Scalable Computing: Practice and Experience
Volume 10, Number 2, pp. 217–228. http://www.scpe.org

ISSN 1895-1767
© 2009 SCPE

POWER-AWARE SPEED-UP FOR MULTITHREADED NUMERICAL LINEAR ALGEBRAIC SOLVERS ON CHIP MULTICORE PROCESSORS

JAYANTA MUKHERJEE[*] AND SOUMYENDU RAHA[†]

Abstract. With the advent of multicore chips new parallel computing metrics and models have become essential for redesigning traditional scientific application libraries tuned to a single chip. In this paper we evolve metrics specific to generalized chip multicore processors (CMP) and use them for parallel performance modeling of numerical linear algebra routines that are commonly available as shared object libraries tuned to single processor chip. The study uses a thread parallel model of parallel computing on CMPs. POSIX threads (pthread) have been used due to the wide acceptance and availability. The shortcoming of the POSIX threads for numerical linear algebra in terms of data distribution has been overcome by tuning algorithms so that a particular thread will operate on a specific portion of the matrix. The paper studies tuned implementations of the conventional a few parallel linear algebra method as examples on a generalized CMP model. For formulating a speed-up metric, this work takes into consideration the power consumption and the effect of memory cache hierarchy.

1. Introduction.

Chip multicore processor (CMP) architectures are designed with the aim to boost performance by providing on-chip parallelism while reducing power consumption and heat output by integrating two or more processor cores on to a single chip. Power has become the most critical constraint in the design of many on-chip systems including the CMP systems. The analysis of both the software and hardware for overall power consumption is needed for CMPs and similar on-chip systems. The main objective of the present study is to introduce power as a parameter into a realistic speed-up metric for numerical linear algebra routines running on CMPs.

For embedded devices the software can be made power-aware by analyzing the pipeline and the data flow from the instruction set [32]. The estimation of power consumption is done using instruction level modeling and considering base cost, effect of circuit state and effect of cache misses. The software then can be tuned to consume less power based on this analysis. The basic idea behind the instruction level power models of a given processor is to study the energy consumption rate [33]. The energy cost of a program is then simply the product of its average power cost and its running time. These concepts can be applied to CMPs executing numerical linear algebra routines.

CMPs use communication fabrics in the form of networks-on-chips (NoCs) that scale to handle the communication demand among the cores [5, 24]. Thus a parallel computing model and speed-up metric for CMPs must take into account the power consumed in on-chip communication as well as that in computation.

We review some of the existing results and concepts leading to the derivation of our speed-up metric. Some duty cycle modulations are used in [4] to emulate performance asymmetry which is effective because of the unpredictability of performance. Also, [4] shows that limitations in scalability arise due to differences in computation power of the individual cores and not due to communication latencies. Thus load-balancing among the on-chip threads is an unexplored aspect of power and heterogeneity aware parallel speed-up metrics for CMPs. In [35] an experimental approach may be found for determining whether the differences in the implementation, design and scheduling of threads would produce significant differences in performance. But a more general approach will be to develop an analytical model that puts together parallel efficiency, granularity of parallelism, voltage/frequency scaling and the power consumption in a CMP. Such a general analytical model is discussed in [27] and the experimental work corroborating the model shows that the optimized parallel computing can bring significant power savings and still meet a given performance target provided granularity and voltage/frequency levels are chosen and combined judiciously. The particular choice, however, is dependent on the application's parallel efficiency curve and the process technology utilized. This analytical model does not take into consideration the cache hierarchy, the cache hit-miss statistics and cache power consumptions. The hierarchical cache traffic interacts with the spatial locality of data. In addition, static power consumption and leakage loss [12, 28] during waiting for synchronization of threads also must be modeled in a power aware speed-up metric.

Available speed-up metrics do not consider any extra overhead due to any check-pointing or fault-tolerance techniques adopted at the software or middleware level. For any application running for very long duration,

[*]Department of Computer Science, Thomas M. Siebel Center for Computer Science, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Urbana, IL 61801, USA.

[†](Corresponding Author) Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560012, India; email:raha@serc.iisc.ernet.in

the roll-backs and recovery of computation which in turn involves power, communication and memory traffic overheads, becomes a necessary consideration.

In [23] recursive blocked algorithms are used to split the numerical linear algebra problems and thus the technique mainly improves on the temporal locality. Most of the work in the recursive algorithm is performed by efficient Basic Linear Algebra Subprograms (BLAS) library routines optimized for the calling application's host architecture. The approach uses OpenMP induced threads and produces the results for both uniprocessor and symmetric multi processing (SMP) platforms. Algorithms for efficient scheduling and synchronizing of threads on multiprogrammed SMPs have been developed in [3]. This work has modified the Linux kernel (version 2.2.15) in order to implement the scheduler as a run-time library. In [18] higher level BLAS and LAPACK-style [2] codes have been used for computing the pivoted Cholesky factorization routine for positive semi-definite matrices. Data distribution among different threads on a CMP can be done using block cyclic data distribution in the similar way as ScaLAPACK routines [10]. Work on new data structures for dense matrices and LAPACK design for chip multicore processors can be found in [8, 17]. More on important multicore architecture issues, programming models, algorithms requirements and software design for numerical library research and development can be found in [20]. Polyhedral models of data dependence and program transformations were considered in [6]. Multigrid applications can be found in [14].

The scalable implementation of BLAS and LAPACK [2] libraries on SMP systems [10] can be the basis for the modification needed in these routines for efficiently running on the CMPs. In this paper, data manipulation and computation for selected BLAS and LAPACK routines with POSIX threads [7] have been done with respect to the latency in physical and logical access to the processing element on the chip. Some of the implementations on a single chip multicore processor result in an SMP-like parallel computing. Further, the speed-up metrics discussed in the present paper lead to tuning the algorithm and implementation for power and memory access latency.

In its experimental part the present paper studies the performance of a few linear algebra subroutines on a simple CMP using multithreading. The computationally studied numerical methods are Jacobi Iteration and Cholesky factorization. The experiments compare the estimated actual speed-up against theoretical estimates obtained by the speed-up metric considering interconnect delay based communication latency, hierarchical cache miss rate and power (including leakage loss) consumed in communication, computation and waiting for synchronization.

2. Speed-up Metric.

The parameters that affect the performance of a program on a CMP are on-chip interconnect delay, connection topology (e.g., hypercube, mesh) affecting switching and communication latency, power leakage and power dissipation (including thermal dissipation) and cache access time (as affected by cache hierarchy). Additionally, for a CMP densely packed with compute and other resources may experience transient faults and consequent maintenance of redundancy, check-pointing and roll-back also affect the performance.

A measure of the effectiveness of the availability of multiple processing units on the chip is given by the speed-up achieved over the same operation being done on a single processing unit [16]. The *speed-up*, S, is defined as the ratio of sequential to parallel execution time.

$$S = \frac{T_1(w_s)}{T_N(w_p)}, \tag{2.1}$$

where, w_s is the sequential and w_p is the parallel workload (instructions or computations). $T_1(w_s)$ is the sequential execution time or the amount of time needed to complete the workload w_s on a single processor and $T_N(w_p)$ is the parallel execution time or the amount of time to complete the workload on N processors. For a CMP system speed-up can be simply defined as the ratio of sequential execution time on a single core to the execution time using multiple cores in parallel:

$$S = \frac{T_1(w_s)}{T_n(w_p)}, \tag{2.2}$$

where, w_s is the sequential and w_p is the parallel workload which is obtained as combining the sequential workload at each node in the task graph and the overhead due to parallelization. We express $w_p = OF \times w_s$ where OF is the overhead factor which is greater than or equal to 1 and $T_n(w_p)$ is the execution time or the amount of time needed to complete the same operation on n cores. Within the parallel workload w_p, let PF be the fraction of the workload that can actually be parallelized. Then, following [19, 21], the relationship between frequency, average number of cycles per instruction and the execution time using n cores is given as

$$T_n(w_p) = (1 - PF)w_p \frac{CPI_1}{f_1} + \frac{PF \times w_p}{DOP} \frac{CPI_n}{f_n}, \tag{2.3}$$

where, CPI is the average number of cycles per instruction and f is the operating frequency. The subscripts denote the number of cores for which the respective quantities are being considered. Here DOP is the *average* degree of parallelism which is defined as the average number of cores that can be busy computing a workload over $\frac{CPI_n}{f_n}$ units of time. The speed-up can be expressed as

$$\begin{aligned} S = \frac{T_1(w_s)}{T_n(w_p)} &= \frac{w_s \frac{CPI_1}{f_1}}{(1 - PF) \times w_p \times \frac{CPI_1}{f_1} + \frac{PF \times w_p}{DOP} \times \frac{CPI_n}{f_n}} \\ &= \frac{\frac{CPI_1}{f_1}}{(1 - PF) \times OF \times \frac{CPI_1}{f_1} + \frac{PF \times OF}{DOP} \frac{CPI_n}{f_n}}. \end{aligned} \tag{2.4}$$

The parallel efficiency [11] of an application running on N processors, $\epsilon(N)$, can be written as

$$\epsilon(N) = \frac{T_1}{NT_N}. \tag{2.5}$$

Parallel efficiency for any application running on a CMP with n cores, $\epsilon(n)$ can be written as

$$\begin{aligned} \epsilon(n) = \frac{T_1(w_s)}{nT_n(w_p)} &= \frac{w_s \frac{CPI_1}{f_1}}{n(1 - PF)w_p \frac{CPI_1}{f_1} + n\frac{PF \times w_p}{DOP} \times \frac{CPI_n}{f_n}} \\ &= \frac{\frac{CPI_1}{f_1}}{n(1 - PF) \times OF \times \frac{CPI_1}{f_1} + n\frac{PF \times OF}{DOP} \times \frac{CPI_n}{f_n}}. \end{aligned} \tag{2.6}$$

2.1. Power Consumption Model. We make same assumptions as in [27] regarding homogeneity, availability, simultaneous activity of the cores on a generalized CMP. It is also assumed that cores that are not being used are completely shut-down. Total power (P_{total}) consumption consists of three components: static power (P_{static}) which is the component required for biasing the circuit elements, active power (P_{active}) that is consumed during computation and leakage power ($P_{leakage}$) which is independent of any activity. Leakage power is consumed even if a core is not doing any computation. Considering all these components in a way similar to the approach in [27], we can write

$$P_{total} = P_{static} + P_{active} + P_{leakage}. \tag{2.7}$$

The static power can be expressed in terms of the biasing current and supply voltage:

$$P_{static} = V_{supply} I_{bias} = V_{DD} I_{static}, \tag{2.8}$$

where, V_{DD} or V_{supply} is the supply voltage and I_{bias} is the biasing current or static current (I_{static}). The leakage power dissipation varies with the CMOS process technology. For CMPs the leakage power is significant due to the smaller feature size of the CMOS process used and and depends on the number of processing entities (i. e., cores) on it. Also, as the CMOS process enters the sub-65 nm technology, the leakage component of power increasingly becomes significant [22, 27]. The active power consumption includes both dynamic power ($P_{dynamic}$) and short-circuit ($P_{shortcircuit}$) power. Active power is proportional to the instruction processing activities. It may be written as

$$P_{active} = P_{dynamic} + P_{shortcircuit},$$

so that

$$P_{total} = P_{static} + P_{dynamic} + P_{shortcircuit} + P_{leakage}. \tag{2.9}$$

Leakage current has five basic components: the sub-threshold leakage current (I_{sub}), the gated oxide leakage current (I_{gate}), the reverse biased PN junction current, the punch through current and the gate tunneling current [9]. We analyze the total leakage power by considering only the gated oxide leakage current I_{gate} and the sub-threshold leakage (I_{sub}) which are likely to be more significant. The sub-threshold leakage current (I_{sub}) is modeled as

$$I_{sub} = I_0 e^{\frac{V_{gs}-V_t-V_{off}}{n_{ds} v_T}} \left(1 - e^{-\frac{V_{ds}}{v_T}}\right) \tag{2.10}$$

where I_0 is the leakage current at 25°C, V_t is the threshold voltage, V_{gs} is the gate to source voltage, V_{off} is an empirically determined model parameter [30], V_{ds} is the drain to source voltage and v_T is the thermal voltage obtained as $\frac{kT}{q}$, q being the electron unit charge, T, the temperature in Kelvin and k, the Boltzmann constant. The constant n_{ds} is experimentally determined for a particular technology. The dynamic power is computed as in [26]: $\alpha C V_{DD}^2 f$, α being the activity factor, C a capacitative constant and f, the switching frequency (taken same as the operating frequency) and V_{DD}, the supply voltage. The short circuit power dissipation is expressed as $P_{shortcircuit} = \frac{1}{2} I_{shortcircuit} V_{DD}$ [1], with $I_{shortcircuit}$ as the short circuit current. The gate oxide leakage current I_{gate} can be expressed as a sum of the gate to source leakage current i_{gs} and the gate to drain leakage current i_{gd} [25]. Thus, we can write,

$$P_{total} = I_{static} V_{DD} + \alpha C V_{DD}^2 f + \frac{1}{2} I_{shortcircuit} V_{DD} + V_{DD}(I_{gate} + I_{sub}). \tag{2.11}$$

We estimate

$$i_{gs} = \frac{127.04 \times L_{eff} \times e^{\left(5.60625\times\ V_{gs}^{-10.6} \times T_{ox}^{-2.5}\right)}}{2}, \tag{2.12}$$

$$i_{gd} = \frac{127.04 \times L_{eff} \times e^{\left(5.60625\times V_{gd}^{-10.6} \times T_{ox}^{-2.5}\right)}}{2} \tag{2.13}$$

following the commonly used gate-oxide leakage models [25] where L_{eff} is the effective gate width in nanometers, T_{ox} is the effective gate oxide thickness in nanometers and V is the effective supply voltage for the number of cores as designated in its subscript. The i_{gs} and i_{gd} are given in micro Ampere per micrometer of transistor width. Considering the above model for a single core processor, the estimate for power dissipation can be written as

$$P_1 = I_{static} V_1 + \alpha C V_1^2 f_1 + \frac{1}{2} I_{shortcircuit} V_1 + V_1(I_{gate} + I_{sub}). \tag{2.14}$$

For a CMP with n cores the power equation can be written as

$$P_n = n I_{static} V_n + n\alpha C V_n^2 f_n + n\frac{1}{2} I_{shortcircuit} V_n + n V_n(I_{gate} + I_{sub}). \tag{2.15}$$

For a multicore processor with n cores the power equation can be re-written in terms of voltage scaling:

$$P_n = n I_{static} \nu V_1 + n\alpha C \nu^2 V_1^2 f_n + n\frac{1}{2} I_{shortcircuit} \nu V_1 + n\nu V_1(I_{gate} + I_{sub}), \tag{2.16}$$

since $V_n = \nu V_1$, the scaled voltage for n cores with respect to a single core. As seen from the above model for CMPs the leakage power increases with increase of number of cores per chip. Hence apart from power actually consumed during computation (i. e., dynamic power) the leakage power plays an important role in on-chip parallel computing with a CMP.

From equation (2.14) we can write the frequency for a single core processor in terms of power being consumed,

$$\begin{aligned} f_1 &= \frac{P_1 - V_1(I_{static} + \frac{1}{2} I_{shortcircuit} + I_{gate} + I_{sub})}{\alpha C V_1^2} \qquad (2.17) \\ &=: \frac{P_1 - V_1 \tilde{I}}{\alpha C V_1^2}. \end{aligned}$$

From equation (2.15) we can derive the frequency for a CMP using n cores as,

$$\begin{aligned} f_n &= \frac{P_n - nV_n(I_{static} + \frac{1}{2}I_{shortcircuit} + I_{gate} + I_{sub})}{n\alpha C V_n^2} \\ &=: \frac{P_n - nV_n\tilde{I}}{n\alpha C V_n^2}. \end{aligned} \tag{2.18}$$

Also, $\frac{f_1}{f_n} = n\nu^2 \frac{P_1 - V_1\tilde{I}}{P_n - n\nu V_1\tilde{I}}$. Putting the expression for the frequencies in equation (2.4) we can express power aware speed-up as follows.

$$S = \frac{1}{(1-PF)\times OF + n\frac{PF\times OF}{DOP}\frac{CPI_n}{CPI_1}\nu^2\frac{P_1 - V_1\tilde{I}}{P_n - n\nu V_1\tilde{I}}}. \tag{2.19}$$

We define $\vartheta := \frac{P_1}{V_1\tilde{I}}$ and $\varsigma = \frac{P_n}{P_1}$. Then, the speed-up can be written as

$$S = \frac{1}{(1-PF)\times OF + n\frac{PF\times OF}{DOP}\frac{CPI_n}{CPI_1}\nu^2\frac{\vartheta - 1}{\vartheta\varsigma - n\nu}}. \tag{2.20}$$

Power aware parallel efficiency, $\epsilon(n)$, for a CMP using n cores can be similarly expressed plugging in the expressions for frequencies in equation (2.6):

$$\epsilon(n) = \frac{1}{n(1-PF)\times OF + n^2\frac{PF\times OF}{DOP}\frac{CPI_n}{CPI_1}\nu^2\frac{P_1 - V_1\tilde{I}}{P_n - n\nu V_1\tilde{I}}}. \tag{2.21}$$

The *power optimization* (P_{opt}) can be expressed in terms of the energy consumption

$$P_{opt} := \frac{\text{Energy consumed in serial code execution}}{\text{Energy consumed in parallel code execution}} \tag{2.22}$$

So, the term Power optimization can be defined as follows.

$$\begin{aligned} P_1 &:= \text{Total execution time for serial code} \times (\text{Static Power} + \text{Leakage Power}) \\ &\quad + \text{System time for serial code} \times \text{Dynamic Power} \\ P_n &:= \text{Total execution time for parallel code} \times (\text{Static Power} + \text{Leakage Power}) \\ &\quad + \text{System time for parallel code} \times \text{Dynamic Power for all } n \text{ cores} \\ P_{opt} &= \frac{P_1}{P_n} \end{aligned} \tag{2.23}$$

When one thread is waiting for another thread to complete some computation, the static, leakage power and the short-circuit power will be consumed even though very little dynamic power is being consumed. In order to optimize a parallel implementation with respect to power consumption it is necessary to design the numerical algorithms in such a way that threads wait for each other a minimum at the barriers. For this the computational load among different cores must be balanced with respect to localization of the threads and the barrier calls.

2.2. Cache Hierarchy Effects. For cache miss, there is associated penalty in terms of loss of parallel efficiency. If cache is hit, let t_h be the time taken for fetching the data from the cache. For a cache miss, let the time taken be t_m where $t_m \gg t_h$. If for any algorithm p is the fraction of computation that has scored a cache hit, then time taken, t_{cache}, in cache operation for fetching the data can be written as $t_{cache} = pt_h + (1-p)t_m$. Then cache penalty C_p can be expressed as

$$C_p = \frac{t_{cache}}{t_h} = \frac{t_m}{t_h} + p\left(1 - \frac{t_m}{t_h}\right), \tag{2.24}$$

where if t_h is in the order of nanoseconds, then t_m is in the order of microseconds, as during cache miss data must be fetched from memory and memory access is a relatively slower process. Actual cache operation time is expressed as,

$$t_{cache} = C_p t_h. \tag{2.25}$$

Depending on cache hierarchy, t_h and p may vary. Time taken on a hit at the cache at core level is less than the time taken on a hit at cache per chip region each covering a group of cores, which in turn will be less than the time taken on a hit at the CMP's global cache. Let time taken for a cache hit from cache at core level be $t_{h.core}$, time taken for a cache hit at a cache for a group of cores be $t_{h.zone}$, and time taken for a cache hit at the global cache for the CMP be $t_{h.chip}$. Generally it would be realistic to assume that $t_{h.core} \leq t_{h.zone} \leq t_{h.chip}$. For a particular application, let there be cache hit at cache per core level p_1 fraction of times, cache hit at the group of cores level be p_2 fraction of times and the cache hit at the CMP's global cache be p_3 fraction of times. Then time taken for cache hit may be expressed as

$$t_h = p_1 t_{h.core} + p_2 t_{h.zone} + p_3 t_{h.chip}, \tag{2.26}$$

where, $p = p_1 + p_2 + p_3$. The speed-up, S, may be modified and written as

$$S = \frac{1}{C_p t_h \frac{f_1}{w_s \times CPI_1} + (1 - PF) \times OF + \frac{PF \times OF}{DOP} \frac{CPI_n}{CPI_1} \frac{f_1}{f_n}}, \tag{2.27}$$

in which frequency expressions from equations (2.18) and (2.19) may be substituted.

Depending on the architecture one has to modify the conventional algorithms to reduce the cache miss rate. So, for any application running on multicore processor, the algorithm should be designed in such a fashion that the amount of data handled or processed by a particular core is optimized, i. e., data driven parallelism is a better option.

2.3. Communication Overheads. The communication overhead including interconnect propagation and switching latency, t_c may be considered and the speed-up in equation (2.27) is re-formulated as

$$\begin{aligned} S_p &= \frac{1}{(C_p t_h + t_c) \frac{f_1}{w_s \times CPI_1} + (1 - PF) \times OF + \frac{PF \times OF}{DOP} \frac{CPI_n}{CPI_1} \frac{f_1}{f_n}} \\ &= \frac{1}{(C_p t_h + t_c) \frac{f_1}{w_s \times CPI_1} + (1 - PF) \times OF + \frac{PF \times OF}{DOP} \frac{CPI_n}{CPI_1} n\nu^2 \frac{\vartheta - 1}{\vartheta\varsigma - n\nu}} \end{aligned} \tag{2.28}$$

giving the power-aware speed-up considering computation, communication and cache hierarchy.

3. Numerical Linear Algebra Routines. The most widely used tools for scientific computing are the Basic Linear Algebra Subprograms (BLAS) and the LAPACK libraries [2, 15]. In addition to tuning the individual numerical linear algebra routine's serial algorithm to a single core's architecture, for a CMP, considerations for the overall tuning of the multithreaded parallel numerical linear algebra operation to the on-chip communication, cache and power architecture are essential. The present paper uses POSIX threads to implement ScaLAPACK [10] algorithms for observing the intended power-aware speed-up. For tuning purposes, data manipulation and computation are done according to physical and logical access of a single processing element (core) or a group of cores on the chip.

3.1. Jacobi Iterative Solver. For a real system of equations, $Ax = b$, the Jacobi iteration is a method of simultaneous corrections, as no component of an approximation $x^{(m)}$ is used until all the components of $x^{(m)}$ have been computed. The method replaces mth iterate $x^{(m)}$ by $x^{(m+1)}$ at once, before the beginning of next cycle. The algorithm may be written as $x^{(m+1)} = b + (I - A)x^{(m)}$. In the thread parallel version of the Jacobi iterative solver, the workload is distributed over all the threads across the rows. Every thread is operates on a block of adjacent rows to get the facility of spatial locality of reference of data. The data-distribution is shown in Figure 3.1. Matrices with $A_{i,j} = N/(i + j + 1)$, N being the matrix column dimension and all diagonal elements set at 10000 is used for computational experiments. The right hand side, $b_i = i$ and initial guess $x_i^0 = 4.0i$ have been used.

3.2. Cholesky Factorization. Symmetric positive definite real square matrix A has Cholesky factorization $A = LL^T$, L being the lower triangular matrix with positive diagonal entries. The linear system can be then solved by forward substitution in lower triangular system $Ly = b$, followed by the back substitution in upper triangular system $L^T x = y$.

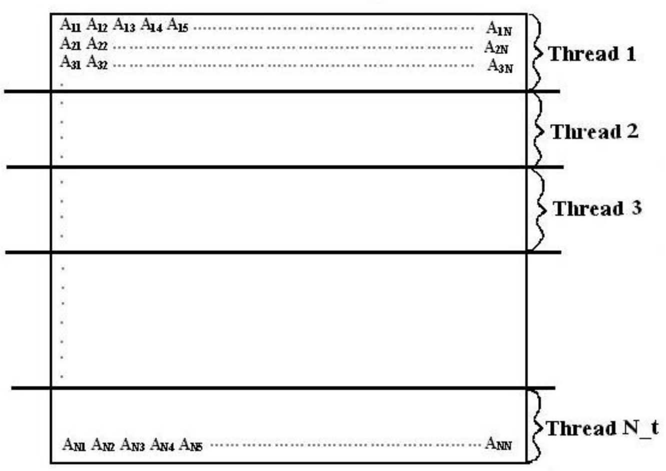
A_{11} A_{12} A_{13} A_{14} A_{15} ... A_{1N}
A_{21} A_{22} ... A_{2N}
A_{31} A_{32} ... A_{3N}
Thread 1
Thread 2
Thread 3
Thread N_t
A_{N1} A_{N2} A_{N3} A_{N4} A_{N5} ... A_{NN}

FIG. 3.1. *Threaded Parallel Jacobi Iterative Solver*

3.2.1. Threaded Parallel Algorithm. In the threaded parallel version of the Cholesky factorization the two basic operations, division (`cdiv`) and modifications (`cmod`), are parallelized. These two basic operations are based on the thread identity so that each thread will operate on some entries within a range of the matrix indices in a synchronized way. To achieve such synchronization blocking calls and thread schedulers available on POSIX thread library are used. We have used Round-Robin scheduling (`SCHED_RR`) for the threads setting minimum priority by using `sched_get_priority_min`(). Also, for Solaris®based systems the concurrencies can be set using `thr_setconcurrency`(). The threads were spawned using `pthread_create`() and `thread_create`() for Linux based and Solaris®based systems respectively. For synchronizing among threads `pthread_create`() (`thread_join`() for Sun Solaris®systems) was used. Scheduling parameters like the priority variable has been set by adding minimum priority and index to minimize thread waiting. By using `pthread_setschedparam`() and `pthread_getschedparam`() scheduling parameters, the Round-Robin Scheduling was configured. Data distribution has been done on the outer most loop in the block cyclic distributed sub-matrix format of Cholesky factorization. Each thread operates with in a range (like k_{start} to k_{end}) which depend on the thread identity or index. The range is proportional to the ratio of number of elements per row by the number of threads being spawned. With this, an attempt is made to keep computational load per thread almost equidistributed so that there is little skew in finishing time of the threads at a barrier. This in turn helps in meeting leakage power dissipation goals via synchronization objectives. The test cases used are symmetric positive definite matrices from [31] in dense format and the table 3.1 gives the list of test matrices used for computation. The schedulers add to the parallel overhead a little more (2-5 per cent) but avoids fine grained parallelization of the division (`cdiv`) and modifications (`cmod`) individually. The reduced granularity of parallelization improves the fraction to be parallelized.

4. Experiments and Results.

In the speed-up equation (2.28), the voltage scaling ν and power scaling ς are determined from the various throttle settings attained by the particular CMP design. We take activity factor $\alpha = 1$ and estimate ϑ, which depends on the architecture and CMOS process of the CMP, with the following tools. Using Cacti [34] we have estimated cache access power characteristic using the cache size, number of banks, (set)-associativity and size of blocks as input. But to obtain leakage power characteristics we have used eCACTI [29] setting the parameters for applicable CMOS (90 nanometer in our experiments) technology parameter along with all the inputs required in Cacti. For the leakage power estimates we have used parameters of the cache hierarchy model of the host CMP in a generalized simplified framework as shown in figure 4.1. By using `g-cache-trace` modules the cache-statistics was obtained. The `g-cache-trace` modules also trace the operating system's cache statistics along with the code execution statistics. About 2 to 5 per cent better cache hit statistics in case of parallel code execution as compared to serial code execution was observed for the matrix computation routines experimented with. The simulator LUNA [12, 13] is used to obtain high level power and delay analysis for a generalized one-to-one link topology among the individual cores. The simulation using LUNA are done while accepting its limitation that the network representation, bandwidth for each link

Table 3.1
Example Matrices from [31]

Matrix Name	Dimension and Non-Zero Entries	Functionality
BCSSTK01	48×48, 224	BCS Structural Engineering Matrices : Small test problem
BCSSTK02	66×66, 2211	BCS Structural Engineering Matrices : Oil rig - statically condensed
BCSSTK03	112×112, 376	BCS Structural Engineering Matrices : Small test structure
BCSSTK04	132×132, 1890	BCS Structural Engineering Matrices : Oil rig—not condensed
BCSSTK05	153×153, 1288	BCS Structural Engineering Matrices : Transmission tower
BCSSTK06	420×420, 4140	BCS Structural Engineering Matrices : Medium test problem
BCSSTK08	1074×1074, 7017	BCS Structural Engineering Matrices : TV studio
BCSSTK09	1083×1083, 9760	BCS Structural Engineering Matrices : Square plate clamped
BCSSTK11	1473×1473, 17857	BCS Structural Engineering Matrices : Ore car—lumped mass
BCSSTK13	2003×2003, 42943	BCS Structural Engineering Matrices : Fluid flow

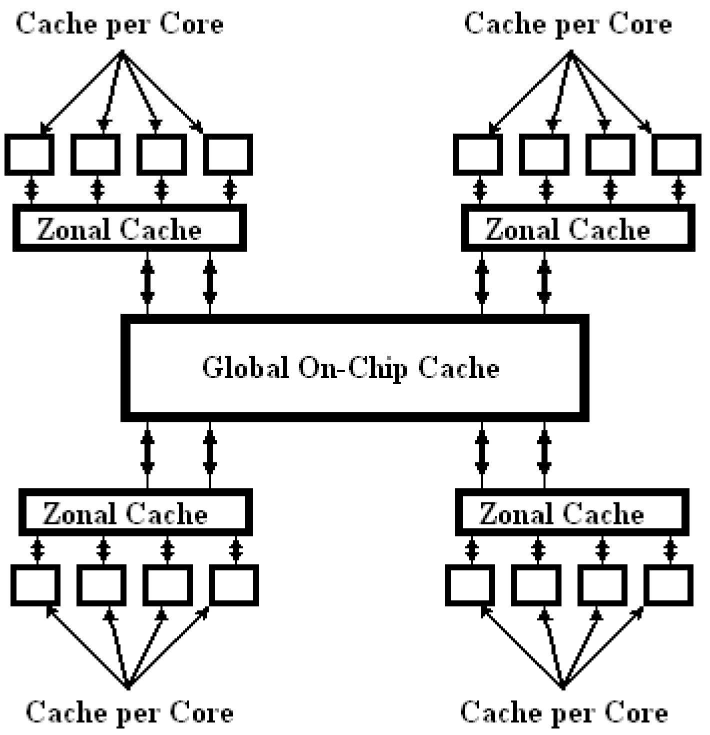

Fig. 4.1. *Cache hierarchy for a typical multicore system*

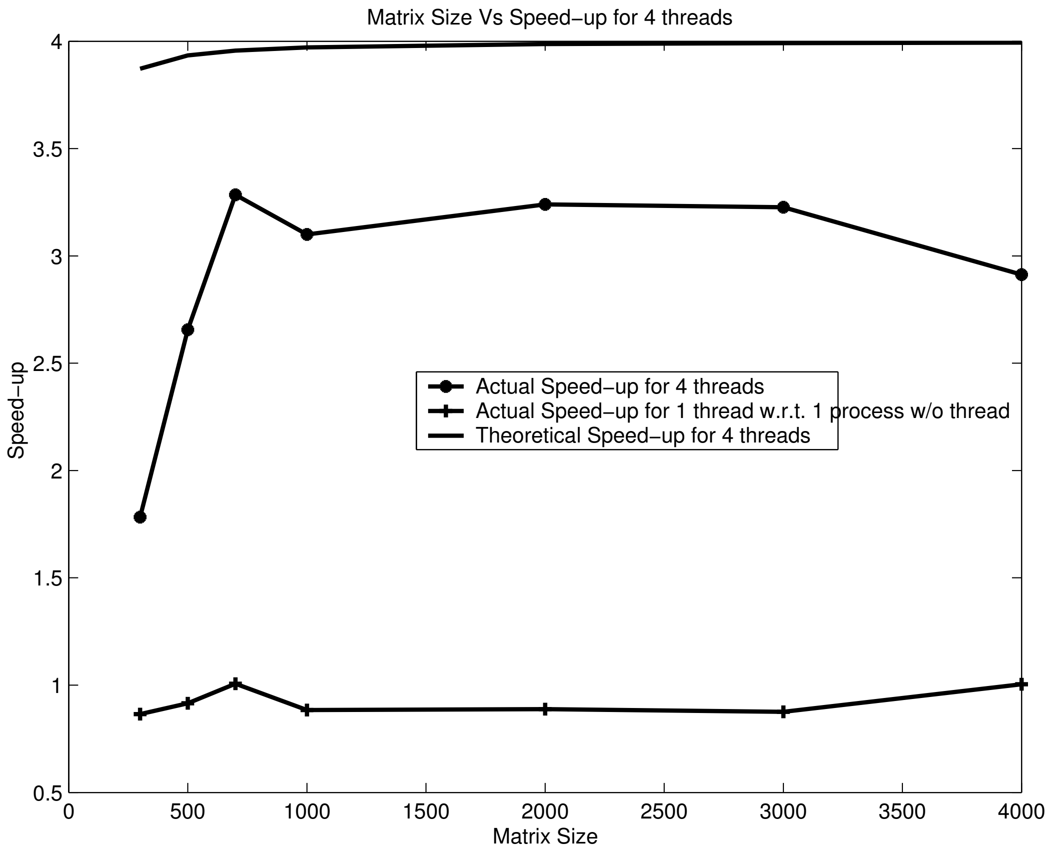

FIG. 4.2. *Speed-up for* 4 *threads for the Jacobi iterative solver*

and detailed data flow path may be very different in the actual CMP. Also, every application may not follow the same data flow path on the CMP every time. Thus LUNA gives only an idealized and statistically expected power behavior for the communication fabric.

4.1. Jacobi Iterative Solver. The speed-up increases nearly in linear fashion as the matrix size increases. In the parallel algorithm the amount of computation to be done in a single iteration has been distributed among different threads in such a manner so that all the threads are as much as possible equally loaded. There are some asymmetries in core performance in a CMP environment. But for our analysis leading to (2.28) we have neglected the performance asymmetry and take into account the delay due to blocking calls in our implementation of the multithreaded Jacobi iteration. These characteristics also implies that due to the unmodeled (in theoretical speed-up) overhead of parallelization in thread creation and blocking functions the actual speed-up is much less than the theoretical speed-up. The unmodeled overhead in thread-creation and blocking calls become relatively insignificant. The actual speed-up then approaches the theoretical speed-up for larger size of the matrix. The speed-up for one thread is less than 1 compared to the single process due to the thread overheads.

The actual execution time based speed-up for large matrices tends to follow the theoretical speed-up pattern for this example. The figure 4.2 shows the speed-up for 4 threads on a SUN Microsystem system based on a dual-core AMD Opteron®Model 2218 2.6GHz 90 nanometer SOI CMOS process CMP for the Jacobi iterative solver. The speed-up reaches more than 3 for square matrix with column dimension size greater than 600 using 4 threads.

The difference between the actual speed-up and theoretical speed-up is mostly due to synchronization overhead for different threads at the barrier calls and this overhead increases with the number of barrier calls as two threads are now localized on each core against the optimal situation of one per core. The theoretically unaccounted for skew due to the cores being busy with kernel threads also contribute to the shortfall with respect to the theoretical speed-up given by (2.28). However, we still get a good overall agreement with the theoretical model for this BLAS level 2 rich algorithm. The active component of power is directly related to the time for which the cores are busy while the static and leakage components are proportional to the total execution time.

4.2. Cholesky Factorization. The speed-up of threaded parallel Cholesky factorization in total execution time for 2 and 4 threads on a SUN system with dual AMD Opteron®dual-core Model 2218 2.6GHz 90 nanometer SOI CMOS process CMPs is studied in this section. One thread was localized per core. The data distribution was a equal sized row block partitioning among the threads. For Cholesky factorization the speed-up does not change much for input matrix sizes beyond the square matrix dimension of 1083. For the actual speed-up a similar slope in the speed-up curve to that of the theoretical curve as per equation (2.28) is observed. The speed-up curve both in theory and in our experiments may be seen to have a steep slope for square matrix with column dimensions between 70 and 450, and are more or less in agreement in terms of the expected profile of the experimental data. For matrices of size smaller than 70×70 the overhead due to parallelization and synchronization is relatively significant and makes multithreading not useful. For matrices larger than 1083×1083 cache misses play a more important role in affecting the actual speed-up. Thus a drop in actual speed-up is expected. The theoretical speed-up was derived assuming an infinite availability of cache and bus which obviously does not hold good for beyond a certain data size since the resources on a single chip multicore processor is finite and fixed unlike a multi-processor system where availability of resources can be more flexible. The simple cache hit-miss model becomes inadequate beyond this point of saturation and the actual speed-up falls far short of the theoretical speed-up. The figure 4.3 shows the speed-up of threaded Cholesky factorization for 2 and 4 threads on the same SUN Microsystems machine based on dual dual-core AMD Opteron®CMPs. The theoretical speed-up increases as the matrix size increases and reaches close to 1.9 and 3.7 for 2 and 4 threads respectively. We note that this is a BLAS level 3 rich algorithm unlike the Jacobi iterative solver and hence the saturation point for the multi-threaded algorithm is reached faster. For large number of threads we observe that the power characteristic is not greatly sensitive to the number of threads. This makes threaded Cholesky Factorization on a CMP an algorithm relatively less sensitive to power aware optimizations.

4.3. Remarks. All the power characteristics are based on the estimates made on the processor models with only public domain manufacturer data using eCACTI and Cacti. This is plugged into the theoretical speed-up and compared with the actual system time and total time lapsed for running the codes. Jacobi iterative solver is easier to tweak for optimal speed-up due to the fact that in each iteration all the values needed are already being computed in the last iteration. This gives a possibility of getting $\Theta(n)$ speed-up where n is the number of cores for a single user system executing only this parallel program. Thread-scheduling do not play a significant role in tuning this solver. For Cholesky factorization, data-parallel model involves a lot more complex constraints. Thread-scheduling plays a significant role in reducing the use of blocking calls and in turn, static and leakage power dissipation related slow-down. Thread-scheduling, as before, plays a significant role. The blocking calls, however, cannot be avoided and is necessary to maintain the data-consistency. This in turn decreases the speed-up. The amount of blocking increases the static power consumption and the leakage loss. The data-parallel model induces a relatively higher cache hit rate which helps in getting better performance with lower power consumption. For Cholesky Decomposition, tuning based on simplistic cache and communication model makes it difficult to get the predicted speed-up for larger matrices.

5. Conclusions. The threaded parallel numerical linear algebra routines is a natural approach to tuning the BLAS and LAPACK/ScaLAPACK routines to a CMP. Power, on-chip cache traffic and on-chip interconnect behavior play a crucial role in getting the speed-up which is desired to be close to the order of the number of threads. Due to the umodeled overheads in thread creation and synchronization, we do not get significant speed-up for smaller matrices. Again for very large problems, cache memory related latencies come into the picture and affects speed-up. A possibility is that a CMP with a large number of cores can be utilized to reduce the cache-misses by understanding the order of data access in the matrix computation. Algorithmic level optimization of the power may be possible by reducing the static and leakage dominated phases of the implementation. By utilizing the cache-hierarchy to the maximum possible extent the overall waiting time can be reduced and consequently the overall execution time as lower power is dissipated. Efficient thread schedulers must be embedded into the tuned implementations to utilize the cores as symmetrically as possible.

As general CMPs with large number of cores and switched on-chip communication fabric become available, it would be possible to observe the power aspects of the speed-up metric in greater detail.

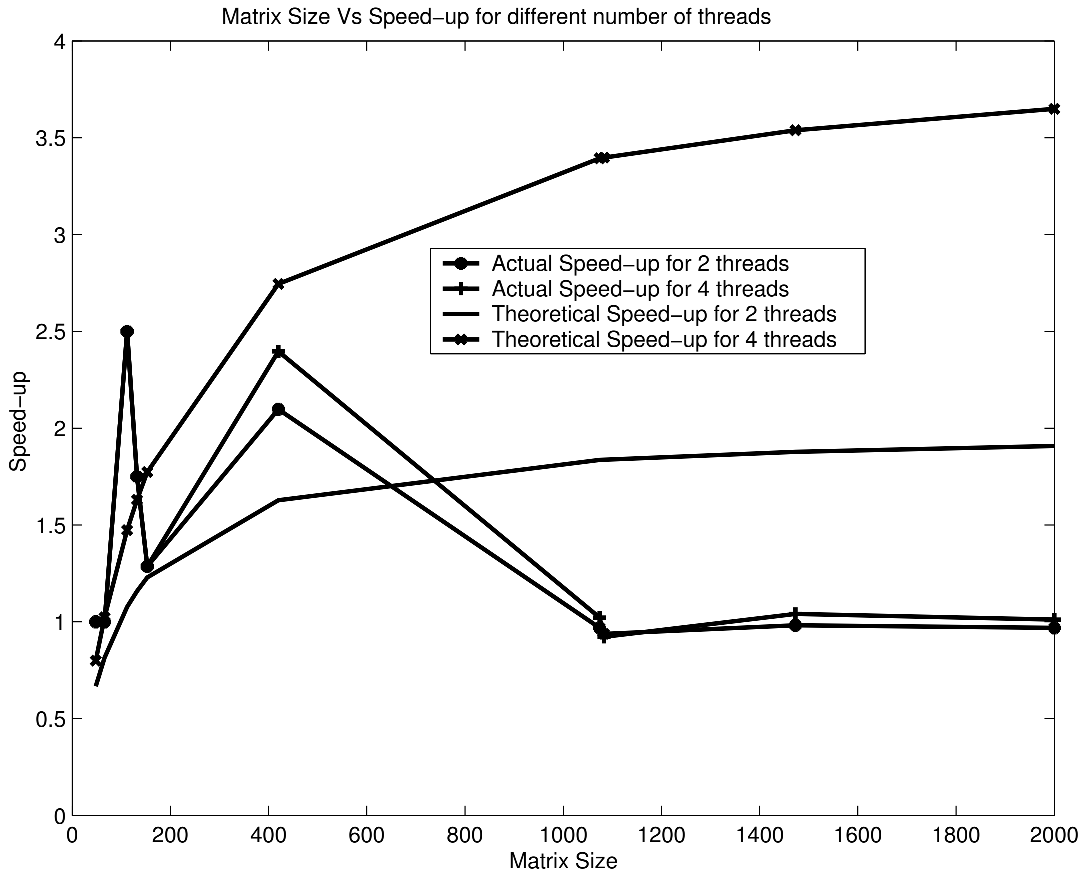

FIG. 4.3. *Speed-up for different thread numbers for the Cholesky factorization. It may be noted that our theoretical speed-up computation is in good agreement with the experimental data for dense matrix size of up to 450. For larger matrices the system becomes saturated and page faults and cache thrashing dominate. The theoretical model is based on the implicit assumption that infinite cache is available. Hence the disagreement beyond a certain matrix size. Unlike a multi-chip multi-processor system, a chip multicore processor displays this limitation more sharply. Also, for very small matrices the thread overheads dominate and both theoretical and actual speed-ups are less than* 1 *indicating that below a certain data size and work load it is not worth multithreading.*

REFERENCES

[1] EMRAH ACAR, RAVISHANKAR ARUNACHALAM, AND SANI R. NASSIF, *Predicting short circuit power from timing models*, in ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation, New York, NY, USA, 2003, ACM, pp. 277–282.

[2] E. ANDERSON, Z. BAI, C. BISCHOF, L. S. BLACKFORD, J. DEMMEL, JACK J. DONGARRA, J. DU CROZ, S. HAMMARLING, A. GREENBAUM, A. MCKENNEY, AND D. SORENSEN, *LAPACK Users' guide (third ed.)*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[3] C. D. ANTONOPOULOS, D. S. NIKOLOPOULOS, AND T. S. PAPATHEODOROU, *Informing algorithms for efficient scheduling of synchronizing threads on multiprogrammed SMPs*, in ICPP '02: Proceedings of the 2001 International Conference on Parallel Processing, Washington, DC, USA, 2001, IEEE Computer Society, pp. 123–130.

[4] SAISANTHOSH BALAKRISHNAN, RAVI RAJWAR, MIKE UPTON, AND KONRAD LAI, *The impact of performance asymmetry in emerging multicore architectures*, in ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture, Washington, DC, USA, 2005, IEEE Computer Society, pp. 506–517.

[5] TOBIAS BJERREGAARD AND SHANKAR MAHADEVAN, *A survey of research and practices of Network-on-chip*, ACM Comput. Surv., 38 (2006), p. 1.

[6] U. BONDHUGULA, M. BASKARAN, A. HARTONO, S. KRISHNAMOORTHY, J. RAMANUJAM, A. ROUNTEV, AND P. SADAYAPPAN, *Towards effective automatic parallelization for multicore systems*, Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, (2008), pp. 1–5.

[7] DAVID R. BUTENHOF, *Programming with POSIX Threads*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[8] ALFREDO BUTTARI, JULIEN LANGOU, JAKUB KURZAK, AND JACK DONGARRA, *A class of parallel tiled linear algebra algorithms for multicore architecture*, in MIMS Preprint, LAPACK Working Note no. 191, 2007.

[9] Xuning Chen and Li-Shiuan Peh, *Leakage power modeling and optimization in interconnection networks*, in ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design, New York, NY, USA, 2003, ACM, pp. 90–95.

[10] Jaeyoung Choi, Jack J. Dongarra, L. Susan Ostrouchov, Antoine P. Petitet, David W. Walker, and R. Clint Whaley, *Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines*, Sci. Program., 5 (1996), pp. 173–184.

[11] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[12] Noel Eisley and Li-Shiuan Peh, *High-level power analysis for on-chip networks*, in CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, New York, NY, USA, 2004, ACM, pp. 104–115.

[13] Noel Eisley, Vassos Soteriou, and Li-Shiuan Peh, *High-level power analysis for multi-core chips*, in CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, New York, NY, USA, 2006, ACM, pp. 389–400.

[14] C. Garcia, M. Prieto, and F. Tirado, *Multigrid smoothers on multicore architectures*, NIC Series Parallel Computing: Architectures, Algorithms and Applications, 38 (2007), pp. 270–286.

[15] G. H. Golub and C.F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore and London, third ed., 1996.

[16] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, *Introduction to Parallel Computing*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[17] Fred G. Gustavson, *The relevance of new data structure approaches for dense linear algebra in the new multi-core/many core environments*, in Exploiting Concurrency Efficiently and Correctly—(EC)2, CAV 2008 Workshop, 2008.

[18] Sven Hammarling, Nicholas J. Higham, and Craig Lucas, *LAPACK-Style codes for pivoted Cholesky and QR updating*, tech. report, MIMS EPrints United Kingdom, 2007.

[19] John L. Hennessy and David A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[20] M A Heroux, *Design issues for numerical libraries on scalable multicore architectures*, Journal of Physics: Conference Series, 125 (2008), p. 012035 (11pp).

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

[22] ITRS, *ITRS Report* , tech. report, The International Technology Roadmap for Semiconductors, 2006.

[23] I. Jonsson, *Recursive blocked algorithms, data structures, and high-performance software for solving linear systems and matrix equations*, tech. report, Academic Archive On-line [http://www.diva-portal.org/oai/OAI] (Sweden), 2003.

[24] Rakesh Kumar, Victor Zyuban, and D.M. Tullsen, *Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling*, in Proc. 32nd Annual International Symposium on Computer Architecture (ISCA'05), Los Alamitos, CA, USA, 2005, IEEE Computer Society, pp. 408–419.

[25] Dongwoo Lee, Wesley Kwong, David Blaauw, and Dennis Sylvester, *Analysis and minimization techniques for total leakage considering gate oxide leakage*, in DAC '03: Proceedings of the 40th conference on Design automation, New York, NY, USA, 2003, ACM, pp. 175–180.

[26] Jian Li and José F. Martínez, *Dynamic power-performance adaptation of parallel computation on chip multiprocessors*, High-Performance Computer Architecture, 2006. The Twelfth International Symposium on, (11-15 Feb. 2006), pp. 77–87.

[27] ———, *Power-performance considerations of parallel computing on chip multiprocessors*, ACM Trans. Archit. Code Optim., 2 (2005), pp. 397–422.

[28] Yingmin Li, K. Skadron, D. Brooks, and Zhigang Hu, *Performance, energy, and thermal considerations for* SMT *and* CMP *architectures*, High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on, (12-16 Feb. 2005), pp. 71–82.

[29] Mahesh Mamidipaka and Nikil Dutt, *eCACTI: An enhanced power estimation model for on-chip caches*, tech. report, University of California, Irvine, Irvine, CA 92697-3435, USA, Sept., 2004.

[30] M. Mamidipaka, Kamal Khouri, Nikil Dutt, and Magdy Abadir, *Leakage power estimation in SRAMs*, tech. report, CECS Technical report, TR 03-32, UC Irvine, Oct. 2003, 2003.

[31] NIST, *Matrix Market*, June 2004.

[32] V. Tiwari, S. Malik, and A. Wolfe, *Power analysis of embedded software: a first step towards software power minimization*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2 (1994), pp. 437–445.

[33] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee, *Instruction level power analysis and optimization of software*, in VLSID '96: Proceedings of the 9th International Conference on VLSI Design: VLSI in Mobile Communication, Washington, DC, USA, 1996, IEEE Computer Society, pp. 326–328.

[34] S.J.E. Wilton and N.P. Jouppi, *CACTI: an enhanced cache access and cycle time model*, Solid-State Circuits, IEEE Journal of, 31 (May 1996), pp. 677–688.

[35] Fabian Zabatta and Kevin Ying, *A Thread Performance Comparison: Windows NT and Solaris on a Symmetric Multiprocessor*, 1998.

Edited by: Dana Petcu
Received: November 6th, 2008
Accepted: March 9th, 2009