Volume 09 Issue 02 Published, May 19, 2005 ISSN 1535-864X DOI: 10.1535/itj.0902

Intel® Technology Journal

Compute-Intensive, Highly Parallel Applications and Uses

Parallel Computing for Large-Scale Optimization Problems: Challenges and Solutions

More information, including current and past issues of Intel Technology Journal, can be found at:
http://developer.intel.com/technology/itj/index.htm

Parallel Computing for Large-Scale Optimization Problems: Challenges and Solutions

Mikhail Smelyanskiy, Corporate Technology Group, Intel Corporation
Stephen Skedzielewski, Corporate Technology Group, Intel Corporation
Carole Dulong, Corporate Technology Group, Intel Corporation

Index words: optimization, linear programming, quadratic programming, interior point method, sparse linear system of equations, Cholesky factorization, backward solver, forward solver, elimination tree, supernode, parallel computing, multiprocessor system, shared-memory programming model, message-passing programming model, problem structure, block-angular matrices, asset liability management

ABSTRACT

Optimization refers to the minimization (or maximization) of an objective function of several decision variables that have to satisfy specified constraints. There are many applications of optimization. One example is the portfolio optimization problem where we seek the best way to invest some capital in a set of n assets. The constraints might represent a limit on the budget (i.e., a limit on the total amount to be invested), the requirement that investments are nonnegative (assuming short positions are not allowed), and a minimum acceptable value of expected return for the whole portfolio. The objective or cost function might be a measure of the overall risk or variance of the portfolio return. In this case, the optimization problem corresponds to choosing a portfolio allocation that minimizes risk, among all possible allocations that meet the firm requirements. Another example is production planning and inventory: the problem is to determine the optimal amount to produce in each month so that demand is met while the total cost of production and inventory is maintained without shortages.

In recent years the Interior Point Method (IPM) has became a dominant choice for solving large optimization problems for many scientific, engineering, and commercial applications. Two reasons for the success of the IPM are its good scalability on existing multiprocessor systems with a small number of processors and its potential to deliver a scalable performance on systems with a large number of processors. IPM spends most of its runtime in several important sparse linear algebra kernels. The scalability of these kernels depends on several key factors such as problem size, problem sparsity, and problem structure. This paper describes the computational kernels that are the building blocks of IPM, and we explain the different sources of parallelism in sparse parallel linear solvers, the dominant computation of IPM. We analyze the scalability and performance of two important optimization workloads for solving linear and quadratic programming problems.

INTRODUCTION

Optimization refers to the minimization (or maximization) of an objective function of several decision variables that have to satisfy specified constraints. It enables businesses to make better decisions about how to commit resources, which include equipment, capital, people, vehicles, raw materials, time, and facilities.

While existing hardware performs well on problems with tens of thousands of constraints and hundreds of thousands of variables, it lacks the necessary computational and bandwidth resources to target future datasets whose solution will require teraflops of computation and gigaflops of bandwidth. As an example, consider the Asset Liability Management (ALM) problem from computational finance, where the goal is to coordinate the management of assets and liabilities over several time periods to maximize the return at the end of the final time periods. To hedge against risk requires diversification of a portfolio with many assets; considering more time periods means better planning. Our simple back-of-the-envelope estimate shows that modeling just three time periods and as few as seventy-four assets creates an optimization problem that takes about one hour to solve on today's platforms, but only ten seconds to solve on a teraflop platform of tomorrow.

In order for an optimization workload to achieve high performance on future parallel architectures, one needs to understand (i) the source of parallelism, (ii) how the parallelism changes for different optimization problems, and (iii) how to extract this parallelism for a given problem.

We focus on the Interior Point Method (IPM), a dominant choice for solving large-scale optimization problems in many scientific, engineering, and financial applications. While complex mathematical analysis is the driving force behind IPM, most of the algorithm's computation time is spent in a few sparse linear algebra functions: sparse linear solvers, matrix-matrix multiplication, matrix-vector multiplication, and a few others. Developing parallel systems that efficiently execute these few functions is paramount to high performance for the IPM.

In this paper, we discuss IPM workloads as well as several approaches to parallelizing IPM. First, we discuss important computational kernels that are the building blocks of IPM. Second, we explain several sources of parallelism in sparse parallel linear solvers, the dominant computation of IPM. We also describe how additional parallelism within IPM can be discovered by exploiting inherent problem structures. Thirdly, we present the scalability results and performance analysis of shared-memory IPM on several datasets from linear programming. This workload utilizes highly optimized, parallel routines from the Intel Math Kernel Library, built using the PCx framework and parallelized by our team. We also present scalability results and performance analysis of a structure-exploiting quadratic IPM workload, the Object-Oriented Parallel interior point Solver (OOPS), on asset liability and management problems.

OPTIMIZATION AND THEIR USAGE MODELS

An optimization problem, has the form

$$\text{minimize } f_0(x)$$

$$\text{subject to } f_i(x) \quad b_i, \; i = 1,\ldots, m$$

Here the vector $x = (x_1,\ldots,x_n)$ is the optimization decision variable of the problem, the function $f_0(x)$ is the objective function, the functions f_i , $i = 1,\ldots,m$, are the (inequality) constraint functions, and the constants $b_1,\ldots,b_m$ are the limits, or bounds, for the constraints. A vector x^* is called optimal, or a solution of the optimization problem if it has the smallest objective value among all vectors that satisfy the constraints.

There are several important classes of optimization problems, characterized by particular forms of the objective and constraint functions. As an example, the optimization problem is called a Linear Program (LP) if the objective and constraint functions f0,…, fm are linear functions of x. The LP optimization problem is of the form

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

where A is m by n the matrix of linear constraints, and vectors x, c, and b have appropriate dimensions.

Another important example, the convex quadratic optimization problem (QP), is of the form

$$\min c^T x + \tfrac{1}{2} x^T Q x, \text{ subject to } Ax=b, \; x \quad 0$$

where Q is n by n positive semidefinite matrix, and A, x, c, and b are the same as in LP.

LP and QP are important not only because many problems encountered in practice can be formulated as either LP and QP problems, but also because many methods for solving general non-linear programming problems (NLP) solve them by solving the sequence of linear (sequential linear programming) or quadratic (sequential quadratic programming) approximations of the original NLP problem.

There are many applications of optimization. In the radiation therapy planning optimization problem, when choosing a plan for any individual patient, one seeks to determine radiation beam directions and intensity with the goals of maximizing the delivered dose to the tumor while minimizing the dose in normal tissue and organs at risk. There exist different formulations of this problem as LP, QP, or NLP.

In production planning and inventory problems, the problem is to determine the optimal amount to produce in each month so that demand is met yet the total cost of production and inventory is minimized and shortages are not permitted. This problem has been traditionally solved using the LP approach.

Another example is the famous portfolio optimization problem where we seek the best way to invest some capital in a set of n assets. The variable x_i represents the investment in the *i*th asset, so the vector $x=(x_1,\ldots,x_n)$ describes the overall portfolio allocation across the set of assets. The constraints might represent a limit on the budget (i.e., a limit on the total amount to be invested), the requirement that investments are nonnegative (assuming short positions are not allowed), and a minimum acceptable value of expected return for the whole portfolio. The objective or cost function might be a measure of the overall risk or variance of the portfolio return. In this case, the optimization problem corresponds to choosing a portfolio allocation that minimizes risk, among all possible allocations that meet

the firm requirements. The problem is known as the Markovitz mean-variance optimization problem and is modeled using QP.

The last example is device sizing in electronic design, which is the task of choosing the width and length of each device in an electronic circuit. Here the variables represent the widths and lengths of the devices. The constraints represent a variety of engineering requirements, such as limits on the device sizes imposed by the manufacturing process, timing requirements that ensure that the circuit can operate reliably at a specified speed, and a limit on the total area of the circuit. A common objective in a device sizing problem is the total power consumed by the circuit. The optimization problem is to find the device sizes that satisfy the design requirements (on manufacturability, timing, and area) and are most power efficient. This problem can be modeled using LP or QP.

INTERIOR-POINT METHOD (IPM)

In the past decade, the IPM has become a method of choice for solving large convex optimization problems. As parallel processing hardware continues to make its way into mainstream computing, it becomes important to investigate whether parallel computation can improve the performance of this commercially vital application.

The IPM has a unified framework for LP, QP, and NLP. The method starts with the initial guess 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. A key to efficient implementation and parallelization of IPM is that all three algorithms depend on four linear algebra kernels listed below:

1. Form linear systems of equations, Mx=b, where M is the symmetric matrix of the form $M = \begin{bmatrix} -Z & A^T \\ A & 0 \end{bmatrix}$, where A is the original matrix of constraints. The matrix of this form is called *augmented* system. For linear programming problems, where Z is a diagonal matrix, one uses substitution of variables in the above linear system of equations, so that matrix M is reduced to *normal* equation form $M = AZ^{-1}A$. This requires a **matrix-matrix multiplication** operation.
2. **Cholesky factorization** of matrix M = L D L^T in order to solve the system of linear equations, Mx=b. Here L is lower triangular, D is diagonal if M is positive definite, and D contains 1 by 1 and 2 by 2 bocks if M is indefinite. This step is normally the most time-consuming step of the IPM.
3. **Triangular solver** uses result of factorization to solve a system of linear equations (L D L^T)x=b, using the following three steps
    a. Forward solver, solves Ly=b
    b. Diagonal solver solves Dz=y. Note that when normal equations are used in the case of LP, this step can be eliminated, because the diagonal matrix D is positive and hence M can be represented as M=(L') $(L')^T$, where L' = L $D^{1/2}$.
    c. Backward solver solves L^Tx=z
4. **Matrix vector multiply**: Ax, A^Tx (transpose matrix vector multiply), and Mx (symmetric matrix-vector multiply).

Other operations, such as inner products, vector additions, and vector norm computation contribute a small amount compared to the above operations.

We see that the parallel efficiency of IPM depends on the efficient parallel implementation of these four linear algebra kernels. For the majority of realistic problems, solving systems of equations (kernels 2 and 3) is the most time-consuming portion of the IPM. For most optimization dataset models, the underlying matrix M is very sparse. As will be explained in later sections, sparsity is important because it uncovers an additional coarse-level parallelism, which is otherwise unavailable in the dense problems.

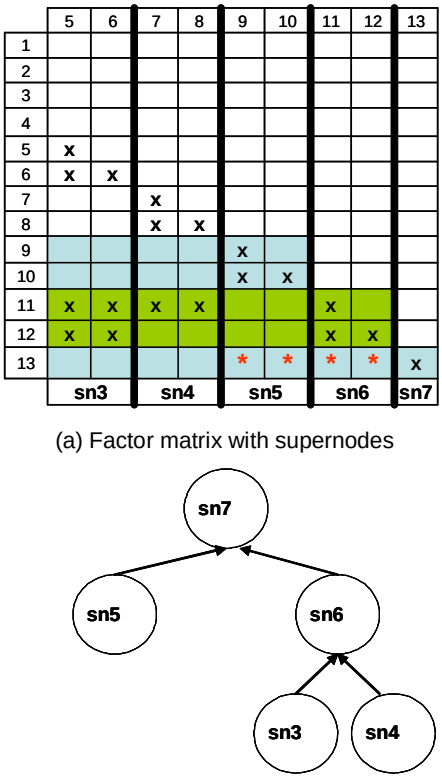
PARALLELIZATION OF IPM

In this section we describe the serial and parallel algorithm for solving sparse linear systems of equation. We discuss different levels of parallelism that can be explored for unstructured problems as well as additional levels of parallelism that become available in structured problems.

Sparse Unstructured Problems

This section deals with sparse unstructured matrices that arise from general optimization LP, QP, and NLP problems. Such matrices possess no distinct and exploitable non-zero structure. In the rest of this paper, we assume that the original symmetric matrix M is scattered into the factor matrix L. Two fundamental concepts behind solving sparse systems of linear equations are ***supernode*** and ***elimination tree***. A supernode is a set of contiguous columns in the factor 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 example of supernode is given in Figure 1(a).



(a) Factor matrix with supernodes

(b) Elimination Tree

Figure 1: Example of factor matrix L supernode and its elimination tree

Figure 1(a) shows 13x13 sparse matrix L: empty entries are assumed to have zero values. In this example there are six supernodes. For example, columns 1 and 2 form supernode sn1, columns 7 and 8 form supernode sn4, and columns 11, 12, and 13 form supernode sn6. Since all columns in the supernode have an identical non-zero structure, in practice non-zero elements of supernodes are compactly compressed into and stored as a dense matrix.

The elimination tree is a task dependence graph that characterizes the computation and data flow among the supernodes of L during Cholesky and triangular solve, and it is defined as follows: *parent(sn_j) = min{sn_i | i > j and at least one of the elements of sn_j which correspond to the diagonal block of sn_i is non-zero}*. In other words, the parent of supernode j is determined by the first sub-diagonal non-zero in supernode i. Figure 1(b) shows an example of the elimination tree for the matrix in Figure 1(a). We see that there is an edge between sn1 and sn5, because as the shaded portion of the figure shows, the second row of the 2 by 2 diagonal block of sn5 corresponds to the non-zero row 10 in sn1. Similarly, there is an edge between sn3 and sn6, because the first two rows of sn6 correspond to non-zero elements in rows 11 and 12 of supernode 3.

Given the elimination tree, the Cholesky factorization forward and backward kernels can all be expressed using the following generic formulation:

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

The order of the tree traversal and the processing task are different for each kernel. Cholesky and the forward solver perform bottom-up traversal of the elimination tree, whereas backward solver performs top-down traversal of the elimination tree. The processing task for forward and backward solver are very similar; however, we do not discuss them here due to space limitations. The details of a Cholesky processing task are discussed next.

Cholesky Factorization

Cholesky factorization is the most time-consuming operation among the four kernels. According to our experiments (see our experimental section results for unstructured problems) on average, IPM spends 70% of the time in this kernel. The high-level pseudo-code of Cholesky is given in Figure 2(a). Cholesky processing task (Lines 3-7) is generally expressed in terms of two primitive operations on the supernode, **cdiv** and **cmod**, both of which are shown in Figure 2(b) and Figure 2(c), respectively.

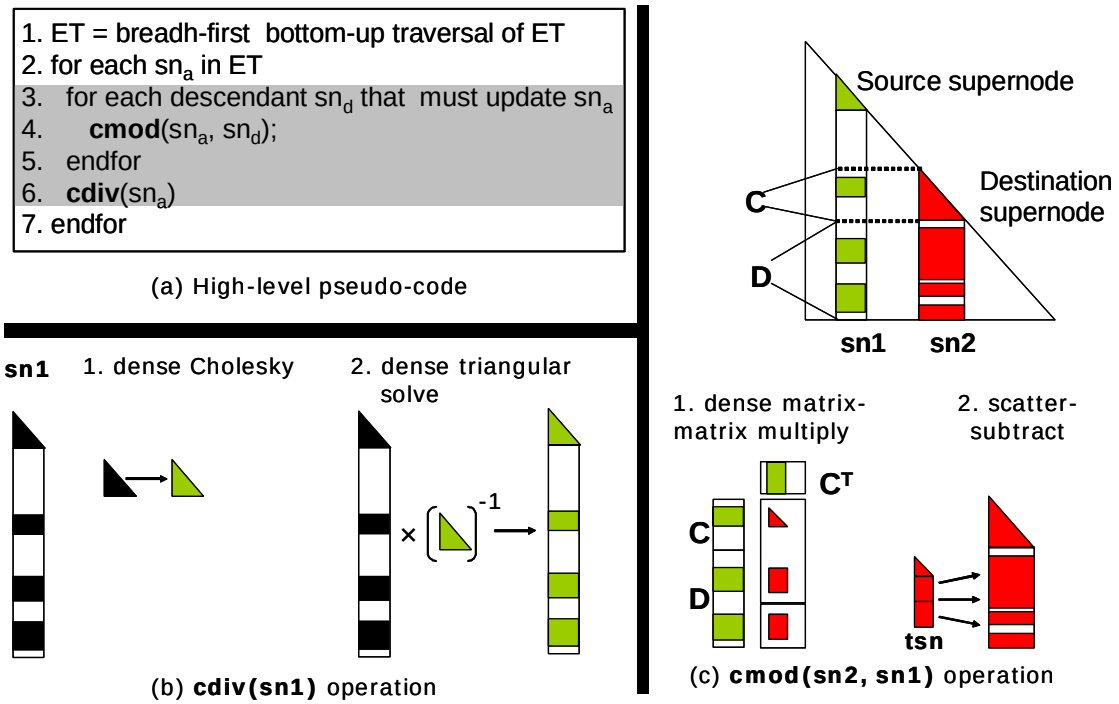


Figure 2: Cholesky factorization

Given supernode sn1, cdiv(sn1), also known as *supernode factorization*, requires multiplication of the dense rectangular portion of the supernode below its main diagonal by the inverse of the supernode's dense diagonal block. The inversion is not actually computed. Rather, this computation is broken into two steps shown in Figure 2(b) for supernode sn1. In the first step, we perform dense Cholesky on the diagonal block of sn1. In the second step, we solve a large number of triangular systems for each nonzero row of the supernode below the main diagonal.

The second and most time-consuming primitive operation in Cholesky factorization is called cmod, which is also known as *supernode-supernode update*. cmod(sn2, sn1) operation adds into destination supernode sn2 the multiple of the source supernode sn1 and, similar to cdiv, consists of two steps, shown in Figure 2(c). The first step uses dense matrix-matrix multiply, to multiply the sn1 by the transpose of its sub-matrix C which corresponds to the dense triangular block of the sn2. This results in the temporary supernode tsn. In the second step, the temporary supernode tsn is scatter-subtracted from the second supernode sn2. The scatter operation is required because tsn and sn2 may have different non-zero structures (which is the case in Figure 2(c)).

The high-level pseudo-code of Cholesky, shown in Figure 2(a), scatters original matrix M into the factor L and performs a series of cdiv and cmod operations on supernodes of L to factorize it. The algorithm performs breadth-first bottom-up traversal of the elimination tree (designated as ET in the figure) starting from the leaves (Line 2). Each supernode sn_a, receives cmod updates from its descendant supernodes sn_d (Lines 3-6). Upon receiving all the updates, a cdiv operation is performed on sn_a to complete factorization of the supernode. The factorized supernode is now ready to update its own ancestors. All ancestor supernodes that require an update from the descendent supernode are known in advance; the leave supernodes require no updates. Note that due to the fact that each supernode is stored as a dense matrix, one can use efficient implementations of dense linear algebra subroutines (such as BLAS 2, BLAS 3, and LINPACK) to perform cdiv and cmod operations.

Understanding Parallelism in Cholesky Factorization

We now examine the opportunities for parallelism in the above implementation of sparse Cholesky in Figure 2(a). This implementation contains parallelism on several different levels:

Level 1: Elimination tree parallelism, which corresponds to the parallel execution of the outermost loop in Lines 2-7. Here several iterations of the loop, which

correspond to independent sub-trees of the elimination tree, can be started in parallel on several processors. In other words, if T1 and T2 are disjoint sub-trees of the elimination tree, with neither root node a descendant of the other, then all of the supernodes of T1 can be factorized completely independently of the supernodes corresponding to T2, and vice versa. Hence, these computations can be done simultaneously by separate processors with no communication between them.

Level 2: The second level of parallelism exists in the innermost loop (Lines 3-5) and is essentially a parallel reduction operation. For a given ancestor supernode sn_a, all updates from its descendents (Line 4) can proceed in parallel. Note, however, that it may happen (more often than not) that two or more descendents will try to scatter-subtract into the same elements of their ancestor. This requires some locking mechanism to guarantee that only one update happens at a time.

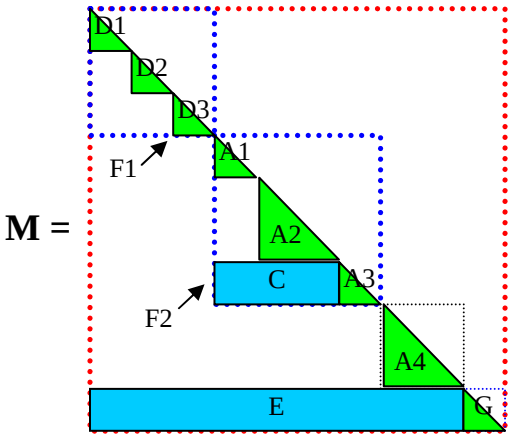
Level 3: The third level of parallelism exists within an individual cmod update operation called from the innermost loop (Line 4). Due to the fact that each cmod operation is composed of dense Cholesky and dense matrix-matrix product operation, each can be further parallelized. The parallelism also exists within the cdiv operation performed on a given supernode sn. As explained earlier, the cdiv(sn) operation involves solving a large independent set of dense triangular systems; hence all such solves can be done in parallel.

To quantify the parallelism inherent to sparse Cholesky factorization, we note that the number of operations required to factorize a typical sparse matrix on a single processor is roughly $\Theta(nnz^{3/2})$, where nnz is a number of non-zeros in the matrix. Assuming only one column per supernode, unlimited hardware resource and zero-communication cost, at each step of parallel computation we will execute as many parallel factorization operations as possible, constrained only by the data-dependencies within the elimination tree, which are inherent to a particular sparse dataset. Therefore the number of parallel steps to factorize the sparse matrix is a critical path through the elimination tree. The height of a well-balanced elimination tree is approximately $\Theta(\ln(N))$, where N is the number of row/columns of M. Thus, the expected ideal speed-up of Cholesky is approximately $\Theta(nnz^{3/2}/\ln(N))$.

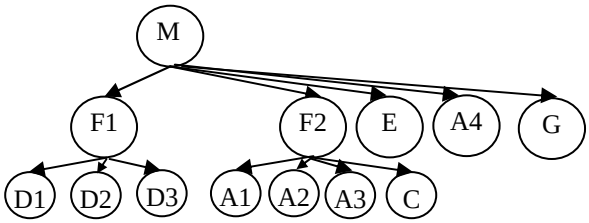
Sparse Structured Problems

Of the four levels of parallelism described above, the elimination-tree-level parallelism is the coarsest and therefore is the most attractive to exploit with parallel processing. However, to exploit this parallelism efficiently requires a balanced elimination tree. Different elimination trees can be constructed for a given symmetric matrix M when the matrix is re-ordered using symmetric row and column permutations. Obviously an elimination tree where all sub-trees have a similar height will result in better parallel speed-up than one where most of the nodes are in one long branch. However, finding a re-ordering of the matrix that leads to a more balanced elimination tree is a non-trivial task (in fact it is NP-complete).

In many situations, however, explicitly constructing a balanced elimination tree is not necessary. Many truly large-scale optimization problems are not only sparse but also display some flavor of block structure that make them highly amenable to parallelism [4]. By a block-structured matrix we understand a matrix that is composed of sub-matrices. A block-structured matrix can be nested, where each sub-matrix is a block-structured matrix itself. The example of the nested block-angular matrix is given in Figure 3(a).



(a) Example of block-angular matrix



(b) Tree representation

Figure 3: Nested block structured matrix and its tree representation

The nested block-structure of a matrix can be thought of as a tree. Its root is the whole matrix, and every block of a particular sub-matrix is a child node of the node representing this sub-matrix. Leaf nodes correspond to the elementary sub-matrices that can no longer be divided into blocks. Figure 3(b) shows an example of the

matrix M's tree. We see that sub-matrix F1 of M has a diagonal structure, whereas sub-matrix F2 of M has a block-angular structure.

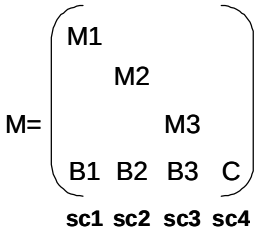
The existence of a well-defined structure is due to the fact that many optimization problems are usually generated by a process involving discretization of space or time (such as control problems or other problems involving differential equations), or of probability distribution (such as stochastic programming). Other sources of structure are possible such as in a network survivability problem where slight variations of the core network matrix are repeated many times. Note that a block of a block-structured matrix can itself be a sparse matrix. Therefore block structure offers the additional coarse level of parallelism, which is coarser than the elimination tree level of parallelism.

Structure-Oriented Cholesky Factorization

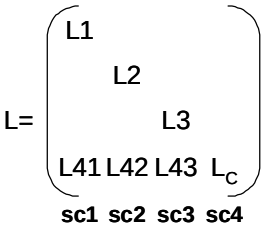
The efficient exploitation of matrix structure by linear algebra routines is based on the fact that any method supported by the linear algebra library can be performed by working through the tree: at every node, evaluating the required linear algebra operation for the matrix corresponding to this node can be broken down into child nodes in the tree. Different structures, however, need their own linear algebra implementation.

In Figure 4(a), we show by means of a simple example how Cholesky factorization can be implemented on block-angular matrix M. This matrix consists of 4 by 4 blocks. It is easy to see that the factor matrix L shown in Figure 4(b) is also block-angular. We partition L into 4-column blocks: sc_i consists of sub-blocks L_i and L_4i for i=1,2,3, and sc4 contains a single block L_C. Note that the column blocks are similar to supernodes for unstructured matrices. The only difference is that a supernode, by definition, must only contain sub-blocks of dense rows, whereas column blocks of a block-structured matrix can contain sub-blocks with an arbitrary sparsity pattern.

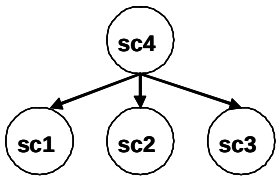
Despite this difference, the same concepts that applied to supernodes equally apply to column blocks. Figure 4(c) shows the elimination for matrix L, and Figure 4(d) shows the application of Cholesky factorization from Figure 2(a) to the column blocks of L. Step 1 is composed of two sub-steps: (i) cdiv(sc1) performs the required operations on sub-blocks L_1 and L_{14},. and (ii) cmod(sc4, sc1) updates L_C of sc4 with the corresponding product of L_{41} L_{41}^T. Note that sc1 only has to update sc4, since, as indicated by the elimination tree, sc4 is its only parent. Similar operations are performed on sc2 and sc3 in steps 2 and 3, respectively. Finally, the cdiv operation is performed on sc4 to factorize the sub-block L_C and to complete the Cholesky factorization of M.



(a) Block-angular matrix M



(b) Factor L of M



(c) Elimination Tree of L

Serial Steps	cdiv(sci)	cmod(scj, sci)
1.	cdiv(sc1): L1=Cholesky(M1), L41 = B1 $L1^{-T}$	cmod(sc4, sc1): $C_{tmp} = C - L41\ L41^T$
2.	cdiv(sc2): L2=Cholesky(M2), L42 = B2 $L2^{-T}$	cmod(sc4, sc2): $C_{tmp} = C_{tmp} - L42\ L42^T$
3.	cdiv(sc3): L3=Cholesky(M3), L43 = B3 $L3^{-T}$	cmod(sc4, sc3): $C_{tmp} = C_{tmp} - L43\ L43^T$
4.	cdiv(sc4): L_C=Cholesky(C_{tmp})	-

(d) Serial computation to factorize M

Figure 4: Exploiting block-angular matrix structure

Understanding Parallelism in Structure-Oriented Cholesky

The serial factorization algorithm in Figure 4(d) lends itself naturally to parallelization as shown in Figure 5. Steps 1, 2, and 3, which are completely independent, get distributed among the three processors, P1, P2, and P3. The resulting computation happens in Phase 1. Notice that instead of simultaneously updating the same matrix C_{tmp}, each processor P_i, stores the result of this update into its private copy of C_{tmp}, C_i. In Phase 2, the global reduction operation is performed, wherein each processor adds its own contribution to C_{tmp}. Finally, in

Phase 3, matrix C_{tmp} is factorized and the result is stored in L_C, which completes parallel factorization of M. Note that the reduction operation in Step 2 can be parallelized. In addition, work performed in Phase 1 on each processor, as well Cholesky factorization in Phase 3, can also be parallelized. If M1, M2, M3, or C_{tmp} is unstructured, the parallel algorithm described in the previous section can be used.

		Phase 1		Phase 2	Phase 3
		Independent Parallel Computation		(Parallel) Reduction	(Parallel) Cholesky
Parallel Steps	P1	L1=Cholesky(M1):L41 = B1 $L1^{-T}$	C1= L41 $L41^T$	C_{tmp}=C– C1–C2–C3	L_C=Cholesky(C_{tmp})
	P2	L2=Cholesky(M2):L42 = B2 $L2^{-T}$	C2=L42 $L42^T$		
	P3	L3=Cholesky(M3):L43 = B3 $L3^{-T}$	C3=L43 $L43^T$		

Figure 5: Split of computations between processors in structure-oriented Cholesky

Hence we see that by exploiting the special structure of the matrix, we are able to exploit the coarser level of parallelism unlike in cases of unstructured matrices.

PERFORMANCE ANALYSIS OF LINEAR AND QUADRATIC IPM WORKLOADS

In this section we present the scalability results and performance analysis of two applications. The first application is the shared-memory implementation of IPM for solving arbitrary unstructured linear programming problems. The second application is the MPI implementation of a structure-exploiting quadratic IPM workload (OOPS) for solving structured asset liability management quadratic programming problems. We performed both experiments on a 4-way 3.0 GHz Intel® Xeon™ processor MP-based system, with 8 GB of global shared memory and three levels of cache on each processor: 16 KB L1, 512 KB L2, and 4 MB L3. The four processors and memory are connected with a ServerWorks GC-HE, capable of delivering the peak bandwidth of 6.4 Gb/s.

Performance Characterization of IPM for Unstructured Linear Programs

For these experiments we use an Interior Point Solver (IPS) workload [3], which our team built for solving linear programming problems. IPS is based on PCx, a serial interior-point linear programming package developed at Argonne National Laboratory in collaboration with Northwestern University [5]. Our implementation of the Cholesky factorization and the solver routines uses a parallel sparse direct solver package, called PARDISO, developed at the University of Basel [6], which is now included as part of Intel's Math Kernel Library. In addition, we implemented and parallelized the routines for sparse matrix-matrix multiplication and sparse matrix-vector multiplication. Note that both the PARDISO code and our routines are parallelized using OpenMP.

Table 1 summarizes the statistics for the datasets used in our experiments. They mostly come from the standard NETLIB test set and represent realistic linear models from several application domains. Columns 1 and 2 show the number of variables and constraints in the constraint matrix for each problem. Column 3 shows the size of M, and Column 4 shows the number of non-zeros in its factor L. Note that the datasets are sorted in the order of increasing number of non-zeros. The last column shows the density of the factor matrix, computed as 100%*non-zeros/(neqns^2). We see that on average, the problems are fairly large and most of them are very sparse.

® Intel and Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Table 1: Characteristics of LP datasets

	nconstraints	nvariables	neqns	nlns	Density (%)
ken-18.dat	78862	128434	78862	2175306	0.034977
fleet12.dat	21616	67841	21616	4085152	0.874294
pds-20.dat	32287	106180	32287	6388010	0.612788
fome13.dat	47872	97144	47872	10668201	0.465509
snp30lp1	297998	953120	297998	20352321	0.022919
gismondy.dat	18262	23266	18262	30887926	9.261729

Figure 6(a) shows the breakdown of total execution time spent in the main optimization loop into the four important parallel regions and the remaining serial region. The parallel regions are Cholesky factorization, triangular solver (both forward and backward), matrix-matrix multiply (mmm), and matrix-vector multiply (mvm). For each dataset, we show four bars corresponding to one (1P), two (2P), and four (4P) processors, respectively. Each bar is broken into five parts, one for each execution region. Note all the times are relative to one processor runtime, and the number on the top of one processor shows the total time spent in the main optimization loop. We see that for many datasets Cholesky is the most time-consuming kernel (main optimization loop of IPM spends on average 70% factorizing the matrix), and it also achieves good scalability for these datasets. The solver, which is the second most-time consuming kernel (17% of time on average), scales worse compared to the Cholesky and will require a considerable tuning effort in order for IPM to scale well on larger numbers of processors. Another 4% of time is spent in parallel mmm, which scales very well up to four processors. An additional 4% of the time is spent in different flavors of parallel mvm. The remaining 5% of the time is spent in the serial region.

Figure 6(b) reports the speed-up of IPS on one, two, and four processors. The highest speed-up (2.7x on four processors) is attained for the gismondi dataset, which is also the largest dataset in terms of the number of non-zero elements. Comparing Table 1 and Figure 6(b), we see that both the run-time and scalability of IMS are almost perfectly correlated with the number of non-zeros in the factor matrix that represents the problem size. This is encouraging as it suggests that parallel computation improves the performance on harder problems.

We used the Intel Thread Profiler, a parallel performance analysis tool, to identify and locate bottlenecks that are limiting the parallel speed-up of IPS. In this paper we only present the results for Cholesky factorization. The Thread Profiler identifies three important factors that adversely impact speed-up:

1. *Load imbalance* is the time the threads that completed execution wait at a barrier at the end of the parallel region until all remaining threads have completed the assigned work of the region. When unequal amounts of computation are assigned to threads, threads with less work to execute sit idle at the region barrier until those threads with more work have finished.
2. *Locks* is the time a thread spends waiting to acquire a lock.
3. *OpenMP overhead* is the time spent inside the OpenMP Runtime Engine that implements OpenMP.

For all datasets, Figure 7 shows the total Cholesky factorization time (summed over all processors) spent executing instructions (busy time), waiting on acquiring the locks (locks time), waiting on barriers due to load imbalance (imbalance time), and the time spent inside the OpenMP engine (OpenMP overhead time). Note that all results on one, two, and four processors are normalized to the one processor time. Perfect speed-up is possible only when the total time does not increase as the processors increase. As expected from Figure 6(b), ken-18 has the worst speed-up because the busy, wait, and OpenMP times increase by 150% for two processors and by as much as 350% for four processors. We can also observe a 30-60% increase in busy time on two and four processors for the other five datasets. By looking at the source code we identified the cause of these increases: they are due to the busy waiting loop inside the PARDISO implementation of Cholesky, wherein several processors try to acquire a shared lock in order to enter a critical region. The modest scalability of the triangular solver is due to the same reasons. The future implementation of the sparse linear solver within PARDISO is likely to address this issue.

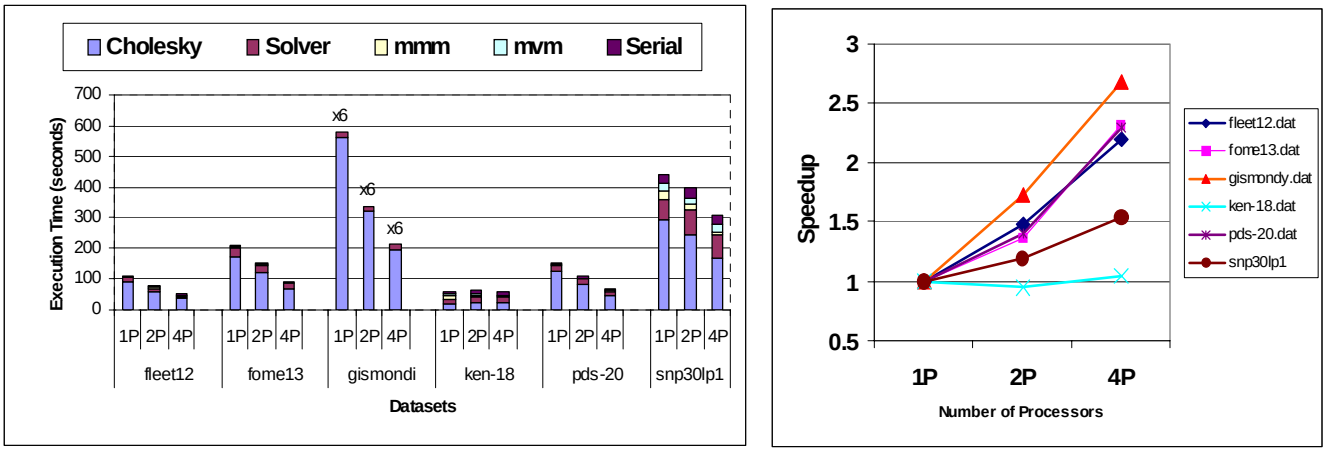


Figure 6: Parallel performance of main optimization loop of IPS

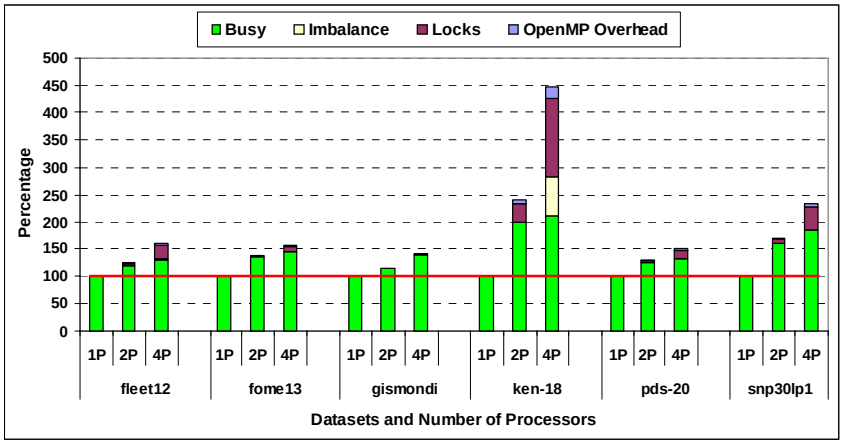


Figure 7: Concurrency and load balance in the parallel Cholesky factorization

Performance Characterization of IPM for the Structured Quadratic Program

For these experiments we used the OOPS workload [7], which we obtained from researchers at the University of Edinburgh. This uses a quadratic programming variant of IPM to solve the ALM problem. ALM is the process of finding an optimal solution to the problem of minimizing the risk of investments whose returns are uncertain. The method associates a risk probability to each asset and uses discrete random events observed at times t = 0, …, T to create a branching scenario tree rooted at the initial time. At each time step the probability of reaching a given node is computed by looking at its predecessor nodes. At the end of the process (time T+1) we can assign a probability to each outcome and compute the asset value at that time. The probability at the leaves of this branching tree will sum to one and we can assess the risk by looking at the asset value vs. the probability graph. The above steps are formulated as a structured quadratic problem with a block-angular structure, which is solved using OOPS.

The research team at the University of Edinburgh also provided problem sets that are summarized in Table 2. Columns 1, 2, and 3 show the number of time steps, the blocked matrices that compose the problem, and the number of assets. The last five columns are the same as given in the linear optimization. Again, we see that these problems are fairly large and very sparse.

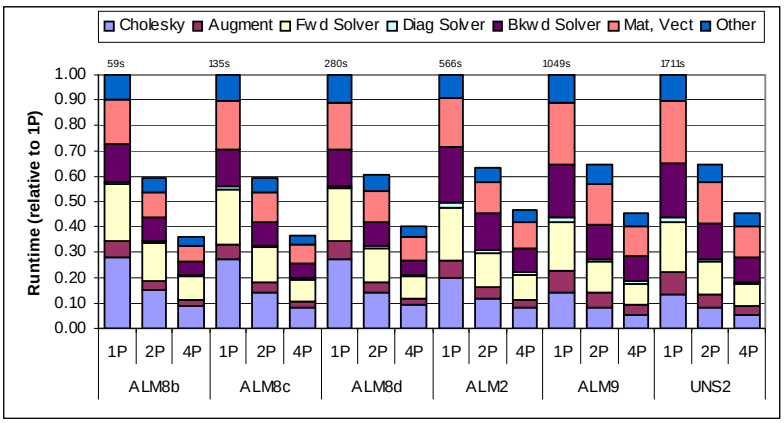
Table 2: Characteristics of ALM datasets

	steps	blocks	assets	nconstraints	nvariables	neqns	non-zeros	Density (%)
ALM8b	3	33	50	57,274	168,451	57,274	1,009,800	0.03%
ALM8c	3	50	50	130,102	382,651	130,102	3.378,750	0.02%
ALM8d	3	70	50	253,522	745,651	253,522	9,070,250	0.01%
ALM2	6	10	5	666,667	1,666,666	666,667	3,611,075	0.08%
ALM9	5	24	4	2,077,207	5,193,016	2,077,207	23,368,500	0.01%
UNS2	109	40	40	2,160,919	5,402,296	2,160,919	27,071,115	0.00%

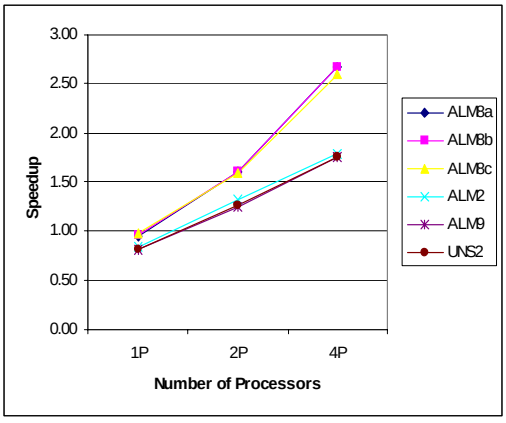
These inputs were run on the same 4-way 3.0 GHz Intel Xeon processor MP-based system that we described earlier. Figure 8(a) shows the breakdown of total execution time for OOPS. The regions appear to be slightly different than in the case of unstructured LP. As explained in the Interior Point Method section, OOPS builds an augmented matrix in each iteration of the optimization loop, whereas IPS performs mmm to form the normal matrix. This matrix is factorized using structure-exploiting Cholesky. Triangular solvers are similar to IPS, but many calls to operations on vectors and matrices are combined into the "Mat, Vect" region. For each dataset, we show four bars corresponding to one (1P), two (2P), and four (4P) processors, respectively. The time shown in this graph is relative to the time taken on the one-processor run. The total time (in seconds) for a one-processor run is given above its bar. The factorization routine has a large parallel section, followed by a global reduction (serial), followed by the redundant Cholesky factorization, which is duplicated in each processor (as described above). This duplication minimizes the communication, but causes the factorization step to exhibit less than linear scalability, as shown in our measurements. A similar pattern (parallel, serial, duplicate-parallel) occurs in the forward and backward solver routines, and we see a similar speed-up as in the factorization step. Since the solver takes a larger fraction of time in OOPS than in IPS, we broke it down into its components. The Mat, Vect section scales in a similar manner to the other routines. These routines have not been heavily optimized and have headroom for additional improvement. We also see a significant amount of overhead (11-13%) on the one-processor run on the larger data sets when compared with a serial version of OOPS. We speculate that this overhead is due to shared memory implementation of MPI, and we plan to investigate the cause of this overhead in our future work.

Figure 8(b) reports the speed-up of IPS for the test datasets on one, two, and four processors. The scalability appears to be correlated with the amount of work required to factor the constraint matrix M. The scaling of OOPS does not yet exploit all of the parallelism that is present in the algorithm.

To understand the performance overhead of the MPI calls, Figure 9 shows the results from running the Intel trace analysis tools on this workload. It instruments the code and measures the time waiting for messages. The instrumented runs show that a relatively small amount of time is spent in the MPI libraries and that almost all of that time is in the MPI reduction routine. It corresponds most closely to the "imbalance" portion of the OMP breakdowns. The rest of the additional time is spent in the OOPS code. We are investigating the source of this extra time.



(a) Execution time breakdown



(b) Speedup

Figure 8: Parallel performance of main optimization loop of OOPS

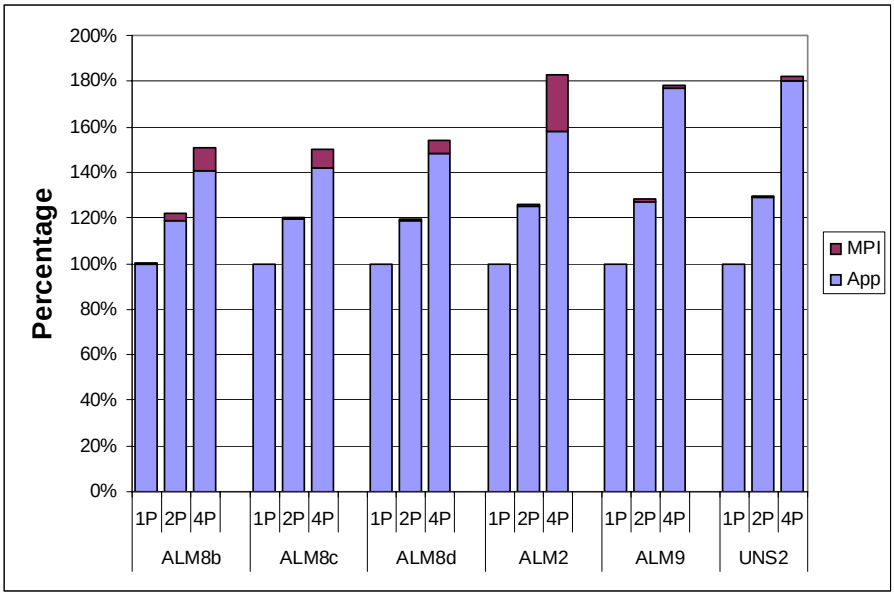


Figure 9: Concurrency and load balance in OOPS

CONCLUSION

In this paper we described a parallel IPM for solving optimization problems. The performance of IPM depends on several key sparse linear algebra kernels. The most important kernel is the solution of the sparse linear system of equations. We described serial and parallel implementations of the sparse linear solver for both unstructured and structured optimization problems.

We have done performance and scalability analysis of IPS–a linear optimization workload for solving unstructured linear programs. We reported up to 2.7x speed-up on the 4-way 3.0 GHz Intel Xeon processor MP-based system for a diverse set of linear problems. We also presented the performance and scalability analysis of OOPS–a structure-exploiting quadratic optimization workload for solving structured quadratic problems. OOPS exposes parallelism by passing structure information from the high-level optimization problems into the linear algebra layer. We achieved up to a 2.7x speed-up on the number of datasets from important asset liability management problems.

Overall, we observed that the scalability of IPM depends on several key factors such as problem size, problem sparsity, as well as problem structure. Although we observed similar performance scalability for the linear unstructured problems and the quadratic structured problems, the structured problems exhibit multiple levels

of parallelism that are not all exploited in the current OOPS implementation. This leaves headroom for performance scalability on systems with large numbers of processors, which we are going to explore in our future work.

One expects the optimization problem size to grow in the future. For example, an increased number of assets in an investor's portfolio will lead to better risk diversification and hence higher return on investment. Many truly large-scale optimization problems are not only sparse but also display block-structure, because these problems are usually generated by discretizations of space or time. These large optimization problems will clearly benefit from a system capable of exploiting multiple levels of parallelism from fine grain to coarse grain.

ACKNOWLEDGMENTS

We acknowledge Radek Grzeszczuk for the key role that he played in the effort to identify and understand Interior Point Method as an important optimization workload. Our thanks go to Jacek Gondzio and Andreas Grothey from the University of Edinburgh for providing the OOPS code and assisting us in understanding its functionality. We also acknowledge Bruce Greer for helping us to acquire and understand the PARDISO solver source code. Finally, we thank Dmitry Ragozin from Intel for his help in understanding the performance bottlenecks of the PARDISO solver.

REFERENCES

[1] J. Nocedal and S.J. Wright, *Numerical Optimization*, Springer-Verlag, New York, Inc, 1999.

[2] S. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge University Press, 2003.

[3] Pranay Koka, Taeweon Suh, Mikhail Smelyanskiy, Radek Grzeszczuk, and Carole 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*)*, October 2004.

[4] 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, December 18, 2004.

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

[6] University of Basel, PARDISO Direct Sparse Solver. http://www.computational.unibas.ch/cs/scicomp*.

[7] Gondzio, J. and A. Grothey, "Parallel Interior Point Solver for Structured Quadratic Programs: Application to Financial Planning Problems," *Technical Report MS-03-001*, School of Mathematics, The University of Edinburgh, April 16, 2003, revised in December 12, 2003.

AUTHORS' BIOGRAPHIES

Mikhail Smelyanskiy is a member of the Research Staff in the Corporate Technology Group. He received his B.Sc., M.Sc., and Ph.D. degrees in Electrical Engineering and Computer Science from the University of Michigan, Ann Arbor in 1996, 1999, and 2004, respectively. Since he joined Intel in September 2003, he has worked on future multi-core computer architecture to efficiently execute applications from the area of optimization and finance. His graduate work was on VLIW compiler scheduling algorithms for efficient resource utilization. His e-mail is Mikhail.Smelyanskiy at intel.com.

Stephen Skedzielewski is a senior researcher in the Workload Analysis Dept. in the Corporate Technology Group. He recently joined this group after spending nine years analyzing IPF compiler performance. His main technical interests are in performance analysis and parallel computing. He has a B.S. degree from Caltech and M.S. and Ph.D. degrees from the University of Wisconsin, Madison. His e-mail is Stephen.Skedzielewski at intel.com.

Carole Dulong is a senior researcher and computer architect in the Microprocessor Technology Lab. She leads a team of researchers working on various data-mining techniques. She joined Intel in 1990. She was a member of the IPF architecture definition team and contributed to the IPF compiler design. She graduated from Institut Superieur d'Electronique de Paris (France). Her e-mail is Carole.Dulong at intel.com.

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

developer.intel.com/technology/itj/index.htm