Scaling Performance of Interior-Point Method on Large-Scale Chip Multiprocessor System

Mikhail Smelyanskiy   Victor W Lee   Daehyun Kim   Anthony D Nguyen   Pradeep Dubey

Microprocessor Technology Labs, Intel

{mikhail.smelyanskiy, victor.w.lee, daehyun.kim, anthony.d.nguyen, pradeep.dubey}@intel.com

ABSTRACT

In this paper we describe parallelization of interior-point method (IPM) aimed at achieving high scalability on large-scale chip-multiprocessors (CMPs). IPM is an important computational technique used to solve optimization problems in many areas of science, engineering and finance. IPM spends most of its computation time in a few sparse linear algebra kernels. While each of these kernels contains a large amount of parallelism, sparse irregular datasets seen in many optimization problems make parallelism difficult to exploit. As a result, most researchers have shown only a relatively low scalability of 4X-12X on medium to large scale parallel machines.

This paper proposes and evaluates several algorithmic and hardware features to improve IPM parallel performance on large-scale CMPs. Through detailed simulations, we demonstrate how exploring multiple levels of parallelism with hardware support for low overhead task queues and parallel reduction enables IPM to achieve up to 48X parallel speedup on a 64-core CMP.

1. INTRODUCTION

Now commonplace, chip multiprocessors (CMPs) provide applications with an opportunity to achieve much higher performance than uniprocessor systems. Examples of CMPs are 8-core IBM CELL Broadband Engine [Gschwind06], 32-core Sun Niagara [Kongetira04], and Intel® Core™ Duo Processor [Gochman06]. Furthermore, as the number of cores on a CMP continues to grow, the performance of the CMP increases commensurately. This trend gives rise to two important questions: (i) how to expose an adequate amount of parallelism to the underlying CMP hardware within a given application, and (ii) which hardware features help CMP platform fully explore this parallelism in order to deliver highly scalable performance.

In this paper we address these two questions in the context of interior-point method (IPM). IPM is an important computational technique that solves optimization problems [Bixby02]. Optimization refers to the minimization (or maximization) of an objective function of several decision variables which satisfy

some constraints [Nocedal06]. Examples of optimization include the optimal choice of a portfolio of stocks, given a budget and diversity requirements, or the optimal design of a truss, given certain load requirements. In our work we focus on linear optimization problems where both the objective function and the constraints are linear. Our findings are directly applicable to non-linear problems, which use a similar set of computational kernels as linear optimization.

IPM spends a majority of its time in a direct sparse linear solver. Parallelism within the sparse solver exists on several levels: coarse-grain and fine-grain. While coarse-grain parallelism suffices to achieve good scalability on a small number of CMP cores, effective utilization of a large number of cores requires exploiting multiple levels of parallelism. This is achieved by partitioning the problem into many tasks and dynamically scheduling these tasks among cores. Such partitioning in general allows for much better load balance among cores, but may result in many small tasks. Consequently, dynamic scheduling schemes, oftentimes implemented using task queues programming model, suffer from high overhead of scheduling small tasks. Scheduling overhead can greatly reduce parallel scalability on many cores. In this work, we evaluate the impact of hardware support for low overhead task queues on IPM. We show how such support enables good load balance while significantly reduces overhead of task scheduling.

Another important source of performance degradation in IPM comes from parallel reduction, where multiple cores update the same memory locations. To guarantee atomicity of such updates, existing schemes use fine-grain synchronization to serialize their execution. Similar to task scheduling overhead, serialization overhead can severely degrade parallel scalability on many cores. We propose and evaluate parallel reduction hardware, which significantly decreases the overhead of reduction.

Hardware support for low overhead task queues and parallel reduction enables CMP to efficiently explore large amount of parallelism available in IPM and achieve high scalable performance on many cores. We have implemented these two techniques in our cycle-accurate CMP simulator, and analyzed the performance of the fully parallelized IPM application [Koka04] on a diverse suite of datasets from NETLIB collection [Gay88]. As a result IPM achieves up to 48X speedup (43X on average) with respect to the serial code performance on a 64-core CMP platform.

This paper is organized as follows. Section 2 introduces IPM and its main computational kernels. Sections 3 and 4 provide a brief introduction to sparse linear solver and its parallelization as well as motivate hardware support for low-overhead task queues and parallel reduction. Section 5 discusses a particular implementation of such hardware support. Section 6 presents

1. Compute $\mathbf{r}_p = \mathbf{b} - \mathbf{A}\mathbf{x}_k$ and $\mathbf{r}_d = \mathbf{c} - \mathbf{z} - \mathbf{A}^T\mathbf{y}_k$
2. Check for convergence, using the norms of $\mathbf{r}_p$ and $\mathbf{r}_d$
3. Form $\mathbf{M} = \mathbf{AQA}^T$, where $\mathbf{Q} = \mathbf{XZ}^{-1}$ is a diagonal matrix
4. Compute Cholesky factor $\mathbf{M} = \mathbf{LDL}^T$, where $\mathbf{L}$ is lower triangular
5. Compute the predictor directions, $\mathbf{d}_p = (\mathbf{d}_{px}, \mathbf{d}_{py}, \mathbf{d}_{pz})$
   5.1. $\mathbf{d}_{py} = \mathbf{M}^{-1}[\mathbf{r}_p + \mathbf{AQ}(\mathbf{r}_p - \mathbf{XZe})]$
   5.2. $\mathbf{d}_{px} = \mathbf{Q}[\mathbf{A}^T\mathbf{d}_{py} + \mathbf{XZe} - \mathbf{r}_d]$
   5.3. $\mathbf{d}_{pz} = -\mathbf{Ze} - \mathbf{Q}^{-1}\mathbf{d}_{px}$
6. Do a ratio test to compute $\mathbf{a}_p$ and $\mathbf{a}_d$, by computing
   6.1. $\mathbf{a}_p = \min\{-x_j / D_{xj}: D_{xj} < 0 \text{ and } j = 1 .. n\}$
   6.2. $\mathbf{a}_d = \min\{-z_j / D_{zj}: D_{zj} < 0 \text{ and } j = 1 .. n\}$
7. Compute the barrier "parameter" u based on (x_k, y_k, z_k), $\mathbf{a}_p$ and $\mathbf{a}_d$
8. Compute the search direction $\mathbf{d}_s = (\mathbf{d}_{sx}, \mathbf{d}_{sy}, \mathbf{d}_{sz})$
   8.1. $\mathbf{d}_{sy} = \mathbf{M}^{-1}[\mathbf{r}_p + \mathbf{AQ}(\mathbf{r}_d + \mathbf{ue} - \mathbf{XZe} - \mathbf{D}_{px}\mathbf{D}_{pz}\mathbf{e})]$
   8.2. $\mathbf{d}_{sx} = \mathbf{Q}[\mathbf{A}^T\mathbf{d}_y - \mathbf{ue} + \mathbf{XZe} + \mathbf{D}_{px}\mathbf{D}_{pz}\mathbf{e} - \mathbf{r}_d]$
   8.3. $\mathbf{d}_{sz} = \mathbf{uX}^{-1}\mathbf{e} - \mathbf{Ze} - \mathbf{Q}^{-1}\mathbf{D}_{sx}\mathbf{D}_{sx}\mathbf{e} - \mathbf{Q}^{-1}\mathbf{d}_{sx}$
9. Do a ratio test to compute $\mathbf{a}_p$ and $\mathbf{a}_d$
   9.1. $\mathbf{a}_p = p \min\{-x_j / D_{xj}: D_{xj} < 0 \text{ and } j=1..n\}$, where $p = 0.99995$
   9.2. $\mathbf{a}_d = p \min\{-z_j / D_{zj}: D_{zj} < 0 \text{ and } j=1..n\}$, where $p = 0.99995$
10. Update the iterate as
   10.1. $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{a}_p\mathbf{d}_{sx}$
   10.2. $\mathbf{y}_{k+1} = \mathbf{y}_k + \mathbf{a}_d\mathbf{d}_{sy}$
   10.3. $\mathbf{z}_{k+1} = \mathbf{z}_k + \mathbf{a}_d\mathbf{d}_{sz}$

Figure 1: A single iteration of primal-dual IPM

performance and scalability analysis of IPM on our simulator. Finally, we discuss related work and conclude.

2. MAIN COMPUTATIONAL KERNELS OF IPM

In the past decade, the IPM has become a method of choice for solving large linear optimization problems of the form:

$$\min c^T x, \text{ subject to } Ax=b, x \geq 0$$

Here the vector $x = (x_1,...,x_n)$ is the optimization decision variable of the problem, the function $c^T x$ is the objective function, A is an m by n matrix of linear constraints, and vectors x, c, and b have appropriate dimensions. A vector $x*$ is called an optimal solution of the optimization problem if it has the smallest objective value among all vectors that satisfy the constraints.

Figure 1 outlines the kth iteration of the main optimization loop of IPM [Lustig96]. The method starts with an initial approximation to the solution of the optimization problem, x. The core of the method is the main optimization loop, which updates the vector x at each iteration until the convergence to the optimal solution vector $x*$ is achieved.

As figure shows, IPM spends most of its computation time in a small number of linear algebra kernels. Optimizing and parallelizing these kernels is key to efficient implementation of IPM. These are the most important kernels:

1. Formulation of linear systems of equations, $Mx=b$, where M is the symmetric matrix of the form $M = A Z^{-1}A$, where A is the original matrix of constraints. This requires a **matrix-matrix multiplication** operation.

2. **Cholesky factorization** of matrix $M = L L^T$ in order to solve the system of linear equations, $Mx=b$. Here L is the lower triangular, and L^T is its transpose. This step is normally the most time-consuming step of the IPM.

3. **Triangular solver** uses the result of factorization to solve a system of linear equations $(L L^T)x=b$, using the following three steps

Table 1: IPM execution time breakdown for linear programming problems

Kernel	ken-18	mod2	pds-10	watson	world	average
Mmm	44.5%	4.7%	2.7%	7.1%	6.6%	13.1%
Cholesky	30.8%	59.3%	67.7%	37.6%	63.5%	51.8%
forward solver	7.0%	11.6%	8.8%	16.4%	9.1%	10.6%
backward solver	9.7%	11.6%	9.4%	20.7%	10.4%	12.4%
MVM	2.5%	5.3%	3.6%	7.2%	4.2%	4.5%
BLAS1	5.5%	7.5%	7.8%	11.0%	6.2%	7.6%

   a.   Forward solver solves $Ly=b$.

   b.   Backward solver solves $L^Tx=z$.

4. **Matrix-vector multiplication (MVM)** computes Ax and A^Tx for different vectors x.

5. **Basic Linear Algebra Subroutines (BLAS1)** performs inner products, vector additions, vector norm and ratio test computation.

Table 1 shows the execution time breakdown of IPM on a single core, for a number of sparse linear programming problems (see Section 6.2 for detailed description of the datasets used in this analysis). We observe that IPM spends large proportion of time in sparse linear solver (kernels 2 and 3). However, good parallel performance of IPM requires an efficient parallel implementation of all kernels. For example, while IPM spends on average only 7.6% of its execution time in BLAS1, leaving this kernel unparallelized limits IPM speedup to 13X regardless of how well the rest of the kernels scale. Therefore in our implementation we parallelize all IPM kernels.
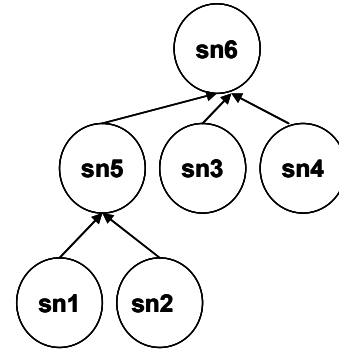
3. INTRODUCTION TO SPARSE LINEAR SOLVER

In this section we introduce the general framework of sparse linear solvers that follows block Cholesky approach [Ng93]. Two fundamental concepts behind solving sparse systems of linear equations are *super-node* and *elimination tree*. Both are defined in respect to the factor matrix L whose non-zero structure is computed prior to factorization.

A super-node is a set of contiguous columns in L whose non-zero structure consists of a dense triangular block on the diagonal and an identical set of non-zeroes for each column below the diagonal. An elimination tree is task dependence graph that characterizes the computation and data flow among the super-nodes of L during Cholesky factorization and triangular solver. In the elimination tree the parent of super-node j is determined by the first sub-diagonal non-zero in super-node i. Figure 2(a) shows an example of the factor matrix L, its non-zero elements (represented with 'X's) and 6 super-nodes (*sn1* through *sn6*). Figure 2(b) shows the corresponding elimination tree. There is an edge between *sn1* and *sn5*, because, as (a) shows (in shade), the second row of the 2 by 2 diagonal block of *sn5* depends on non-zero row 10 in *sn1*.

Given an elimination tree (ET), each of the three steps of linear solver can be expressed using the following generic formulation:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	x												
2	x	x											
3			x										
4			x	x									
5					x								
6					x	x							
7							x						
8							x	x					
9			x	x					x				
10	x	x							x	x			
11					x	x	x	x			x		
12					x	x					x	x	
13	x	x	x	x					x	x	x	x	x
	sn1		sn2		sn3		sn4		sn5		sn6		

(a) Factor matrix with super-nodes

(b) Elimination Tree

Figure 2: A triangular matrix with 6 super-nodes and the corresponding elimination tree.

> *T = breadth-first traversal of ET (bottom-up or top-down)*
> *for each super-node sn_i in T*
>   *perform processing task on sn_i*
> *endfor*

The super-nodes are processed in breadth first order: bottom-up for Cholesky and backward solver, top-down for forward solver. Super-nodes are stored using dense matrix representation. As described in the following section, the processing tasks involve various dense matrix operations.

4. PARALLELIZATION OF SPARSE LINEAR ALGEBRA KERNELS OF IPM

In this section we describe sparse linear algebra kernels in more detail as well as show how to expose and exploit parallelism within each of the kernels.

4.1 Cholesky Factorization

Figure 3 shows pseudo-code for Cholesky factorization. A Cholesky processing task (Lines 3-6) is generally expressed in terms of two primitive operations, *cdiv* and *cmod update*. The cdiv(*sn1*) operation (Line 3) multiplies the dense rectangular portion of the super-node *sn1* below its main diagonal by the inverse of the *sn1*'s dense diagonal block. cmod(*sn2, sn1*) update multiplies *sn1* with the transpose of its sub-matrix *C*, which corresponds to the dense triangular block of the *sn2*; it then scatter-adds the result of multiplication into destination super-node *sn2*. Such scatter-add operation is also known as the *reduction* operation.

> 1. *T = breadth-first bottom-up traversal of ET*
> 2. *for each super-node sn_a in T*
> 3.   **$cdiv(sn_a)$**
> 4.   *for each descendant super-node sn_d that must be updated by sn_a*
> 5.   **$cmod(sn_d , sn_a)$**
> 6.   *endfor*
> 7. *endfor*

Figure 3: Cholesky factorization pseudo-code.

The main loop in Figure 3 performs breadth-first bottom-up traversal of the super-nodes in the elimination tree starting from the leaves (Line 2). Each ancestor super-node performs cmod update on its decedents in the tree (Lines 4-6). By the time a super-node is traversed, it has collected all updates from its ancestors. At this point cdiv operation is performed to complete its own factorization (Line 3).

Parallelism in Cholesky factorization exists on three different levels:

Level-1: Coarse-grain parallelism exists within the elimination tree, where two or more super-nodes belonging to independent sub-trees of the elimination tree are traversed in parallel on different cores. Note, however, that two or more ancestors may simultaneously update the same descendent super-node of the elimination tree. When simultaneous updates occur to the same memory location, the result may be incorrect. To avoid this situation requires a mechanism to guarantee atomicity of such updates.

Level-2: The second level of parallelism exists in the innermost loop (Lines 4-6). Each ancestor super-node sn_a can simultaneously cmod update its descendents (Line 5). The second level of parallelism also exists within the two steps of cdiv operation, where independent dense triangular systems are solved for different right-hand sides.

Level-3: Fine-grain parallelism exists within individual cmod updates (Lines 5). Exploiting this parallelism amounts to

parallelization of the dense matrix-matrix product as well as scatter-add operations.

4.2 Triangular Solvers

Cholesky factorization results in a lower-triangular factor matrix L, such that $M=LL^T$. The solution to the system of equations $Mx=b$ is then obtained by first performing the forward solver $Ly=b$ and second the backward solver $L^Tx=y$. Similar to Cholesky computation, forward and backward solvers are made more efficient by taking advantage of the super-nodal structure of the factor matrix L.

The high-level pseudo code of forward solver is shown in Figure 4. Forward solver traverses the elimination tree from top to bottom. For each super-node sn_i, it performs a forward solve of a dense lower triangular system involving dense triangular block of sn_i, $L_{sni,sni}$ (Line 4), multiplies the solution vector by the dense rectangular portion of the sn_i, $L*,sni$ (Line 5), and scatter-adds the result of multiplication into the solution vector y (Line 6) taking non-zero structure of sn_i into account.

---

1. $y=b$
2. T = breadth-first top-bottom traversal of ET
3. for each super-node sni in T
4. $y_{sni} = [L_{sni,sni}]^{-1} y_{sni}$
5. $ty = L*,_{sni} y_{sni}$
6. $y = y - ty$
7. end for

Figure 4: Forward solver pseudo-code.

Backward solver is similar to forward solver, except that elimination tree is traversed from bottom to top and no scatter-add reduction is required. Both sparse forward and backward solvers contain parallelism on two levels:

Level-1: Similar to Cholesky, coarse-grain parallelism in the triangular solver exists among the super-nodes from independent sub-trees of the elimination tree. Performing scatter-add into the solution vector y (Line 6) may also result in simultaneous updates to the same elements of y, unless there exists mechanism to ensure atomicity of such updates. In backward solver the solution vector updates (Line 6) issued from parallel super-nodes are applied to disjoint elements of the solution vector and thus require no atomicity.

Level-2: Fine-grain parallelism exists within matrix-vector multiply, scatter-add reduction operation, and dense forward/backward solver of the dense diagonal block of the super-node.

4.3 Parallelization using Task Queue Model

A simple parallel implementation of Cholesky factorization and triangular solver only exploits coarse grain Level-1 parallelism. However, we observe that parallelism varies across elimination tree. At the bottom of the tree there are many small independent super-nodes with large amounts of coarse-grain parallelism. At the top of the tree there are few large super-nodes with small amounts of coarse-grain but large amounts of fine-grain parallelism.
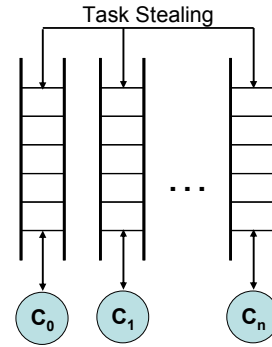

Figure 5: Distributed task queues

Hence only exploiting coarse grain parallelism results in limited parallel scalability due to load imbalance, as there is insufficient amount of work for each core to be fully utilized. Therefore to achieve high scalable performance of IPM, our implementation exploits all existing levels of parallelism: Level-1, Level-2 and Level-3 in Cholesky factorization, and Level-1 and Level-2 in forward and backward solvers. To accomplish this, we use a completely dynamic partitioning approach together with a task queue model of parallel programming.

Dynamic partitioning approach divides a task at each level of parallelism into smaller sub-tasks at a lower level, where the size of each sub-task is dynamically determined based on the size of the original task. Thus if the size of the original task is small, no subdivision is done. If the original task is large, it is partitioned into sub-tasks to keep system fully utilized. In forward solver, for example, the bottom of the tree has many small tasks at Level-1; hence no further partitioning is done. In the middle of the tree the number of Level-1 tasks decreases, while their size increases. As a result, our implementation starts partitioning these tasks into smaller Level-2 sub-tasks. At the top of the tree, where only a few large super-nodes are left, the Level-2 tasks dominate. Dynamic partitioning exposes larger amount of parallelism while at the same time reduces task size variability. This leads to improved load balance and better scalability on many cores.

To take advantage of dynamic partitioning and to further improve load balance, we use a task queue runtime system, which is responsible for en-queuing, de-queuing and scheduling tasks on a set of persistent threads. To this end it uses a popular mechanism called *distributed task queues* with *task stealing*, which is shown in Figure 5. In this scheme, each thread has its own local queue on which it primarily operates. When a thread en-queues a task from the partition, it places it in its local queue. When it finishes executing a task and needs a new task to execute, it first looks in its local queue. If there are no tasks available in its own queue, it *steals* a task from one of the other queues. Note that each of the queues is shared and needs to be protected by locks. Task stealing assures that all cores are utilized as long as there are available tasks.

Overall, dynamic partitioning and task queuing are the key to efficient parallel implementation of sparse linear solver.

4.4 Matrix-Matrix Multiplication and Other Kernels

While sparse linear solver is an essential part of IPM, it is important to parallelize the remaining kernels to achieve high scalability on many cores. We use well-known data partitioning schemes which divide these kernels into independent tasks.

Matrix-matrix multiplication is highly parallel, because each element of matrix M is computed independently as a dot product of the corresponding rows of A. Therefore, each task is comprised of a contiguous sub-set of non-zero elements of M.

To achieve good scalability of matrix-vector multiplication $y=Ax$, each task works with a contiguous subset of non-zero elements of A. As a result, the task may contain one or more rows of A. The task performs the dot product of non-zero elements of each such row i of A with the corresponding elements of x, and stores the result into $y[i]$. Since two or more tasks may contain elements of the same row of A, multiple threads may simultaneously update the same element of y. We guard each update to y with a fine-grain lock, assigning one lock per cache line, to guarantee atomicity of such updates. For our datasets the number of such simultaneous updates is small compared to the rest of the computation, therefore the overhead of fine-grain locking is also small.

5. EXPLOITING PARALLELISM IN IPM WITH HARDWARE SUPPORT

The previous section describes how to expose an abundant amount of parallelism in the sparse linear solver kernel of IPM. This section describes hardware support for low overhead task queues and parallel reduction to efficiently explore this parallelism on many CMP cores.

5.1 Low Overhead Task Queues

To efficiently utilize a large number of cores, a dynamic partitioning scheme creates a large number of parallel tasks. As a result, there are many small tasks, especially toward the top of the tree where only fine-grain parallelism prevails.

When task sizes are small, the overhead involved in software implementation of task queues can significantly degrade scalability of an application on a large number of cores [Kumar2007]. The overhead is due to several factors. The most significant factor is contention overhead, when multiple threads simultaneously access the same queue to steal the tasks. Such situations are common in sparse linear solver. For example, as discussed in Section 4, a large super-node at the top of the tree is partitioned by a given thread into many smaller sub-tasks, which get en-queued into its local queue. Other threads attempt to simultaneously de-queue these tasks from the queue. This creates contention over the shared queue and effectively serializes execution, as each thread has to wait until the previous thread atomically de-queued the task.

Another significant factor is the instruction overhead of managing the queue during task en-queuing and de-queuing operations. This overhead arises from grabbing and releasing the lock that protects the queue from simultaneous accesses by multiple threads, as well as incrementing (or decrementing) queue head pointer to point to the next empty position in the queue.
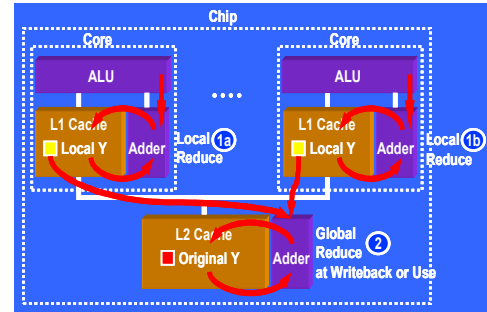

Figure 6: Parallel reduction hardware

In this work we use low overhead task queues, proposed in [Kumar2007], which accelerate task scheduling on CMP by implementing en-queuing and de-queuing operations in hardware. More specifically, in this proposal the tasks are stored in hardware queues, and are prefetched to the cores so that each core can start a new task as soon as it finishes its current one. This results in significant reduction of task scheduling overhead and improved scalability on CMP system.

5.2 Parallel Reduction Hardware

Scatter-add reduction operation is a common operation in many linear algebra kernels. As shown in Sections 4.1 and 4.2, Cholesky and forward solver used in IPM both require reduction. In general, reduction involves combining a set of data values into a single value. Implementing high performance reduction in parallel is challenging as it requires a mechanism to prevent data corruption caused by simultaneous updates from different cores. At the software level, synchronization is required to serialize updates. At the hardware level, the cache lines containing the reduction target need to be migrated from one core's cache to another before update can happen. While necessary, these forms of data corruption prevention mechanisms introduce significant amount of overhead and often cause performance degradation.

There are several software approaches to reduce impact of serialization in parallel reduction. Some approaches use fine-grain locks, each of which guard one or several cache lines. Other approaches rely on some form of hardware fetch-and-add mechanism for atomically updating a memory location. However, both approaches suffer from cache line migration during multiple concurrent updates to this cache line. Alternatively, one can privatize shared data, so that each thread operates on its local copy. Global reduction of all private copies is performed at the end of the parallel region. In both Cholesky and forward solver, parallel reduction involves updates to only a few non-contiguous locations of the large data structure. As a result, global reduction of the entire data structure results in substantial space and computation overhead.

Hardware support can substantially reduce the overhead in parallel reduction. Many supercomputers, such as the Connection Machine CM5 [Leiserson96] and IBM BlueGene [BlueGene02], provide a dedicated network for reduction operations. Our work takes a different approach that modifies the cache coherence protocol to simultaneously maintain multiple modified copies of a cache line for reduction. While this approach is similar to the solution proposed by [Kim03] and [Garzaran01] in the context of distributed shared memory multi-processors, we extend its

Table 2: Simulation Parameters					
Core Parameters		Memory Hierarchy Parameters		Contentionless Memory Latencies	
# of Cores	1 to 64	Private (L1) Cache	32kB, 2-way, 64B line	L1 hit	3 cycles
Architecture	2-issue, in-order core	Shared L2 cache	8MB, 16-bank, 8-way/bank, 64B line	L2 hit	18 - 58 cycles
Functional Units	2 Int ALU, 1 FPU; 1 Mul/Div, 2-port LSU	Interconnect	Bi-directional ring	Main Memory	298 - 338 cycles
Branch Pred.	G-share, 2k entries				

implementation to CMP environments, as shown in Figure 6. In our scheme, the cache lines which hold the reduction target are marked non-coherent and each core participating in the reduction operation is allowed to have a modified copy of the cache line while computing the partial reduced value. For example, in case of forward solver the solution vector y is marked as the reduction target. At the start of the reduction operation, the L1 cache controller sends a special request to the home directory controller to initiate the reduction operation. The L1 cache controller also allocates a cache line for temporary storage. This line will be initialized with reduction neutral value (e.g. zero for addition) and the core immediately performs reduction operation. Note this is different from software-based implementation where the core blocks while waiting to receive a single globally shared copy of the line to perform the reduction. When the first reduction request arrives to the home directory, the coherence protocol invalidates the line from all other L1 caches, transitions the line into a special reduction state, and finally adds the requesting core to the list of owners of the line. For subsequent reduction requests from other cores participating in the reduction the protocol only adds a new participant to the owner lists. When the reduction target is accessed (via read, read-for-ownership or eviction request), the home directory controller collects all non-coherent lines at the cores to which they belong and computes global reduction value before forwarding it to the consumer.

Note that to compute global reduction value, the directory controller has to be enhanced with execution units that support the required reduction operations. Due to the fact that all elements of a line can be processed in parallel or in a pipelined fashion the performance of parallel reduction can be further improved by pipelining these execution units or adding more units.

Overall, parallel reduction hardware described above significantly reduces amount of contention when multiple cores update data elements in the same cache line. Moreover, the scheme alleviates overhead of cache line migration as well as overhead of arithmetic operation required to perform reduction operation. This results in significant improvement in parallel scalability and performance on large-scale CMP system.

6. EXPERIMENTAL RESULTS

6.1 System Modeled

We use a cycle-accurate, execution-driven CMP simulator for our experiments. This simulator has been validated against real systems and is used extensively in our lab. Table 2 summarizes our base system configuration.

We model a CMP where each core is in-order, has a private L1 data cache, and all cores share an L2 cache. Each L1 cache has a hardware stride pre-fetcher. The cores are connected with a bi-directional ring, and the L2 cache is broken into multiple banks and distributed around the ring. A given cache line can exist in only one L2 bank according to an address hashing function (XORs the most significant bits with the least significant). Inclusion is enforced between the L1s and L2. Coherence between the L1s is maintained via a directory-based MSI protocol. Each L2 cache line also holds the directory information for that line. The ring has 41 stops, each of which can have two components connected to it (i.e., core or L2 cache bank). Our system models an aggregated memory bandwidth of 160GB/s which is similar to 16 channels of DDR3-1333. Finally, our simulator models aggregate ring bandwidth of 512GB/s.

For experiments which involve hardware support for low overhead task queues and parallel reduction, we add hardware as described in Sections 5.1 and 5.2 to the system.

6.2 IPM Implementation and Datasets

Our interior-point method is based on PCx - a serial interior-point predictor-corrector linear programming package [Czyzyk96]. Our parallel implementation of the Cholesky factorization and the solver routines uses as a baseline sparse direct solver package, called PARDISO [Schenk00], which is part of Intel's Math Kernel Library [MKL07]. Our baseline implementation uses a highly tuned version of software task queues [Kumar06] for load balancing, as well as fine-grain locking, with one lock per cache line, for parallel reduction.

Table 3 summarizes the statistics for the datasets used in our experiments. They come from the standard NETLIB test set [Gay88] and represent linear programming models from several application domains.

Column 1 lists the name of the datasets. Column 2 shows the number of constraints in the linear programming problem, which corresponds to the number of rows of matrix A. Column 3 shows the number of decision variables in the problem, which corresponds to the number of columns of A. Our problems range from medium size problems with tens of thousands constraints and variables to large size problems with hundreds of thousands constraints and variables. Column 4 shows the number of non-zeros in the matrix $M = A Z^{-1} A$. Column 5 shows the density of M, which shows that our datasets are very sparse. Column 6 shows the number of non-zeros in the L factor of M. Compared with original matrix, the factor matrix has substantially more non-zero elements than original matrix, due to fill-ins. The amount of fill-in

Table 3 : Dataset statistics

Dataset	LP Rows	LP Columns	M nnz	M Density (%)	L nnz
mod2	28761	56348	219039	0.0265%	1454339
ken-18	105127	154699	291082	0.0026%	2175306
pds-10	16558	49932	79866	0.0291%	1180995
watson	209614	411177	1263788	0.0029%	3776935
world	28653	58028	211001	0.0257%	1327756

can be reduced with re-ordering techniques. Our algorithm uses Minimal Degree Reordering from METIS [Karypis98].

6.3 Scalability of Sparse Linear Solver

In this and later sections we present scalability results of individual IPM kernels as well as the entire application from our cycle-accurate simulator. The results are shown for 1 to 64 cores in a stacked bar-chart format. Each bar is broken down into multiple sub-bar segments, where each segment represents an incremental improvement due to a given software or hardware optimization. The speedup is reported relative to the performance of serial implementation of IPM.

Inherent Parallelism in Sparse Solver

To better understand the speedup sparse linear solver achieves on CMP system, we compute the maximum ideal speedup of Cholesky, forward and backward solvers for different levels of parallelism. The ideal speedup is computed as the ratio of the work performed by the serial implementation of the algorithm over the work on the elimination tree's longest path when a given level of parallelism is explored. The amount of work on the longest path is the cumulative sum of work performed by each super-node along the path. In this idealized study the work per super-node is approximated using total number of floating-point operations required to process the super-node. When coarse-grain parallelism is explored, the work within each super-node is performed serially, while work across independent super-nodes is performed in parallel, constrained only by the data-dependencies among super-nodes in the elimination tree. When fine-grain parallelism is explored, the work within each super-node can also be done in parallel, constrained only by the true data dependencies within super-node's processing task.

The results for Cholesky factorization, backward and forward solvers are shown in Table 4. Column 2 shows ideal speedup for Cholesky factorization when coarse-grain (Level-1) parallelism is explored. We observe that exploring only Level-1 parallelism limits the ideal speedup to less than 20X for three out of five datasets. For these datasets the longest path is dominated by the large super-nodes at the top of the elimination tree. Column 3 shows ideal speedup when all three levels of parallelism are explored. This allows the work within each super-node to be done in parallel and, given large amount of inherent fine-grain parallelism within each super-node, results in tens of thousand-fold ideal speedup for all datasets.

Columns 4 and 5 show ideal speedup for backward solver. The results are almost identical for forward solver. Similar to Cholesky, Level-1 parallelism is limited, while the combined Level-1 and Level-2 parallelism is abundant.

Table 4: Inherent parallelism in sparse linear solver

Datasets	Cholesky Factorization		Backward (Forward) Solver	
	Level-1	Level-123	Level-1	Level-12
ken-18	147	353608	134	5024
mod-2	13	91074	12	1230
pds-10	8	105867	5	970
watson	68	134703	138	7663
world	16	83235	15	1155

These results suggest that sparse direct solver has a potential for high scalability on parallel architecture, due to a large amount of inherent parallelism that exists within the given datasets. Understanding the difference between the ideal and achieved speedup points to algorithmic and hardware limitations, which prevent application from realizing architecture's full potential. Furthermore, as shown in the remainder of this section, it helps guide performance tuning of an application, as well as make hardware improvements to the baseline architecture.

Forward Solver

Figure 7(a) shows the scalability of forward solver. The bottom bar segment shows the speedup achieved by the baseline version, which only explores Level-1 parallelism using software task queues and fine-grain locking for parallel reduction. On average, the speedup does not exceed 3.8X on 64 cores and begins to drop significantly on the configurations with 16 or more cores. As Table 4 shows, for three datasets, *mod-2*, *pds-10* and *world*, such poor scalability is due to the limited amount of Level-1 parallelism. As the number of cores increases beyond limit, scalability doest not improves further. Meanwhile, the amount of synchronization among the cores also increases. This results in slowdown on 16 and more cores. While the remaining two datasets, *ken-18* and *watson*, have sufficient amount of Level-1 parallelism, their limited scalability is due to the high overhead of task queuing and parallel reduction, as explained later in this subsection.

The second bar segment (from the bottom) shows the additional speedup (on top of the baseline) which results from exploring both Level-1 and Level-2 parallelism using the same software task queues and fine-grain lock reduction as in the baseline version. Most datasets show no improvement, while *pds-10* shows negligible improvements on 16 cores. To understand the sources of such poor parallel performance, we profile the parallel execution of forward solver. Figure 7(b) shows, for 1 to 64 core configurations, the amount of time spent in actual computation (top segment), parallel reduction (middle segment) and task queues (bottom segment). To achieve linear scalability, time spent in each of these three regions should decrease linearly as the number of cores increases. However, as the figure shows, time spent in task queues decreases only slightly for 4 datasets, and even increases for *mod2*, *pds-10* and *world*, on 32 and 64 cores. This is due to software overhead of scheduling small tasks, which result from exploring both levels of parallelism. Similarly, the time spent in reduction increases for all but one dataset (*mod2*) as the number of cores increases. This is due to the fact that the overhead of fine-grain locking increases with the number of cores, because the contention for the shared data also increases.
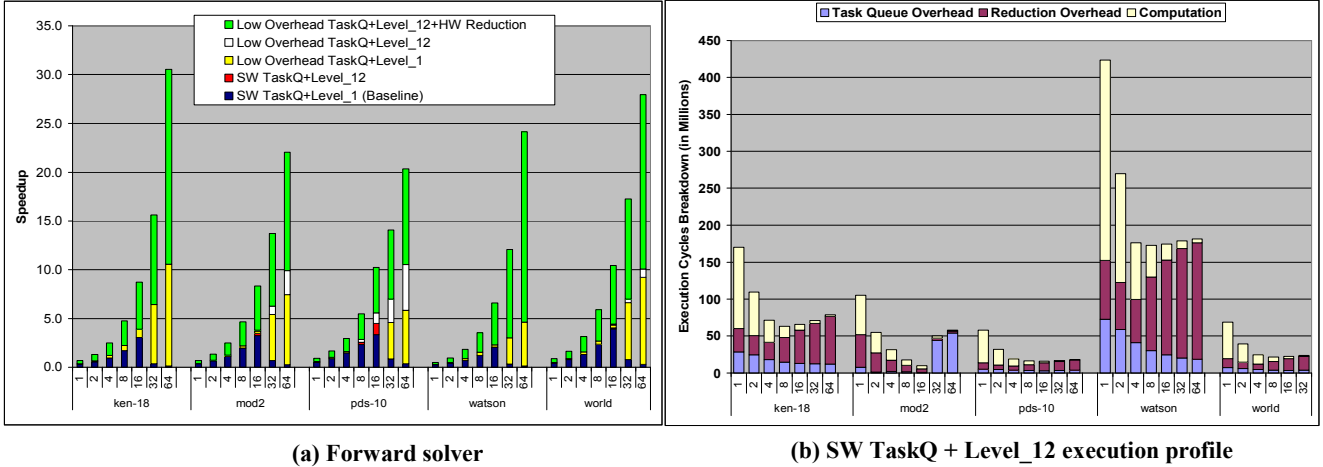
(a) Forward solver   **(b) SW TaskQ + Level_12 execution profile**

Figure 7: Scalability of forward solver

The contention is especially serious in *ken-18* and *watson* which leads to significant reduction overhead.

The third bar segment in Figure 7(a) shows an additional speedup achieved by forward solver with the help of low overhead hardware task queues when only Level-1 parallelism is explored. Scalability of all datasets improves significantly all the way to 64 cores. The experiment shows that software task queue overhead is the reason why we have not seen this level of scaling in the previous two experiments. As shown in Table 4, while *ken-18* and *watson* have ample amount of Level-1 parallelism, other datasets need to rely on Level-2 parallelism to achieve higher speedup. This is confirmed with the fourth bar segment which shows the speedup achieved using low overhead task queues when both Level-1 and Level-2 parallelism are explored. Except for *ken-18* and *watson*, the remaining datasets show additional speedup.

The achieved speedup is still far from the ideal, even when both levels of parallelism are explored. The topmost bar segment demonstrates forward solver performance with added hardware support for parallel reduction. Reduction hardware more than doubles scalability of most of the datasets on 64 cores compared to fine-grain locking, resulting in speedup between 20X and 30X. Note that scalability of *watson* increases by more than 5X on 64 cores, which is due to high overhead of the reduction in this particular dataset, which is alleviated with the reduction hardware. Overall, with the help of hardware support for low overhead task queues and reduction hardware, forward solver has improved parallel scalability, on average, from 3.8X to 24X on 64 CMP cores.

Backward Solver

Figure 8(a) shows scalability of backward solver. The bottom shows the speedup achieved by original parallel PARDISO version of the backward solver which only explores Level-1 parallelism using software task queues. The second bar segment from the bottom shows the speedup when both levels of parallelism are explored. Similar to forward solver, the speedup varies between 3X and 6X across all datasets, and begins to drop after 16 cores. Execution time analysis similar to Figure 7(b) shows that such poor scalability is to due to high overhead of software task queues.

The third and fourth bar segments show an additional speedup achieved by backward solver using low overhead task queues for Level-1 and Level-1/Level-2 parallelism, respectively. Results are similar to forward solver; hardware task queues result in the substantial speedup of 19X to 36X on all datasets. The backward solver scales better than the forward solver because it does not require reduction operation.

Cholesky factorization

As execution time breakdown in Table 1 shows, Cholesky factorization is the most time-consuming kernel in IPM. Figure 8(b) shows the scalability of Cholesky factorization.

The bottom bar segment shows the additional speedup achieved by original parallel Cholesky found in PARDISO solver which only explores Level-1 parallelism using software implementation of task queues. The speedup varies across datasets, and is consistent with the ideal speedup in Table 4. For examples, *watson* and *ken-18*, which according to the table have a large amount of Level-1 parallelism, achieve the highest speedup of 8X and 38X on 64 cores. On the other hand, *mod2*, *pds-10*, and *world*, whose ideal Level-1 speedup is limited, achieve a more modest speedup on 64 cores.

The second bar segment (from the bottom) shows the additional speedup achieved when all three levels of parallelism are explored. The speedup of *mod2*, *pds-10*, and *world* improve above 30X on 64 cores. The impact of fine-grain parallelism is limited on *ken-18* and *watson* due to the fact that both have plenty of coarse Level-1 parallelism. Overall, we see that in order to achieve good parallel speedup on Cholesky, it is important to exploit all levels of parallelism.

In contrast to forward and backward solvers, using low overhead task queues results in negligible performance improvement. This can be seen from by the third and fourth bar segments. The third segment shows an additional speedup achieved by Cholesky when Level-1 parallelism is explored. Similarly, the fourth segment shows an additional speedup when all three levels of parallelism are explored. We see that, compared to both solvers, the impact of low overhead task queues is almost unnoticeable on Level-1 parallelism and is small, between 8% and 16%, on all three levels of parallelism. This due to the fact that Level-1, Level-2 and Level-3 tasks in Cholesky are much larger
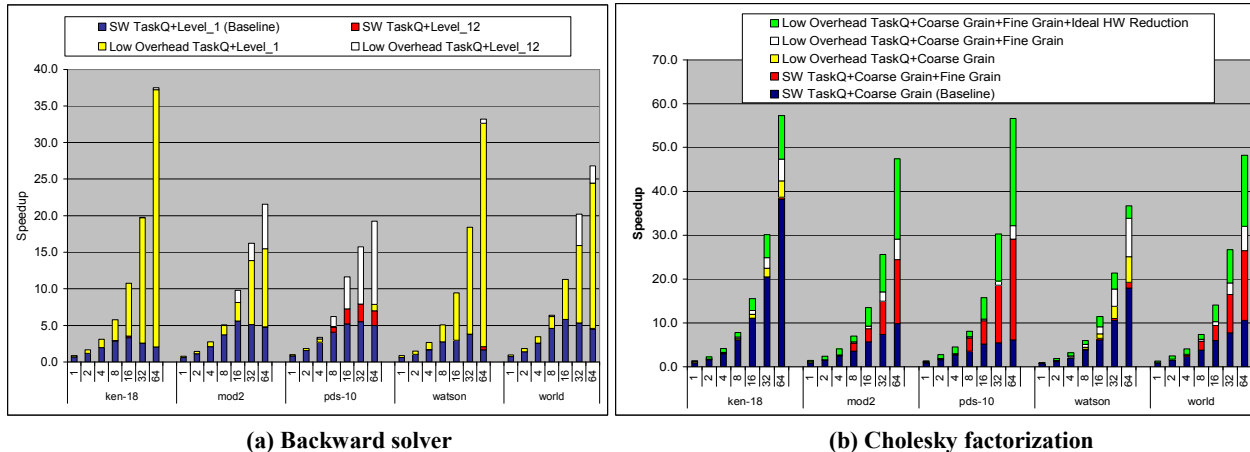
| (a) Backward solver | (b) Cholesky factorization |

Figure 8: Scalability of backward solver and Cholesky factorization

than in forward and backward solvers, and can effectively hide the overhead of task scheduling.

Even when three levels of parallelism are explored in all datasets, the utilization is still less than 50% on 64 cores. This is due to the overhead of reduction operation. As shown by the topmost bar segment, the use of reduction hardware improves scalability significantly, resulting in close to linear speedup on 64 cores in 4 out of 5 datasets. *watson* scalability is still limited to 30X. This is due to the following reason. Our implementation uses shared counters, one per tree node, to keep track of parent/child relationship among nodes. When the child node updates the parent node, during cmod update operation (see Section 4.1), it atomically decrements the shared counter to register its update. When multiple children, running on different threads, try to simultaneously update the same counter, execution is serialized to ensure atomicity of updates. *w atson* has a significant number of tree nodes with many children; this results in substantial parallel overhead due to such serialization. For other datasets, the average number of children per tree node is much smaller than in *watson*, hence serialization overhead is negligible.

6.4 Scalability of MMM, MVM and BLAS1

We have also simulated three remaining kernels, MMM, MVM and BLAS1. MMM scales almost linearly up to 64 cores for all datasets. Partitioning MMM into the blocks of non-zero elements, as described in Section 4.4, creates many large tasks. This results in good load balance and high parallel scalability.

MVM and BLAS1 exhibit more modest scalability from 16X to 40X across all datasets. These routines have little amount of data reuse within a single iteration of IPM, while their working set does not stay resident in L2 cache across consecutive iterations of IPM, due to large memory footprint required by sparse linear solver. As a result, memory bandwidth limits scalability of these two kernels.

6.5 Scalability of Interior Point Method

Figure 9 shows incremental scalability results of the entire IPM application. These results track the scalability of individual IPM kernels, most importantly the sparse linear solver.

The bottom bar segment shows the scalability of the original version of IPM which uses a baseline sparse solver. The speedup does not exceed 8X on 16 cores and exhibits a slow-down on 32 and 64 cores for all datasets. This behavior is expected as *mod-2*, *pds-10* and *world* have a limited amount of Level-1 parallelism in the sparse solver. While Cholesky scales well with *ken-18* and *watson*, IPM scalability for these two datasets is limited by a high overhead of task queuing in forward and backward solvers and parallel reduction forward solver. As indicated by the third bar segment, Level-1 parallelism with low overhead task queues significantly improves scalability in these two datasets. As expected, this is due to large scalability gains in forward and backward solvers.

The second bar segment (from the bottom) shows that exploring all levels of parallelism with software task queues provides modest scalability improvement in *mod-2*, *pds-10* and *world*. Exploring all levels of parallelism with low overhead task queues significantly improves the scalability of these three datasets, as shown by the fourth bar segment. Compared to software implementations, hardware support for low overhead task queues improves IPM performance 8-fold to 20-fold on 64 cores.

Finally, hardware support for parallel reduction improves IPM performance up to 2-fold compared to fine-grain locking used in the baseline version. As shown by the topmost bar segment, hardware support for low overhead task queues and parallel reduction enable IPM to achieve up to 48X speedup (43X on average) on a 64-core CMP.

7. RELATED WORK

Previous work on parallel IPM has been done in the context of shared memory or message passing multiprocessing systems. To our knowledge, this is the first work which parallelizes and analysis IPM on CMP platform using cycle accurate simulation, and achieves as high as 75% parallel efficiency on 64 cores.

A number of authors [Eckstein92][Lustig92][[Gondzio04] have parallelized interior point methods. However, their work targets specially structured problems, whereas our implementation targets general unstructured sparse datasets. Parallel implementation of IPM reported in [Karypsis94] achieves less than 50% parallel efficiency on 64 processors of CUBE 2 message

passing multi-processor. [Lustig96] presents results of parallelization of CPLEX IPM and reports up to 38% parallel efficiency on 32 processors of Power Challenge shared memory supercomputer. More recently, parallel implementation of IPM based on PCx [Koka04] and PARDISO report modest speedup on 4-way SMP systems.

There is a great body of work on parallelization of sparse linear solver, which is an important computational kernel of IPM. Solvers, such as SuperLU [Li96], WSMP [Gupta00], and PARDISO [Schenk00], are designed to run on shared memory parallel systems. Several others, such as block-oriented solver in [Rothberg93], MUMPS [Amestory01], and SuperLU DIST [Li03], are implemented using message passing to run on distributed memory message passing machines. None of these solvers has been parallelized to scale on the emerging large-scale CMP systems.

To the best of our knowledge all previous implementations of IPM use software implementation of task queues. We are the first to demonstrate the significant impact of hardware support for low overhead task queues on scalability of IPM. Hardware support for parallel reduction has been proposed and studied in the past by many authors [Gottlieb84][Leiserson96][BlueGene02]. While these proposals provide a dedicated network for reduction operations, our work integrates parallel reduction hardware support into coherence protocol, and is the first to demonstrate how it improves IPM performance on CMP platform.

8.   CONCLUSIONS

In this paper we describe parallelization of interior-point method (IPM) aimed at achieving scalable performance on large-scale chip-multiprocessor (CMP). We present parallelization of IPM computational kernels, as well as address major bottlenecks preventing scalability on many cores. Furthermore, we evaluate the impact of several hardware features to improve IPM parallel performance on large-scale CMP. Through our cycle accurate simulator, we demonstrate how exploring multiple levels of parallelism, with the help of hardware support for low overhead task queues and parallel reduction enables IPM to achieve up to 48X speedup (43X on average) on 64-core CMP.

Acknowledgments

9.   BIBLIOGRAPHY

[Amestory01] P. R. Amestoy, I. S. Du, J. Koster, and J. Y. L'Excellent. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001

[Bixby02] Robert E. Bixby. Solving Real-World Linear Programs: A Decade and More of Progress. *Operations Research,* 50(1):3–15, 2002.
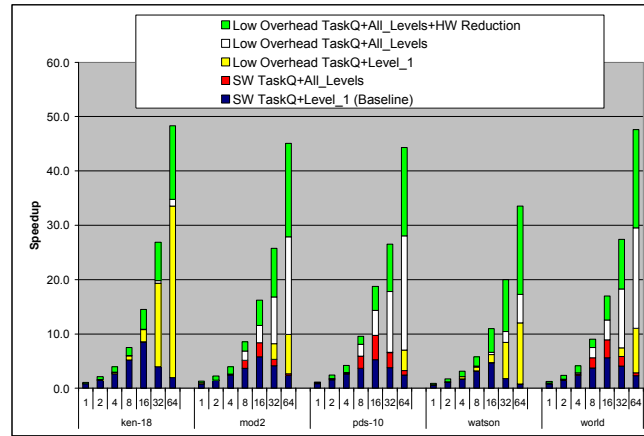

Figure 9: Scalability of entire Interior Point Method

[Czyzyk96] J. Czyzyk, S. Mehrotra, and S. J. Wright. PCx User Guide. Technical Report OTC 96/01, Optimization Technology Center, Argonne National Lab and Northwestern University, May 1996.

[Eckstein92] J. Eckstein, R. Qi, V.I. Ragulin and S. A. Zenios. Data parallel implementation of dense linear programming algorithms. Technical Report TMC-230, Thinking Machines Corporation, Cambridge, MA, 1992.

[Garzaran01] M. J. Garzaran, M. Prvulovic, Y. Zhang, A. Jula, H. Yu, L. Rauchwerger, and J. Torrellas. Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT),* 2001.

[Gay88] D.M. Gay. Electronic Mail Distribution of Linear Programming Test Problems. In *Committee on Algorithms Newsletter*, No. 13, pages 10–12.

[Gochman06] S. Gochman, A. Mendelson, A. Naveh, E. Rotem. Introduction to Intel® Core™ Duo Processor Architecture. *Intel Technology Journal*, Volume 10, Issue 2, 2006.

[Gschwind06] M Gschwind. Chip multiprocessing and the cell broadband engine. In *Proceedings or ACM Computing Frontiers* 2006, ACM Press, pages 1–8, 2006.

[Gondzio04] Gondzio, J. and A. Grothey. Exploiting Structure in Parallel Implementation of Interior Point Methods for Optimization. Technical Report MS-04-004, School of Mathematics, The University of Edinburgh, 2004.

[Gottlieb84] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – designing an MIMD parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, 1984.

[Gupta00] A. Gupta. WSMP: Watson Sparse Matrix Package (Part-II: Direct Solution of General Sparse Systems). Technical Report RC 21888 (98472), IBM T. J. Watson Research Center, 2000.

[Karypsis94] G. Karypis, A. Gupta, and V. Kumar. A parallel formulation of interior point algorithms. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, 1994.

[Karypis98] G. Karypis and V. Kumar. METIS - A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. Version 4.0. University of Minnesota, 1998.

[Kim03] D Kim, M Chaudhuri, and M Heinrich. Active Memory Techniques for ccNUMA Multiprocessors. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[Koka04] P. Koka, T. Suh, M. Smelyanskiy, R. Grzeszczuk, and C. Dulong. Construction and Performance Characterization of Parallel Interior Point Solver on 4-way Intel Itanium Multiprocessor System. *IEEE 7th Annual Workshop on Workload Characterization (WWC-7)*, 2004.

[Kumar2007] S Kumar, C. J. Hughes, and A Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2007.

[Kongetira04] P. Kongetira, K. Aingaran, and K. Olokotun. Niagara: A 32-Way Multithreaded SPARC Processor. IEEE Micro, 25(2):21–29, March/April 2006.

[Leiserson96] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.

[Li96] X. S. Li. Sparse Gaussian Elimination on High Performance Computers. PhD Dissertation, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1996.

[Li03] X. S. Li and J. W. Demmel. SuperLU DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Transactions on Mathematical Software*, 29(2):110-140, June 2003.

[Lustig92] J. Lustig and G. Li. An implementation of parallel primal and dual interior-point method for block structured linear programs. *Computational Optimization and Applications (COAP)*, 1 (1992) 141-161.

[Lustig96] I. J. Lustig and E. Rothberg. Gigaflops in Linear Programming. *Operations Research Letters*, 18(4):157–165, 1996.

[MKL07]*Intel® Math Kernel Library Reference Manual*, 2007

[Ng93] E. Ng and B. Peyton. Block Sparse Cholesky Algorithms on Advanced Uniprocessor Computers. *SIAM Journal on Scientific Computing,* 14(5):1034–1056, 1993.

[Nocedal06] J. Nocedal and S.J. Wright. Numerical Optimization, Springer, 2nd edition, 2006.

[Rothberg93] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Proceedings of ACM/IEEE conference on Supercomputing*, pages 503-512, 1993.

[Schenk00] O. Schenk. Scalable Parallel Sparse LU Factorization Methods on Shared memory Multiprocessors. Ph.D. Dissertation, Swiss Federal Institute of Technology, Zurich, Switzerland, 2000.