute every part of the program $T_{cp} = \sum \Delta T_{cp}$. Therefore, the average number of active threads over the entire execution of the program is the average of the values of $A(m,\infty,t)$ with weights $\Delta T_{cp}$:

$$A(m,\infty) = \frac{\sum A(m,\infty,t)\Delta T_{cp}}{T_{cp}} \tag{4}$$

and

$$D(m) = m - \frac{\sum A(m,\infty,t)\Delta T_{cp}}{T_{cp}} \tag{5}$$

It can be argued that since $\tau = x(m,t) \cdot \Delta T$, we do not need to measure $\tau_j$. However, simply computing $\Delta T_{cp}$ as an average, $\Delta T_{cp} = \frac{x(m,t) \cdot \Delta T}{x(m,t)+q(m,t)}$, would not account for the load imbalances between threads, which may lead to underestimating the data dependency of the program.

Next, we show the derivation of the average number of active threads when the program is executing $m$ threads on an arbitrary $n$ cores. We start from the profile of the active number of threads $A(m,\infty,t)$ over time. If during interval $\Delta T$ there are $A(m,\infty,t)$ active threads, then on $n$ cores the number of executing threads is $min\{n, A(m,\infty,t)\}$. Therefore, the time required to execute them on $n$ cores, $\Delta T(n)$, is:

$$\Delta T(n) = \frac{\Delta T_{cp} \cdot A(m,\infty,t)}{min\{n, A(m,\infty,t)\}}$$

The average number of active threads, $A(m,n)$ is determined as the average number of active threads, $min\{n, A(m,\infty,t)\}$, weighted to $\Delta T(n)$ for part of the program, similarly to equation 4:

$$A(m,n) = \frac{\sum A(m,n,t)\Delta T(n)}{\sum \Delta T(n)} \tag{6}$$

We implemented the model using sampling. The size of the run-queue is sampled using a constant $\Delta T$ time interval. For every sample, we measure $A(m,\infty,t)$ and the vector of service time of the threads, $\tau_j$.

*Memory Contention in Multicore Systems*

We propose a model to derive the parallelism loss due to memory contention, $\omega(n)$ for shared-memory programs.

Our model is centered on the memory contention among different cores. Unlike many existing studies [20], [21], we are not interested in the absolute value of stall cycles, but in the growth of the stall cycles due to memory contention among cores. We therefore are interested in the growth of stall cycles relative to a baseline value on one core, where there is no memory contention among cores.

We model multi-socket systems with both UMA and NUMA memory access. In UMA, each socket has a dedicated bus to the single memory controller, while in NUMA each socket has its own bus to the local memory controller, as well as a connection to any other sockets. The sockets and the cores are considered identical. This simple model covers previous generation and state-of-the-art multicore systems, based on Intel Core microarchitecture (UMA), and Intel Nehalem and AMD K10 (NUMA).

Let $C(n)$ denote the total number of cycles spent by $n$ cores on behalf of a shared-memory program. We divide the cycles into three categories:

(i) Work cycles: $W(n)$;
(ii) Stall cycles that are not due to resource contention, such as pipeline hazards, branch mispredictions, cache hits and uncontented memory accesses: $B(n)$;
(iii) Stall cycles due contention for shared resources, such as accesses to caches and memory controllers by multiple cores: $M(n)$.

$$C(n) = W(n) + B(n) + M(n)$$

On a single core, the contention for shared resource among cores is zero, therefore $M(1) = 0$ and $C(1) = B(1) + W(1)$ can be considered the useful work part of the program. Although $B(1)$ are stall cycles, we consider them a component of the useful work, i.e. the number of stall cycles required to fetch the data when there is no contention among cores. Thus:

$$\omega(n) = \frac{M(n)}{U} = \frac{M(n)}{W(n) + B(n)}$$

From measurement analysis we observed that $W(n)$ and $B(n)$ are constant with $n$, when the number of cache misses and the total number of instructions do not change with $n$. The intuitive explanation for this behavior is that, since $B(n)$ represents the stalls due to uncontented resources, it does not matter how many cores split these stalls, because their total remains the same. Similarly, the execution time of floating point and integer instructions depends only on the availability of the operands. If caching does not change when $n$ changes, then operand availability does not change, and neither does the number of cycles required to execute them. Moreover, we are interested in modeling the number of total cycles for large program runs, with long steady-state balance of shared caches, therefore small, transient deviations from this assumption do not make the objective of this study. Furthermore, we have evaluated this assumption empirically on all our workload tests and present evidence to support it.

Based on this observation, the modeling of cache contention can be simplified:

$$\omega(n) = \frac{M(n)}{W + B} = \frac{C(n)}{C(1)} - 1 \tag{7}$$

The total number of cycles, $C(n)$ is modeled using a hierarchical approach, as follows. We derive first an equation for $C(n)$ for one socket and subsequently we model the effect of interconnecting multiple sockets.

The system has multiple socket, and each socket has $c$ cores. The cores have inclusive caches, therefore only last level misses are sent to the memory controllers. For both UMA and NUMA, we consider one memory controller to be a single server system, in which requests are serviced in first-come-first-serve order. Requests arriving concurrently from different cores queue for bus access. We apply a $M/M/1$ model to derive the response time of the memory requests. Since the memory requests start from the cores but are filtered by two or three levels of cache, the inter-arrival times of the request are assumed independent and identically distributed. Considering the superscalar and out-of-order nature of modern processors, cores issue memory requests and flops/integer operations instructions at the same time. This means that while cores are waiting for memory requests to be completed, they are also executing instructions for which the operands have been fetched from memory. Therefore, for programs with significant memory contention, *the critical path of the execution time of a program is dominated by the response time of memory requests*. Let $C_{req}(n)$ be the response time (expressed in number of CPU cycles) of one memory request that has arrived at a memory controller which services $n$ cores. From the $M/M/1$ model [22]:

$$C_{req}(n) = \frac{1}{\mu - \lambda}$$

where $\mu$ is the service rate of the memory controller and $\lambda$ is the arrival rate of the memory requests. Let $r(n)$ denote the total number of last level cache misses, and $L$ the arrival rate of requests from one core.

For a single socket system, with $n$ cores active, $\lambda = n \cdot L$:

$$C(n) = r(n) C_{req}(n) = \frac{r(n)}{\mu - n \cdot L} \tag{8}$$

Next we extend the model to multiple sockets. In UMA, each socket has its own bus, and therefore requests from different sockets queue for memory access separately. Therefore, queueing time is modeled separate for each socket. In a two socket system, if $n_1$ cores are active in the first socket and $n_2$ in the second, $C_{UMA}(n) = C(n_1) + C(n_2) + \Delta C$, where $\Delta C$ represents the increase in number of cycles due to the increase in load on the controller, which services requests from two sockets, instead of one. If $n$ changes using a fill socket first policy, when changing from $c$ cores (all on the first socket) to $c+1$ cores ($c$ on the first socket and 1 on the second socket), the difference between $\Delta C = C(c+1) - C(c)$ reflects the increase in response time from increasing the load on the memory controller:

$$C_{UMA}(n) = C(c) + C(n-c) + \Delta C \tag{9}$$

For NUMA, when two sockets and two memory nodes are active, there is an additional delay to send the memory request to a remote node. Let $\delta(n)$ be the additional time required to send the memory requests to a remote memory controller, compared to the case when only the local controller is active. The total number of cycles is therefore:

$$C(n) = r(n) \cdot C_{req}(n) + r(n) \cdot \delta(n)$$

and $\delta(n)$ depends on the ratio of remote memory accesses to total memory accesses. If $n$ cores are split as $c$ on the first socket and $n-c$ cores on the second, on average the memory accesses will be split $\frac{c}{n}$ on the first memory controller and $\frac{n-c}{n}$ on the second memory controller. Compared to the case where all memory requests go to the local controller, the number of cycles increases proportionally to the number of requests that go to the remote controller. Therefore $\delta(n) = \delta \cdot (n-c)$ and

$$C_{NUMA}(n) = C(c) + r(n) \cdot \delta \cdot (n-c) \tag{10}$$

In equations 8, parameters $L$ and $\mu$ implicitly model the effect of memory request and memory performance on the number of cycles. Similarly, in equations 9 and 10, the parameters $\Delta C$ and $\delta$ account for the increase in cycles when activating additional sockets. A detailed model of these parameters is beyond the objective of this paper, because from a practical point of view, they can be extracted using linear regression from a set of measured values of $C(n)$. Since we have observed that $r(n)$ is constant, parameters $L$ and $\mu$ can be linearly regressed using equation 8 and at least two points of $\frac{1}{C(n)}$. Similarly, $\Delta C$ and $\delta$ can be determined using measured values of $C(c)$ and $C(c+1)$. Therefore, to apply the memory model, we need the following set of input parameters:

(i) For a single socket system, two runs of the program on $n_1$ and $n_2$ cores and measurements of $C(n_1)$ and $C(n_2)$ are required. Parameters $L$ and $\mu$ are regressed through the coordinates $\{n_1, 1/C(n_1)\}$ and $\{n_2, 1/C(n_2)\}$.

(ii) For a multiple socket system, is required:
- On UMA, measurements on 1, $c$ and $c+1$ cores. We measure $\Delta C = C(c+1) - C(c)$, in addition to regression of $L$ and $\mu$.
- On NUMA, measurements on 1, $c$ and $c+1$ cores. Parameter $\delta$ is regressed from the line $\{c, C(c)\}$ to $\{c+1, C(c+1)\}$, in addition to $L$ and $\mu$.

Therefore, $\omega(n)$ can be determined from at most three measurements of $C(n)$ via equation 7.

*Limitations.* Our model predicts the average number of active threads for programs with a parallelism profile independent of execution constraints, such as the ones using OpenMP 2.5. When a program is composed of programming language tasks, such as OpenMP 3.0 tasks or Cilk tasks, the degree of parallelism of the program may not be fixed and might be dictated by a run-time scheduler. For such programs, when partitioned into $m$ threads, subjecting them to a execution on $n < m$ cores might result in a different active threads profile compared to an execution on $n = m$ cores. This is because by solving a task at a particular moment in time, the program might "unlock" several other tasks which would affect the parallelism of future regions. The current model can be extended to account for the performance of the run-time task scheduler to analyze the performance of such programs. Secondly, if thread synchronization is done through busy waiting for significant periods of time, our model might underestimate the data-dependency of the program.

## III. Evaluation

In this section, we discuss the validation of our model against measurements, and the use of our model in parallelism analysis. We study the performance six HPC dwarfs from the NPB 3.3 benchmark [16] (Table I). The NPB dwarfs are

| Name | Parallel kernel |
|---|---|
| BT | Dense linear algebra: use matrices/vectors to store data |
| EP | Embarrassingly parallel: low data dependency, low memory |
| FT | Spectral methods: fast Fourier transform |
| IS | Parallel sorting: bucket sort on integers |
| CG | Sparse linear algebra: data with many 0 values |
| SP | Structured grid: pentadiagonal solver |

TABLE I
Six OpenMP HPC dwarfs from NPB 3.3

written in OpenMP 2.5 and represent parallel kernels of widely used scientific and high performance computing applications. The six programs differ in terms of achieved speedup, data dependency and memory requirements. EP is highly parallel while BT, CG and SP have significant data dependency, and FT and SP have high memory requirements. The programs are compiled using `gcc` with full optimizations (`-O3`) and relaxed floating points options (`-ffast-math`). To exclude the effect of memory thrashing, we selected the largest problem size that fits in our system memory size, i.e., class C for all dwarfs and class B for FT. Our measurements are conducted on two systems with different memory architecture:

(i) **UMA**: Dual socket Intel E5320 (Clovertown), 1.87 GHz, 4 cores and 2x4 MB L2 cache per socket, 4 GB RAM DDR2, Linux 2.6.22,
(ii) **NUMA**: Dual socket Intel E5520 (Gainestown), 2.27 GHz, 4 cores with 8 hardware threads with 4x256 kB L2 and 8 MB L3 cache per socket, 24 GB RAM DDR3, Linux 2.6.31

For the NUMA system, we treated the hardware threads provided by the system as processor cores, and therefore the NUMA system has 16 cores. The trace of the operating system run-queue was obtained using a C program that samples the `procfs` entries to log the number of runnable threads and their user-level CPU service time. We used `time` system utility to measure the wall clock time, `sched_setaffinity` system call to restrict the number of cores allocated to a program, `numactl` utility to specify the memory access nodes, and PAPI 3.7.0 (UMA) / PAPI 4.1.0 (NUMA) for hardware counters access. Unless otherwise stated, we run each experiment five times, and for the cases where we found significant differences among the runs, we present the relative difference among the runs.

### A. Model Validation

We discuss our model validation using measurements. Firstly, we determine the baseline run configuration. We discuss the sensitivity of the average number of threads prediction to the run-queue sampling interval and to the number of cores on which we perform the baseline run. Secondly, we validate the speedup prediction for all six dwarfs, for different configurations of threads and cores. Lastly, we validate the memory contention against measurements for UMA and NUMA systems. As our analytical model results are close to measurements, we present summary results for all dwarfs, and show in detail the validation for program BT, since it has the lowest model accuracy.

*1) Baseline Run and Sensitivity to Sampling Interval:* The choice of run-queue sample interval is crucial in determining the correct parallelism profile, and we consider two opposing aspects:

(i) If the sample interval is large, there is a higher probability of not detecting parallelism changes within the interval. The sample interval must thus be lower than the time between two consecutive changes in the value of $A(m, \infty, t)$.
(ii) It the sample interval is too small, the service time of the threads cannot be accurately measured, leading to incorrect compensation for load imbalances.

We perform a quantitative analysis to determine the optimal *run-queue sample intervals* and *number of cores* for the baseline run. Fig. 1 shows the modeled $A(m, \infty)$ for program BT class C (BT.C), partitioned in 8 threads, with baselines conducted on 1, 2, 4 and 8 cores. Fig. 1 shows that sample

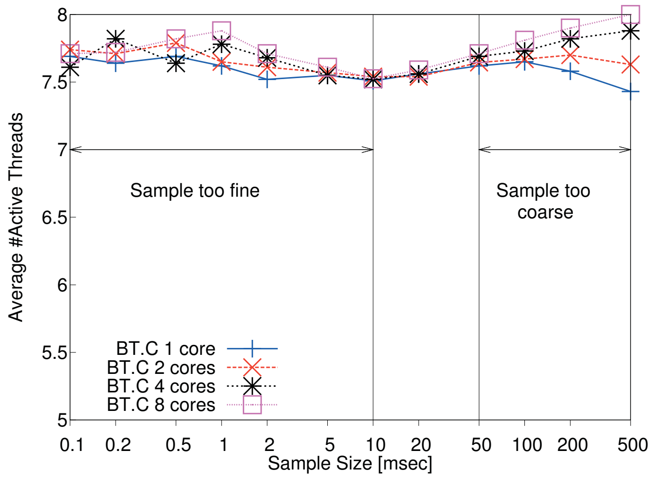

Fig. 1. Average number of active threads: effect of run-queue sample interval

intervals lower than 10 $msec$ are impractical, because the *procfs* updates the CPU service time of the threads in 10 $msec$ intervals. Because on our systems the scheduler quanta is around 100 $msec$ for UMA and 80 $msec$ for NUMA, the useful interval of sampling is between 10-80 $msec$. Based on our analysis, we conclude that the accuracy of our model depends on the relative difference between sample intervals and threads inter-synchronization time. We therefore opt for a value of 10 $msec$, the smallest in the useful range. Furthermore, the number of cores in the baseline run does not change the predicted value of $A(m, \infty)$ significantly. Therefore we select *one core* as the value for the baseline runs, to capture the largest number of samples.

*2) Speedup:* To validate the parallel speedup we compare modeled values of speedup against measurements. To evaluate the accuracy of the data dependency model independent of the accuracy of the memory contention model, we use predicted and measured values of $\omega(n)$ in modeling $S(m, n = m)$, and present both. Firstly, we validate the speedup of programs partitioned in different number of threads, and running on enough cores to execute them concurrently. Secondly, we fix the number of threads and execute them on different number of cores.

Partitioning all six program into 2, 4 and 8 threads, we perform baseline runs on one core to derive the average number of active threads. Using both modeled and measured values of $\omega(n)$, we compare the speedup model against measurements, when the program is running on 2, 4 and 8 cores. Table II shows the validation results. The average relative error of the model is $7.5\%$ for measured $\omega$ and $11.3\%$ for modeled $\omega$.

| Program | m | Measured $\mathbf{S(m, n = m)}$ | Modeled $\mathbf{S(m, n = m)}$ | |
|---|---|---|---|---|
| | | | measured $\omega$ | modeled $\omega$ |
| BT.C | 2 | 1.77 | 1.80 | 1.80 |
| | 4 | 2.52 | 2.65 | 2.52 |
| | 8 | 3.50 | 3.91 | 3.50 |
| EP.C | 2 | 1.98 | 1.99 | 1.99 |
| | 4 | 3.96 | 3.99 | 3.99 |
| | 8 | 7.83 | 7.98 | 7.98 |
| FT.B | 2 | 1.63 | 1.72 | 1.72 |
| | 4 | 2.23 | 2.33 | 2.30 |
| | 8 | 2.80 | 2.83 | 3.11 |
| IS.C | 2 | 1.93 | 1.95 | 1.95 |
| | 4 | 3.45 | 3.47 | 3.69 |
| | 8 | 5.02 | 5.08 | 6.65 |
| CG.C | 2 | 1.75 | 1.82 | 1.82 |
| | 4 | 2.27 | 2.20 | 2.76 |
| | 8 | 2.33 | 4.18 | 3.91 |
| SP.C | 2 | 1.32 | 1.30 | 1.30 |
| | 4 | 0.99 | 0.95 | 0.86 |
| | 8 | 0.97 | 0.81 | 0.83 |

TABLE II
SPEEDUP OF SIX DWARFS: MODEL VS MEASUREMENT ON UMA

We partition BT.C in $m$=8 threads for UMA system and $m$=16 threads on NUMA. We determine $A(m, \infty)$ using baseline runs on one core. For the $S(m, n)$ prediction, we used values of $\omega(n)$ predicted by our model and values directly measured, and we present both. We compared our prediction against speedup measurements, keeping the number of threads fixed and increasing the number of cores from 1 to $m$.

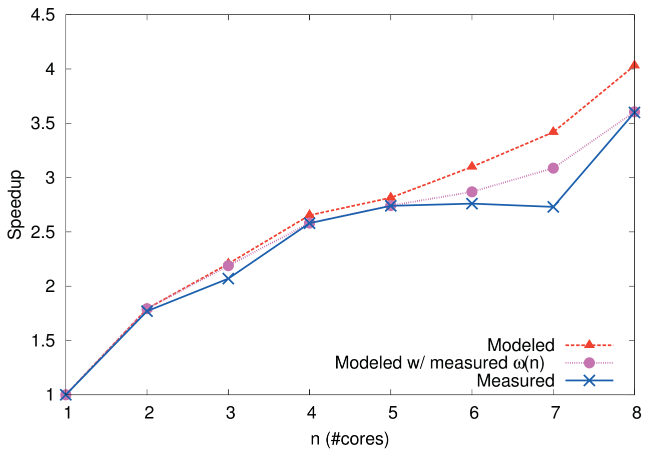

Fig. 2. Speedup of BT.C: model vs measurement on UMA

Fig. 2 and 3 show the predicted versus measured speedup on UMA and NUMA. For UMA, the accuracy of the prediction is very good, especially for runs on even number of cores.

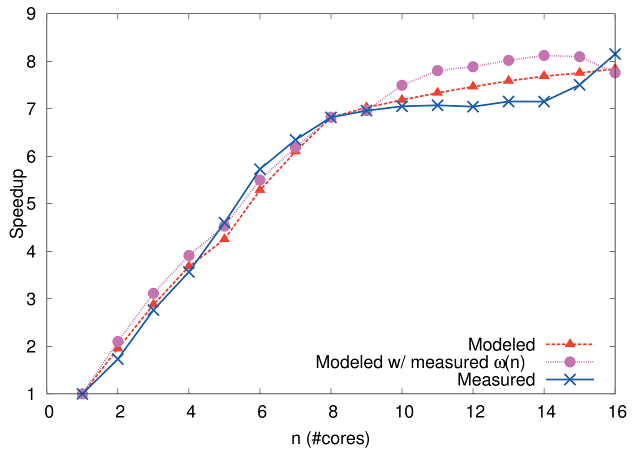

Fig. 3. Speedup of BT.C: model vs measurement on NUMA

For BT.C we noticed that the measured speedup varies among the five runs, especially when the number of threads $m$ does not divide by the number of cores $n$. Considering that measured speedup is our control value, the accuracy of the model is the lowest when $m$ does not divide by $n$ (maximum error is $25\%$ for predicted $S(8, 7)$ using measured $\omega$ and $13\%$ for predicted $S(8, 7)$ using modeled $\omega$). For the other cases, the model has much better accuracy. Overall, the average error for BT.C on UMA is $9\%$ for predicted $S(m, n)$ using measured $\omega(n)$ and $4\%$ for predicted $S(m, n)$ using modeled $\omega(n)$. On NUMA, the execution time on one core is determined using the PAPI virtual timer.

Two factors affect the accuracy of the speedup model: (i) the inaccuracy of $\omega(n)$, which is the most significant source of error for BT, SP and FT, and (ii) inaccuracy of $A(m, n)$. We hypothesize that $A(m, n)$ is related to inter-barrier time of the program. We observe a variation of up to $11\%$ for IS.C and $23\%$ for CG.C among the values of the average number of active threads predicted from the five baseline runs. We suspect that the inter-barrier time both programs may be smaller than our sample size of 10 msec, which is in-line with observations made by [19]. Overall, the average error across all programs is around $6\%$ for UMA and $11\%$ for NUMA. EP is the most straightforward to model, with an average error of less than $1\%$. These results show that the run-queue size is a good proxy for determining the average number of active threads of a program.

*3) Contention on Different Memory Systems:* To validate the memory contention model, we compare measured values of $\omega(n)$ on UMA and NUMA systems with predictions from our proposed model. We used hardware counters to determine the number of events `PAPI_TOT_CYC` (number of CPU cycles $C(n)$), `LAST_LEVEL_CACHE_MISSES` on NUMA and `PAPI_L2_TCM` on UMA for $r(n)$, and `PAPI_RES_STL` to determine the number of stall cycles. We derived work cycles as the difference between total cycles and stall cycles.

We first present experimental evidence for the assumption that $W(n)/r(n)$ does not change significantly when only one memory controller is used. We measure $W(n)$ and $r(n)$ and observe that on UMA, the number of $r(n)$ does not change significantly neither among different runs nor when changing the number of cores. On NUMA, the $r(n)$ increases with $n$, and therefore, we use a normalized value of $W(n)$ to $r(n)$. We computed the coefficient of variation of $W(n)$ and $r(n)$ for UMA and of $\frac{W(n)}{r(n)}$ for NUMA. For NUMA, we restricted the memory access to the local controller only. Table III shows

| System | Coefficient of variation | Programs | | | | | |
|---|---|---|---|---|---|---|---|
| | | **EP.C** | **BT.C** | **SP.C** | **FT.B** | **IS.C** | **CG.C** |
| UMA | $W(n)$ | 0.00 | 0.00 | 0.04 | 0.02 | 0.01 | 0.02 |
| | $r(n)$ | 1.30 | 0.02 | 0.11 | 0.06 | 0.00 | 0.03 |
| NUMA | $\frac{W(n)}{r(n)}$ | 3.30 | 0.03 | 0.06 | 0.02 | 0.04 | 0.83 |

TABLE III
LAST LEVEL MISSES $r(n)$ AND WORK CYCLES $W(n)$: COEFFICIENT OF VARIATION, WHEN $n$ = 1 TO 8 CORES USING ONE MEMORY CONTROLLER

that the variation of $W(n)$ to $r(n)$ is very small, confirming our assumption. Furthermore, Fig. 4 shows that the growth of the total cycles as a function of $n$ is correlated to the growth of the number of stall cycles and not to the growth of the work cycles.

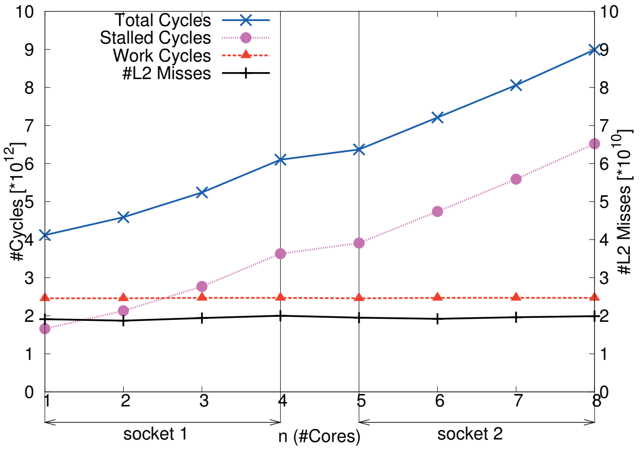

Fig. 4. CPU cycles of BT.C: measured on UMA

We validated experimentally the predicted value of $\omega(n)$ with different number of cores and two memory access policies:

(i) *Local* - using both sockets on UMA and one socket on NUMA for $n$ between 1 and 8

(ii) *Local Then Remote* - using memory controller 1 when only socket 1 is active ($n$ between 1 and 8) and both memory controllers when both sockets are active ($n$ between 9 and 16).

We chose these two policies as a trade-off between the increased memory latency when using both controllers and the increased memory contention when using only one memory controller. Moreover, when only $c$ cores from the total of $n$ are active, the rest of $n - c$ are not used to run any other program, and therefore we consider them not to contend for memory.

We validate all six dwarfs, by partitioning into $m = 8$ on UMA and $m = 16$ on NUMA, and running them on $n$ = 1 to 8 cores on UMA and on $n$ = 1 to 16 on NUMA. We did not bind individual threads, allowing the OS freedom to enforce our

| Program | Measured | | | | Modeled | | | |
|---|---|---|---|---|---|---|---|---|
| | UMA | | NUMA | | UMA | | NUMA | |
| | $\omega(4)$ | $\omega(8)$ | $\omega(8)$ | $\omega(16)$ | $\omega(4)$ | $\omega(8)$ | $\omega(8)$ | $\omega(16)$ |
| EP.C | 0.00 | 0.00 | -0.09 | 0.37 | 0.00 | 0.00 | -0.09 | 0.32 |
| IS.C | 0.07 | 0.56 | -0.01 | 0.51 | 0.07 | 0.39 | -0.01 | 0.45 |
| BT.C | 0.44 | 1.18 | 0.15 | 1.00 | 0.44 | 0.95 | 0.15 | 0.98 |
| CG.C | 0.91 | 2.41 | 0.35 | 1.62 | 0.60 | 0.91 | 0.50 | 1.61 |
| FT.B | 0.72 | 1.76 | 0.36 | 2.11 | 0.72 | 1.51 | 0.36 | 1.84 |
| SP.C | 3.34 | 7.04 | 1.93 | 5.19 | 3.34 | 6.86 | 1.93 | 3.99 |

TABLE IV
MEMORY CONTENTION: MODEL VS MEASUREMENTS ON UMA & NUMA

specified NUMA policy. Since we model two-socket systems, we need 3 measurements of $C(n)$ as the model inputs: for UMA, we use $C(1)$, $C(4)$ and $C(5)$ while for NUMA we use $C(1)$, $C(8)$ and $C(9)$.

In the prediction of the memory contention, the model faithfully tracks measured value of memory contention, with two distinct intervals of inaccuracies: (i) on small number of cores on NUMA and (ii) on large number of cores for UMA. On NUMA, for IS, BT and CG, oversubscription affects caching in a positive way, improving the performance when $n = 2$ to 4 cores compared to the run on one core, and therefore $\omega(n) < 0$ for these values. On UMA, for large number of cores, we suspect that the $\Delta C$ parameter from equation 9 might underestimates the impact of memory request intensity when large number of cores are contending. When determining the relative error for memory contention, we chose the maximum value of measured contention as the baseline, $\omega(n = 16)$ for NUMA and $\omega(n = 8)$ for UMA, because some memory contention values were very close to zero. Our results are presented in Table IV.

We hypothesise that the source of the inaccuracy for small values of $n$ is the heavy oversubscription. As shown by [19], when the oversubscription factor (i.e. ratio of threads to cores) is higher than four, NUMA systems have a performance penalty induced by context switching. However, to the best of our knowledge, there is no published work on models for predicting scheduler fairness in oversubscription scenarios, and therefore we have used the processor sharing discipline in our model of speedup.

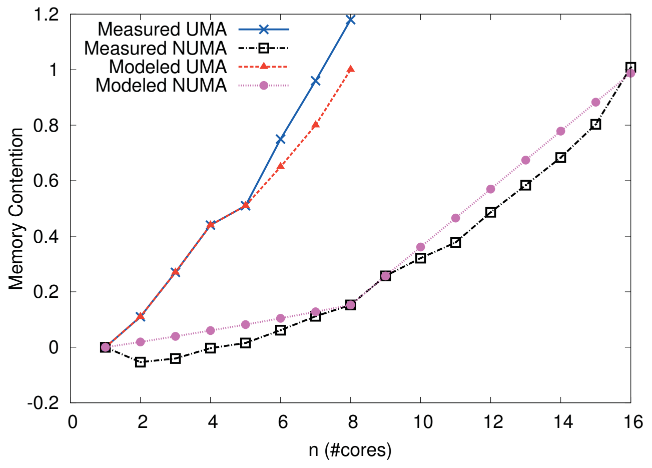

Fig. 5. Memory contention for BT.C: model vs measurement

Our model shows that memory contention is significant for both UMA and NUMA. On UMA, the growth rate of $C(n)$

is fairly constant, with a pronounced per-socket pattern of growth. In Fig. 5 the growth from $n$=1 to $n$=4 cores of $\omega(n)$ UMA resembles the growth from $n$=5 to $n$=8. This is due to the fact that each socket has a dedicated bus to the main memory, and the two buses are of identical bandwidth and latency. On NUMA, the growth from $n$=1 to $n$=8 cores, when only the local controller is active, is smaller than the growth from $n$=9 to $n$=16, which highlights the performance impact of accessing both local and remote controllers.

Table IV shows predicted and measured values of the memory contention for all six dwarfs. We found that the programs have very different memory contention values, from EP where the contention is completely negligible, to SP which seems to be heavily impacted by the memory wall.

## B. Analysis

*1) Impact of Problem Size on Speedup Loss:* We apply our model to predict the speedup loss due to data dependency and memory overhead for program SP using problem sizes W, A, B and C (with ratio of problem sizes W:A:B:C≈1:4:16:64). We chose this dwarf because it has the highest speedup loss from all the studied programs. For clarity of interpretation we predict the loss for a two-socket UMA system.

We predict the speedup loss of SP when partitioned from $m$ = 1 to 8 threads. We have run the baseline run for predicting $A(m,\infty)$ and we derive $\omega(n)$ using measured $C(1)$, $C(4)$ and $C(5)$ on our UMA system.

Fig. 6 shows the speedup loss for SP with four problem sizes. SP has a counter-intuitive behaviour: *the speedup reduces as the problem size is increased.* We can see that small problem sizes lead to larger speedup loss due to data dependency. This is expected considering that data dependency in SP is mostly induced by barriers [16]. Smaller input sizes lead to small intra-barrier times. Therefore, it is expected that problem W has data dependency as the most significant source of speedup loss. For larger inputs, data dependency reduces significantly.

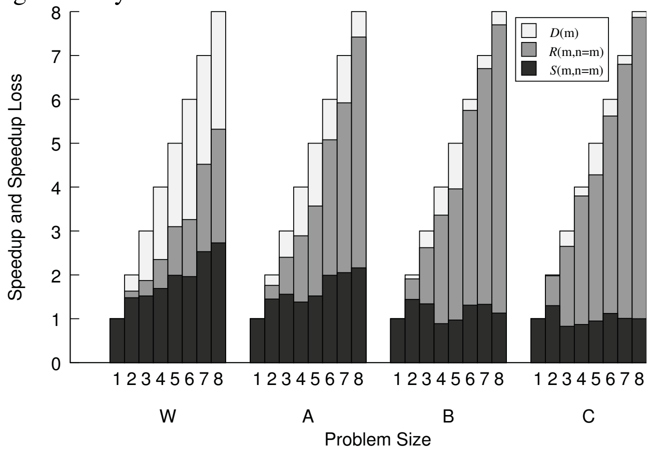

Fig. 6. Speedup of SP: varying problem sizes on UMA

For large input sizes, the main source of speedup loss is the memory overhead, which increases both with the number of cores and with the problem size. For problem size C, $S(m, n=m)$ degrades very close to 1, when $m>2$, which means that allocating more than two cores to SP.C decreases performance and increases number of resources used.

This analysis has two conclusions: (i) that SP is particularly impacted by resource contention, and therefore higher memory bandwidth, either through faster memory, NUMA or increased caching should benefit it, and (ii) matching the input size to the right number of cores should reduce significantly both execution time and number of cores required. The second implication is very important relative to program execution cost. For example, if the cost is expressed as the energy usage of the program, then the cost depends both on execution time and number of cores used, and our model can help reduce both factors.

*2) Predicting Number of Cores that Increases Speedup:* Motivated by the conclusion of the analysis in the section III-B1 we apply our model to determine the optimal number of cores that should be allocated to a program. Specifically, we are interested in finding the range of $n$ for which $S(m,n)$ is increasing as $n$ increases. Using equation 1 the condition for which $S(m,n)$ is increasing with $n$ is (if $\omega(n)>-1$ and assuming there are enough threads, so $m>n$):

$$\frac{d\omega(n)}{dn} < S(m,n) \cdot \frac{\partial A(m,n)}{\partial n} \tag{11}$$

The optimal number of cores is the value of $n$ that maximizes $S(m,n)$ and for which equation 11 holds. We use numerical differentiation to compute the values of $n$ for which $S(m,n)$ increases. The range of $n$ that satisfies equation 11 is a union of $j$ intervals $\cup_j[n_{min,j}, n_{max,j}]$. As we are interested in maximizing the $S(m,n)$, then the optimal number of cores is $n_{max,j}$ for which $max\{S(m, n_{max,j})\}$.

For SP, we have determined the optimal values of $n$ (values rounded to integers): $n_W$ = 12, $n_A$ = 8, $n_B$ = 2, $n_C$ = 2. For SP.B and SP.C, the optimal values are therefore less than the average number of active threads. Increasing the number of cores past 2 cores for SP.B and SP.C is detrimental, since it increases both execution time and number of resources used. To verify the validity of this prediction, we have run the

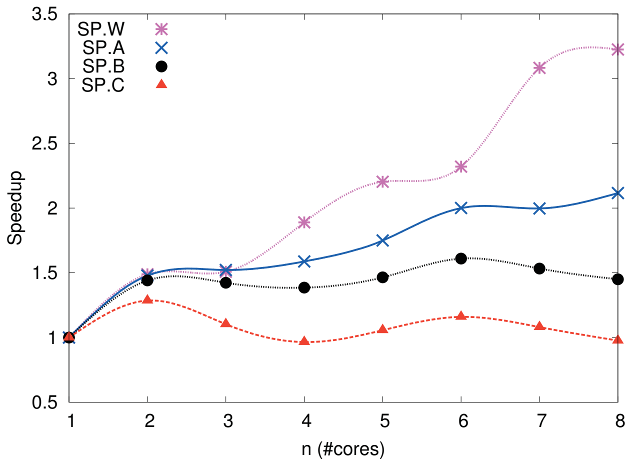

Fig. 7. Measured speedup of SP: varying problem sizes on UMA

programs with $m=8$, on the UMA system (5 times for A, B and C, 10 times for SP.W) and computed averages of the speedups. Fig. 7 shows the plot of the measured speedup, with spline smoothing. The model accurately describes the intervals of $n$ for which $S(8, n)$ is increasing with $n$. For SP.C the value of $n = 2$ appears to be the maximum value. For SP.B, values $n = 2$ and $n = 6$ both lead to local maxima, but since the relative difference between $S(8, 2)$ and $S(8, 6)$ is 8%, our model selects $n = 2$ instead of $n = 6$ as the number of cores that maximizes exploited speedup.

*3) Impact of Changing from UMA to NUMA:* As NUMA systems have recently become the dominating architecture in multicore servers, an important question for users of parallel program is what is the impact of switching from UMA to NUMA for memory-bounded programs.

As SP is severely memory-bounded, we apply the model to derive the speedup for a two-socket NUMA, comparing the results with a two-socket UMA. Fig. 8 shows the growth

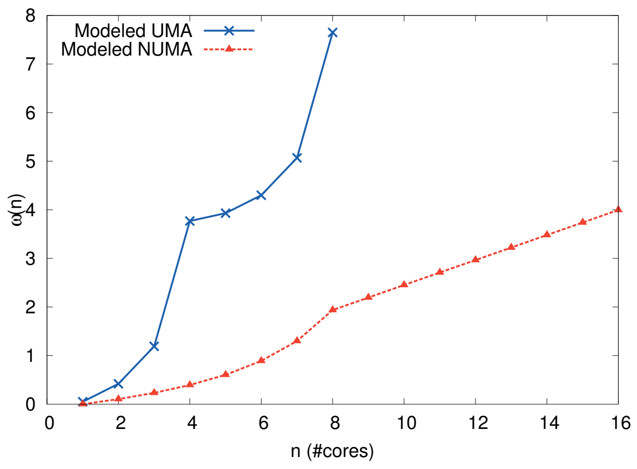

Fig. 8. SP.C: modeled memory contention on UMA & NUMA

of memory contention with $n$ for UMA and NUMA. The NUMA has clear performance superiority, both in terms of single-socket performance and dual-socket performance. We apply the model to derive the speedup for SP.C partitioned into 16 threads. Fig. 9 shows that changing the system from UMA to NUMA results in improved scalability for a memory bounded program such as SP. The maximum speedup

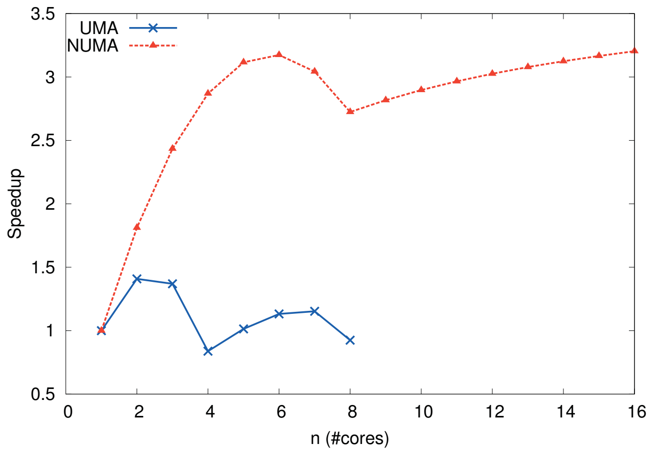

Fig. 9. SP.C: modeled speedup on UMA & NUMA

increases more than two times, from maximum value of $S_{UMA}(16, 2) = 1.40$ to $S_{NUMA}(16, 6) = 3.17$. There are many reasons for the performance improvements on NUMA compared to UMA: two memory controllers instead of one, higher memory throughput for each controller, faster bus speeds and more cache memory. However, even with these improvements, SP still hits the memory wall. The number of cores that maximizes the speedup is six on NUMA, almost three times lower than the number of cores available. Since when using one to eight cores, the system uses only one memory controller, it follows that SP does not benefit much from the second memory controller, because the maximum speedup is reached when using six cores. From nine cores onwards, the second memory controller is activated, and the speedup grows, but overall its value does not exceed $S(16, 6)$ cores. The conclusion of the analysis is that SP.C benefits from the increased cache size and bus/memory speed, but not from adding an additional memory controller.

## IV. Related Work

**Performance Prediction of Program Parallelism.** Current work on performance prediction for parallel program can be classified based on the trade-offs among accuracy, intrusiveness and ease of use.

Models relying on instrumentation, such as [4], [5], [6] obtain detailed insight and generally have good accuracy. However, instrumentation is intrusive and often prevents aggressive compiler optimizations. For example, the OPARI instrumentor, which is a component of the KOJAK, TAU, Scalasca [23] and ompP [24] profilers, prevents the usage of implicit barriers, which in turn prevents the OpenMP NOWAIT clause, thus forcing the threads to perform an additional synchronization operation. Instrumentation may slow down the program or interfere with cache sensitive areas, and therefore increase the overhead of the parallel programs. Some vendors provide highly optimized versions of popular parallel kernels (such as BLAS or LAPACK) which come directly compiled as libraries. Without access to the source code of such products, prediction methods relying on instrumentation may not be possible to apply. Lastly, instrumentation methods lack generality, because often they cannot be applied across different programming languages, threading packages and runtime systems. In contrast, our model does not require any instrumentation and can be applied independent of the programming language.

Recently, there is a shift towards empirical models for performance prediction [8]. The performance of large multicore programs often depends on hundreds of parameters including programming model, memory architecture, problem size, partition size among others. Models based on empirical data [7], [9], [10], [11] are very promising for studying this wide parameter space. Generally, these models use multiple linear regression [9], [11], machine learning [10] or neural networks [11] using data measured in various parameter configurations to relate the measured performance metrics of the program to the changes in the configuration parameters. These models have good accuracy and generally low intrusiveness, and therefore they are used for decision making in ACTOR runtime system [25]. Our approach shows that important performance insights can also be obtained by using an analytical model with inputs derived from measurements. Furthermore, our model and existing regression methods can be used together to reduce the measured data required to apply the empirical models.

Among the general models that predict the performance of a program, Amdahl's law [12] is the most widely known. However, this model and extensions [26], [27] do not explain the effect of resource contention. Directed acyclic graph models [13], [14], [18] use the inherent parallelism to bound the speedup and the parallel efficiency, with a theoretical error for estimating the speedup of 33%. However, it is not clear how the model inputs are derived. In contrast, we model both the relationship between speedup, average number of active threads and overhead, and also derive practical methods of deriving them.

To the best of our knowledge, this is the first proposed model that uses the run-queue of the operating system as a proxy to quantify the parallelism of a shared-memory program. Kelly et al. [28] have studied the effect of increasing and decreasing the number cores on parallel web servers with dynamic workload and derived three operational analysis laws that describe the queuing delays and the performance implications of expanding or reducing capacity. Their main focus is a model for capacity planning and dynamic resource provisioning for parallel web servers, but we propose a model to quantify the speedup of a program and the speedup loss due to data dependency and memory contention.

**Memory Contention in Multicore Systems**

Technological trends suggest that increasing the number of cores cannot be matched by the increase in memory bandwidth and speed [29], and thus the memory contention problem [30] is receiving renewed attention. In general, recent analytical models are proposed in conjunction with software and hardware improvements to available multicore systems. These models can be grouped into two categories: approaches to reduce the burden on the memory and approaches to increase the efficiency of existing memory systems. In the first category, cache integrated network interfaces [31] and utility-based cache partitioning [32] have been proposed. In the second category, Liu et. al [33] propose an analytical model that captures the effects of bandwidth sharing among cores with the goal of deriving the optimal sharing scheme. Recently, increased attention has been devoted to studying scheduling disciplines for memory requests inside the memory controllers [34], [35]. These studies rely on instruction accurate simulations to validate their approaches [33]. However, as shown in our analysis, program size affects significantly the performance of memory bounded programs, and simulation is feasible only for small problem sizes. Furthermore, the majority of existing studies support design decisions for software or hardware improvements to available multicore systems, while we focus on the performance penalty of existing UMA and NUMA systems. On this topic, we have found little existing work overlapping our objective. Long et. al [36] provide a taxonomy of programs based on their memory requirements. Mostley et. al [37] propose an analytical model of the contention for on-chip resources among hardware threads in simultaneous multithreading processors. However, we are interested in predicting how the workload is affected by contention for off-chip resources, such as buses and memory controllers. Our approach offers a complementary model for studying how various large shared-memory programs are affected by off-chip resource contention in UMA and NUMA systems.

## V. Conclusions and Future Work

In this paper we proposed an analytical model for the parallelism performance of a shared-memory program. Our model determines the speedup of a program, deriving the speedup loss from data dependency and runtime overhead due to memory contention in UMA and NUMA systems. Using the trace of the run-queue size and hardware events counters, our model shows that important performance insights for a shared-memory program can be derived using generally available inputs obtained from non-invasive methods. The number of inputs required by the model is independent of any characteristic of the program such as number of threads or cores, and depends only on the architecture of the memory system. For a single socket system, the model requires measurements of CPU cycles on two configurations, and for multiple sockets the model requires three configurations. This confers considerable advantages in terms of practicality of applying the model and generality with respect to programming language, runtime system and memory model. We validated the model against measurements conducted on six OpenMP dwarfs from the NPB 3.3 benchmark and showed that model results differ from measurements on average by 4-9% on UMA and 6-11% on NUMA. We applied the model to study the impact of the memory wall, and showed that for a program with large memory contention, increasing the problem size decreases performance. We used our model to derive analytically the number of cores that maximizes the speedup. This has the potential to significantly reduce execution cost, by reducing both the execution time and number of cores required. Furthermore, we applied our model to study the benefits of switching from an UMA system to a NUMA system, and showed that for programs with very large memory contention, such as SP.C, even adding an additional memory controller may not improve speedup.

Our model can be extended in several directions. Because it requires generally available and low-overhead inputs, it can be extended to provide decision support for online autotuners of parallel programs. In another direction, we are currently extending the memory contention model to account explicitly for various subcomponents, especially the overhead of maintaining cache coherence and the burstiness of memory traffic. Lastly, the suitability of the run-queue to derive the parallelism profile of programs with significant I/O operations, distributed-memory or distributed shared-memory needs to be investigated.

## Acknowledgments

We are indebted to Y. C. Tay for suggestions on modeling the memory contention.

## References

[1] G. L. Steele, Jr., "Parallel Programming and Code Selection in Fortress," in *Proc. of 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–1, New York, USA, 2006.

[2] T. El-Ghazawi and F. Cantonnet, "UPC Performance and Potential: A NPB Experimental Study," in *Proc. of 14th ACM/IEEE Conference on Supercomputing*, pages 1–26, 2002.

[3] R. W. Numrich and J. Reid, "Co-arrays in The Next Fortran Standard," *SIGPLAN Fortran Forum*, 24(2):4–17, 2005.

[4] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Caşcaval, "How Much Parallelism Is There in Irregular Applications?", in *Proc. of 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–14, Raleigh, USA, 2009.

[5] M. Kumar, "Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications," *IEEE Transactions on Computers*, 37(9):1088–1098, 1988.

[6] D. Sehr and L. V. Kale, "Estimating the Inherent Parallelism in Prolog Programs," in *Proc. of 3rd International Conference on Fifth Generation Computer Systems*, pages 783–790, Tokyo, Japan, 1992.

[7] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores," in *Proc. of 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 250–259, Toronto, Canada, 2008.

[8] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel, "Practical Performance Models for Complex, Popular Applications," in *Proc of 31st ACM International Conference on Measurement and Modeling of Computer Systems*, pages 1–12, New York, USA, 2010.

[9] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, "A Regression-based Approach to Scalability Prediction," in *Proc. of 22nd Annual International Conference on Supercomputing*, pages 368–377, Island of Kos, Greece, 2008.

[10] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, "A Case for Machine Learning to Optimize Multicore Performance," in *Proc of 1st USENIX Conference on Hot Topics in Parallelism*, Berkeley, USA, 2009.

[11] K. Singh, M. Curtis-Maury, S. McKee, F. Blagojevic, D. Nikolopoulos, B. de Supinski, and M. Schulz, "Comparing Scalability Prediction Strategies on an SMP of CMPs," in *Proc. of 16th International European Conference on Parallel and Distributed Computing*, pages 143–155, Ischia, Italy, 2010.

[12] G. M. Amdahl, "Validity of The Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proc. of AFIPS Spring Joint Computer Conference*, 1967.

[13] D. Eager, J. Zahorjan, and E. Lazowska, "Speedup versus Efficiency in Parallel Systems," *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[14] K. Sevcik, "Characterizations of Parallelism in Applications and Their Use in Scheduling," in *Proc. of 10th ACM Conference on Measurement and Modeling of Computer Systems*, pages 171–180, Berkeley, USA, 1989.

[15] A. B. Downey, "A Model for Speedup of Parallel Programs," Techical Report, EECS Department, UCB/CSD-97-933, University of California, Berkeley, USA, 1997.

[16] H. Jin, M. Frumkin and J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and its Performance," Technical Report, NAS System Division, NAS-99-011, NASA Ames Research Center, Moffett Field, USA, 1999.

[17] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill Higher Education, 1992.

[18] A. B. Downey, "A Parallel Workload Model and its Implications for Processor Allocation," in *Proc. of 6th IEEE International Symposium on High Performance Distributed Computing*, pages 112-123, Portland, USA, 1997.

[19] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng, "Oversubscription on Multicore Processors," in *Proc of 24th IEEE International Symposium on Parallel & Distributed Processing*, pages 1-11, Atlanta, USA, 2010.

[20] T. Karkhanis and J. E. Smith, "A Day in the Life of a Data Cache Miss," in *Proc. of 2nd Workshop on Memory Performance Issues*, Anchorage, USA, 2002.

[21] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Computer Architure News*, 23(1):20–24, 1995.

[22] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, 1991.

[23] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The SCALASCA Performance Toolset Architecture," in *Concurrency & Computation: Practice and Experience*, 22(6):702–719, 2010.

[24] K. Fürlinger and M. Gerndt, "ompP: A Profiling Tool for OpenMP," in *Proc. of 1st International Workshop on OpenMP*, Eugene, USA, 2005.

[25] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos, "Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes," *IEEE Transactions on Parallel Distributed Systems*, 19(10):1396–1410, 2008.

[26] D. H. Woo and H.-H. S. Lee, "Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era," *IEEE Computer*, 41(12):24–31, 2008.

[27] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *IEEE Computer*, 41(7):33–38, 2008.

[28] T. Kelly, K. Shen, A. Zhang, and C. Stewart, "Operational Analysis of Parallel Servers," *Proc. of 16th IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, page 227–236, Baltimore, USA, 2008.

[29] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the Bandwidth Wall: Challenges and Avenues for CMP Scaling," *SIGARCH Computer Architure News*, 37(3):371–382, 2009.

[30] A. K. Nanda, H. Shing, T.-H. Tzen, and L. M. Ni, "Resource Contention in Shared-Memory Multiprocessors: A Parameterized Performance Degradation Model," *Journal of Parallel and Distributed Computing*, 12(4):313–328, 1991.

[31] S. G. Kavadias, M. G. Katevenis, M. Zampetakis, and D. S. Nikolopoulos, "On-chip Communication and Synchronization Mechanisms with Cache-integrated Network Interfaces," in *Proc. of 7th ACM International Conference on Computing Frontiers*, pages 217–226, Bertinoro, Italy, 2010.

[32] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proc. of 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, Orlando, USA, 2006.

[33] F. Liu, X. Jiang, and Y. Solihin, "Understanding How Off-chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance," in *Proc. of 16th International Symposium on High Performance Computer Architecture*, Bangalore, India, 2010.

[34] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *Proc. of 16th International Symposium on High Performance Computer Architecture*, Bangalore, India, 2010.

[35] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith, "Fair Queuing Memory Systems," in *Proc. of 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, Orlando, USA, 2006.

[36] G. Long, D. Fan, and J. Zhang, "Characterizing and Understanding the Bandwidth Behavior of Workloads on Multi-core Processors," in *Proc. of 15th International European Conference on Parallel and Distributed Computing*, pages 110-121, Delft, The Netherlands, 2009.

[37] T. Moseley, J. Kihm, D. Connors, and D. Grunwald, "Methods for Modeling Resource Contention on Simultaneous Multithreading Processors," in *Proc. of 23rd International Conference on Computer Design*, pages 373–380, San Jose, USA, 2005.