Construction and Performance Characterization of Parallel Interior Point Solver on 4-way Intel Itanium 2 Multiprocessor System

P. Koka [§†] T. Suh[§‡] M. Smelyanskiy[§] R. Grzeszczuk[§] C. Dulong[§]

[†]Department of ECE University of Wisconsin Madison

[‡]Department of ECE Georgia Institute of Technology

[§]Architecture Research Lab Intel Corporation

ABSTRACT

In recent years the interior point method (IPM) has became a dominant choice for solving large convex 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. The scalability of a parallel IPM is determined by several key issues such as exploiting parallelism due to sparsity of the problem, reducing communication overhead and proper load balancing.

In this paper we present an implementation of a parallel linear programming IPM workload and characterize its scalability on a 4-way Itanium® 2 system. We show a speedup of up to 3-times for some of the datasets. We also present a detailed micro-architectural analysis of the workload using VTune™ performance analyzer. Our results suggest that a good IPM implementation is latency-bound. Based on these findings, we make suggestions on how to improve the performance of the IPM workload in the future.

1. INTRODUCTION

Dramatic progress made in recent years in speed and robustness of convex optimization algorithms can be attributed to a combination of hardware and software improvements. Today, optimization problems involving millions of variables and constraints are solved routinely on a reasonably priced desktop workstation. This made optimization software an indispensable tool in numerous application domains, such as VLSI design, manufacturing, telecommunications, travel, etc.

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

In this work, we describe a shared-memory formulation of a scalable parallel linear programming IPM, which we call Interior Point Solver, or IPS. IPS combines a robust, publicly available interior-point predictor-corrector linear programming package, called PCx [4, 18], with a highly optimized parallel routines from Intel® Math Kernel Library (Intel® MKL) [9] and our own parallel implementation of several sparse-matrix routines.

We perform a scalability analysis of IPS on a 4-way Itanium® 2 shared-memory multiprocessor system. To establish the baseline, we compare the runtime of IPS against the industrial-strength solver called CPLEX® [8]. We show that for a number of challenging linear programs, our implementation runs at about half the speed of CPLEX. We then continue to present a detailed scalability analysis of the entire workload and of each parallel region.

We also perform a detailed micro-architectural characterization of IPS. We break down the CPU-time into independent components such as processor idle time, pipeline flushes and cache misses. We also analyze the bus transaction events and the bus bandwidth. We find that IPS is memory latency-bound rather than bandwidth-limited, which suggests that future performance improvements are likely to come from larger caches and hardware/software latency hiding techniques such as prefetching, multi-threading and code scheduling optimizations.

The remainder of the paper is organized as follows. Section 2 describes previous work on parallelization and performance analysis of interior point methods. Section 3 presents a high level overview of the application, while Section 4 gives details on the construction and implementation of IPS. Section 5 describes our experimental methodology and presents performance analysis of IPS on a 4-way Itanium 2 system. The final section presents the conclusions.

2. RELATED WORK

In this paper we focus on using the interior point method [24] to solve linear programming problems. The same framework can be used to solve other classes of convex optimization problems, with important practical applications, such as quadratic programming and second order cone programming [3]. Our implementation of the IPM is based on a primal-dual predictor-corrector algorithm introduced by Mehrotra [17].

Recent advances in building parallel solvers for symmetric and unsymmetric sparse linear systems of equations are central to this work. A new parallel distributed-memory multi-frontal approach, called MUMPS [1], uses a fully asynchronous approach with dynamic scheduling of the compu-

0-7803-8828-3/04/$20.00 © 2004 IEEE

tational tasks. Asynchronous communication was chosen to enable overlapping between communication and computation and ensures good scalability of the algorithm on systems with large number of processors.

PARDISO [22] is a parallel direct solver for sparse symmetric and structurally symmetric systems of equations designed for shared-memory multiprocessing systems. The algorithm uses a dynamic two-level scheduling scheme to significantly reduce synchronization events. It tries to reduce cache conflicts and interprocessor communication while maintaining good processor load balance and Level 3 BLAS performance. We describe the algorithm in more detail in Section 4.3. PARDISO now ships with Intel MKL. We choose it as the factorization engine for our interior point solver.

Lustig and Rothberg present results of parallelization of the CPLEX interior point solver on a shared-memory system in [7]. Rothberg *et al.* [19] use a detailed cache simulation to analyze and characterize the cache performance of different implementations of Cholesky factorization. To the best of our knowledge, no other prior work has been reported on a detailed micro-architectural study of the IPM on a commercial multiprocessor system.

The impact of fill-reducing ordering on large-scale linear programming problems is analyzed in [20].

3. INTERIOR POINT METHOD

The primal-dual predictor-corrector algorithm for linear programming simultaneously solves a primal and a dual linear problem. The primal linear problem is

$$\text{minimize } \mathbf{c}^T\mathbf{x} \quad \text{subject to } \mathbf{Ax} = \mathbf{b},\ \mathbf{x} \geq 0, \qquad (1)$$

where $\mathbf{c} \in \mathbf{R}^n$ and $\mathbf{b} \in \mathbf{R}^m$ are given and $\mathbf{A}$ is a matrix describing m constraints in n variables. The corresponding dual problem is

$$\text{maximize } \mathbf{b}^T\mathbf{y} \quad \text{subject to } \mathbf{A}^T\mathbf{y} + \mathbf{z} = \mathbf{c},\ \mathbf{z} \geq 0, \qquad (2)$$

where $\mathbf{y} \in \mathbf{R}^m$ is a vector of dual variables and $\mathbf{z} \in \mathbf{R}^n$. See [4, 24] for details.

Figure 1 outlines the kth iteration of the main optimization loop of interior point solver following [7]. The iteration starts with the primal and dual variables $(\mathbf{x}_k, \mathbf{y}_k, \mathbf{z}_k)$ from Equations (1) and (2). The algorithm proceeds to compute a new search direction $\mathbf{d}_s = (\mathbf{d}_{sx}, \mathbf{d}_{sy}, \mathbf{d}_{sz})$ (Step 8), and the new interior point $(\mathbf{x}_{k+1}, \mathbf{y}_{k+1}, \mathbf{z}_{k+1})$ is obtained in Step 10. The algorithm terminates when the optimal solution $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ is found. Note that, matrices $\mathbf{X}$ and $\mathbf{Z}$ are diagonal matrices formed by putting vectors $\mathbf{x}_k$ and $\mathbf{z}_k$ on the diagonal. Similarly, matrices $\mathbf{D}_{px}$ and $\mathbf{D}_{pz}$ are diagonal matrices formed by putting vectors $\mathbf{d}_{px}$ and $\mathbf{d}_{pz}$ on the diagonal (Steps 8.1 and 8.2).

Computationally intensive steps of the IPM iteration are

1. Matrix-matrix multiplication $\mathbf{M} = \mathbf{AQA}^T$ (Step 3)
2. Cholesky factorization $\mathbf{M} = \mathbf{LDL}^T$ (Step 4)
3. Solving for $\mathbf{d}_{py}$ and $\mathbf{d}_{sy}$ (Steps 5.1 and 8.1)
4. Matrix-vector products $\mathbf{Av}$ (Steps 1, 5.1, 8.1) and $\mathbf{A}^T\mathbf{v}$ (Steps 1, 5.2, 8.1) for different vectors $\mathbf{v}$.

Depending on the problem structure, either the Cholesky factorization or the matrix-matrix multiply are the most

1 Compute $\mathbf{r}_p = \mathbf{b} - \mathbf{Ax}_k$ 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 $\mathbf{d}_{py} = \mathbf{M}^{-1}[\mathbf{r}_p + \mathbf{AQ}(\mathbf{r}_d - \mathbf{XZe})]$
  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 $\mathbf{a}_p = \min\{-x_j / D_{xj}: D_{xj} < 0 \text{ and } j = 1 \ldots n\}$
  6.2 $\mathbf{a}_d = \min\{-z_j / D_{zj}: D_{zj} < 0 \text{ and } j = 1 \ldots n\}$
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 $\mathbf{d}_{sy} = \mathbf{M}^{-1}[\mathbf{r}_p + \mathbf{AQ}(\mathbf{r}_d + u\mathbf{e} - \mathbf{XZe} - \mathbf{D}_{px}\mathbf{D}_{pz}\mathbf{e})]$
  8.2 $\mathbf{d}_{sx} = \mathbf{Q}[\mathbf{A}^T\mathbf{d}_{sy} - u\mathbf{e} + \mathbf{XZe} + \mathbf{D}_{px}\mathbf{D}_{pz}\mathbf{e} - \mathbf{r}_d]$
  8.3 $\mathbf{d}_{sz} = u\mathbf{X}^{-1}\mathbf{e} - \mathbf{Ze} - \mathbf{Q}^{-1}\mathbf{D}_{sx}\mathbf{D}_{sz}\mathbf{e} - \mathbf{Q}^{-1}\mathbf{d}_{sx}$
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: One iteration of primal-dual IPM

time-consuming portions of the IPM iteration. For problems with many more columns than rows, the computation of $\mathbf{AQA}^T$ is dominant, but for dense problems, or problems with a large number of rows, the factorization time is dominant. However, in many instances, each of the other two routines listed above consumes a significant fraction of the execution time. It is therefore important to have an efficient parallel implementation for each of these routines as we discuss next.

4. WORKLOAD CONSTRUCTION

This section outlines the implementation details of our parallel shared-memory interior point solver. Section 4.1 starts by describing the software environment that we used to construct our workload. Section 4.2 discusses the preprocessing step which must be performed before the main optimization loop. Sections 4.3-4.6 present a parallel shared-memory implementation of the four main IPS routines.

4.1 Software Environment

IPS is based on PCx—an interior-point predictor-corrector linear programming package developed at Argonne National Laboratory in collaboration with Northwestern University [4, 18]. Our implementation of the Cholesky factorization and the solver routines uses a parallel sparse direct solver package, called PARDISO, developed at University of Basel [23], which is now included as part of Intel's Math Kernel Library. In addition, we implemented the routines for parallel sparse matrix-vector multiplication (MVM) and matrix-matrix multiplication (MMM).

To avoid the overhead of copying data between PCx and PARDISO, we replaced PCx's internal data structures for sparse matrices $\mathbf{A}$ and $\mathbf{M}$ with PARDISO's compressed row storage (CRS) format. This format is also used for the MVM and MMM routines.

Problem	Rows	Columns	Nonzeros in A	% nonzeros
dfl001	6071	12230	35632	0.0480
fome13	48568	97840	285056	0.0060
qismondi	18262	23211	136324	0.0322
ken-18	105127	154699	358171	0.0022
osa-60	10280	232966	1397793	0.0584
pds-10	16558	48763	106436	0.0132
pds-20	33874	105728	230200	0.0064

Table 1: The statistics of the problems used in our experiments. The last column shows the percentage of nonzero entries in matrix A.

Problem	CPLEX iterations	CPLEX runtime	IPS iterations	IPS runtime	Factor FP ops
dfl001	24	14	39	28.08	4.85E+08
fome13	27	131.73	40	237.60	3.97E+09
qismondi	8	574.96	39	1564.68	1.45E+11
ken-18	29	25.01	30	94.80	1.49E+08
osa-60	22	16.18	32	29.76	1.22E+05
pds-10	51	11.47	35	27.30	1.10E+08
pds-20	40	53.33	44	146.52	7.60E+08

Table 2: The run-time comparison of IPS and the CPLEX barrier LP solver.

Table 1 summarizes the statistics for the datasets used in our experiments. They mostly come from the standard NETLIB [5] test set and represent realistic linear models from several application domains. The problems are fairly large and difficult and some of them are very sparse.

Table 2 reports the total number of iterations to converge to the optimal solution and the total run-time of IPS and CPLEX on our test datasets. The run-times were measured using the system described in Table 3. As can be seen from the table, our implementation compares quite favorably with a highly optimized, industrial strength solver. IPS is on average only two times slower than CPLEX. This makes us confident that our application will not diverge significantly from what we would expect from a highly optimized software.

The last column reports a number of floating point operations required to factor matrix **M**. It shows a correlation between the amount of work required to factor the constraint matrix and the time to solve the corresponding LP.

4.2 Preprocessing

Proper initialization is crucial for ensuring a good performance of the IPM. The goal of this step is to read the linear model, convert it into a sparse matrix representation **A**, analyze the sparsity structure of **M** and **L**, and determine a reasonable way to partition independent tasks across processors in the system. The initialization involves several steps, which we outline subsequently.

Presolver

The *presolver* enhances efficiency and robustness of the linear programming algorithm by eliminating empty and duplicate rows and columns. The algorithm performs multiple passes through the data until no further improvements are found. The computational cost of the presolver is negligible compared to the cost of one iteration of IPM. Our workload uses the presolver provided by PCx.

Matrix Ordering

A typical constraint matrix **A** is very large and sparse. See Table 1 for sample statistics. Applying Cholesky factorization directly to the matrix **M** can significantly increase the number of non-zero entries in the factor matrix **L**. In case of very sparse problems, this has highly adverse effects on computation and memory requirements for the factorization. Through an *ordering* of row and columns of matrix **M**, it is possible to reduce the amount of fill-in. While the problem of finding an optimal matrix ordering is NP-complete, many ordering techniques exist that perform well in practice.

Beside minimizing fill-ins, a good matrix ordering can also increase the amount of parallelism of a given factorization problem, since it determines the load balancing and the task scheduling during parallel factorization. In our implementation, fill-in reducing permutation matrix **P** is computed using the *nested dissection* algorithm [14]. The other popular choice for fill-reducing ordering is the *minimum degree* algorithm [15]. These ordering algorithms are included as part of MKL implementation of Cholesky factorization. While the time to compute an ordering can be significant, this operation needs to be done only once before the main optimization loop. Therefore, its cost is amortized over the course of running the interior point solver. Although there exists parallel implementations of the ordering algorithms [13], they are currently not included into MKL.

Elimination Tree

For a given ordering of matrix **M**, there exists an *elimination tree*—a directed acyclic graph with nodes corresponding to the columns of the matrix and the edges marking the dependence relations among columns during the factorization [16]. Column j of **M** can be processed only after the columns that are descendants of j in the elimination tree have been processed. However, columns in different subtrees correspond to the independent tasks that can be executed in parallel. Thus, in parallel implementation of Cholesky factorization, the elimination tree helps determine which columns can be factorized independently.

Note that different matrix orderings result in different elimination trees and therefore different amount of parallelism. See [12] for an analysis of the impact of the matrix ordering on load balancing and fill-in and [6] for an analysis of its impact on memory usage.

Symbolic Factorization

The *symbolic factorization* pre-computes locations of nonzeros in the factor matrix **M**. Since the non-zero structure of **M** remains fixed for every iterations of IPM, we can allocate all required storage for **M** prior to the main optimization loop. This procedure also identifies the *supernodes* in factorization, which constitute groups of columns with identical nonzero structure. Grouping of columns into supernodes is an important source of instruction and task-level parallelism. Each supernode can be represented and stored as a dense matrix and take advantage of Level 3 BLAS operations [19]. To take better advantage of the parallelism, the code performs so called *node amalgamation* [2], which introduces extra fill-ins in the matrix (by treating some zero elements as non-zeros) in order to increase the width of the supernodes.

4.3 Parallel Cholesky Factorization

As mentioned earlier, we have chosen PARDISO as the factorization engine for our interior point solver. PARDISO is designed to exploit different sources of parallelism existing in direct methods of solving sparse systems of equations [22]. Elimination tree parallelism is relatively easy to achieve but, in practice, it often accounts for a relatively small improvement in concurrent performance, since much of the factorization cost is located in the large nodes near the root of the tree.

To explore more parallelism and reduce load imbalance, PARDISO implements a two-phase parallel scheduling algorithm. In the first parallel phase of scheduling, PARDISO statically maps independent subtrees of the elimination tree to different processors. Since the subtrees are independent, each processor internally factorizes the nodes in its own subtree in parallel with other processors, hence there is no interprocessor communication. When each processor is finished with its own subtree, it waits on a barrier for other processors to finish. As soon as all processors reach the barrier, the second parallel phase begins. In this phase, each processor that has completed (factorized) a number of supernodes stores them into the dynamic queue. Each processor monitors the queue, picks up a new factorized supernode as soon as it becomes available, and performs an external update to its ancestor supernodes. Note that the ancestor may reside on a different processor. Therefore, in contrast to the first phase, this phase may incur inter-processor communication. The processor that performs the last update to the supernode puts the completed supernode into the work queue. Since the queue is shared among the processors, it is guarded by a *mutex* so that only one processor can access it at a time. The mutex may result in additional waiting time for processors that are trying to access the queue at the same time.

4.4 Parallel Forward and Backward Solve

Sparse triangular solver uses Cholesky factorization $\mathbf{LL}^T$ to the system of equations $\mathbf{LL}^T\mathbf{y} = \mathbf{x}$. The forward and backward substitution performed by the solver is difficult to parallelize. These tasks have a low computation-to-communication ratio with inherent recursion. Still, some parallelization is possible. PARDISO provides a parallel triangular solver which works together with its Cholesky factorization. The solver uses the same supernode partitioning and the same elimination tree to order the computation as the factorization [21].

4.5 Parallel Matrix-Matrix Multiply

Our software forms the symmetric matrix $\mathbf{M} = \mathbf{AQA}^T$ one element at a time. The element m_{ij} of $\mathbf{M}$ is computed as

$$m_{ij} = \mathbf{A}_i\ \mathbf{Q}\ \mathbf{A}_j^T = \sum_{k=1}^{n} a_{ik}\ q_{kk}\ a_{jk},$$

where $\mathbf{A}_i$ and $\mathbf{A}_j$ are the rows i and j of matrix $\mathbf{A}$, respectively; a_{ij} is the element of $\mathbf{A}$ in ith row and jth column, and q_{kk} is the kth diagonal element of $\mathbf{Q}$.

Since each element of matrix $\mathbf{A}$ is computed independently, this routine is highly parallel. The load balancing during the parallel computation is achieved by evenly distributing the rows of matrix $\mathbf{M}$ among the processors based on the computation required to evaluate all non-zero elements m_{ij} in row i. This load assignment is performed statically before the first iteration of the IPM. Since the non-zero structure of matrix $\mathbf{M}$ remains fixed at every iteration, the assignment does not have to be recomputed.

The MMM routine maintains one work vector $\mathbf{w}$ of length m per processor. The elements of $\mathbf{A}_i\mathbf{Q}$ are scattered into $\mathbf{w}$. The indices of non-zero entries of $\mathbf{A}_i$ are used to determine the appropriate locations in $\mathbf{w}$. The indices of non-zero entries of $\mathbf{A}_j$ are then used to compute the dot-product $\mathbf{wA}_j$ to obtain m_{ij}.

Processor Speed	1GHz
Number of Processors	4
L1 caches	16 KB I-cache, 16 KB D-cache, hit latency: 1 cycle
L2 cache	256 KB unified, hit latency: 5-9 cycles
L3 cache	3 MB unified, hit latency: 16+ cycles miss latency: approximately 210 cycles
System Chipset	Intel E8870
System Bus Speed	400 MHz
Maximu Bus Bandwith	6.4 GB/s
Memory Size	8 GB

Table 3: Summary of the base system configuration.

4.6 Parallel Matrix-Vector Multiply

At each iteration of the interior point solver, matrix-vector multiplications $\mathbf{Ax}$ and $\mathbf{A}^T\mathbf{x}$ are computed for three different values of $\mathbf{x}$. This computation can be expensive for large problems with many non-zeros.

When computing $\mathbf{Ax}$, we partition the rows of matrix $\mathbf{A}$ into submatrices $\mathbf{S}_i$ in such a way as to maximize the load balance. Each product $\mathbf{S}_i\mathbf{x}$ can be computed entirely independently. When computing $\mathbf{A}^T\mathbf{x}$, we again distribute the rows of $\mathbf{A}$ among the processors. In this case, however, each processor computes $\mathbf{S}_i^T\mathbf{x}_i$, where vector $\mathbf{x}_i$ corresponds to the row partition of submatrix $\mathbf{S}_i$. When all processors are finished, the local work vectors are added into the main result vector in parallel.

5. WORKLOAD CHARACTERIZATION

5.1 Experimental Methodology

This section presents several characteristics of IPS workload obtained using hardware performance counters on a 4-way Itanium 2 Dell 7250 server with Intel E8870 chip-set. The Itanium 2 processor, which is an implementation of the IPF architecture, is an in-order machine. It can issue up to 6 instructions including 4 memory and 2 floating point instructions in a single cycle from its 11-issue ports. The Itanium 2 has a low-latency, high bandwidth memory system comprising three levels of cache hierarchy. The L1D cache is a write-through, no-write allocate cache with one-cycle load latency. The L1D cache serves only integer loads. All floating point load instructions receive data directly from the unified L2 cache. The L2 cache is arranged in an array of 16 banks with a latency of 5,7 or 9 cycles. Both L2 and L3 use MESI protocol to enforce cache coherency. The main system parameters are summarized in Table 3.

The Itanium 2 system has a rich set of performance events, with four performance counters measuring over 200 unique events. The counters can be selected by setting the appro-

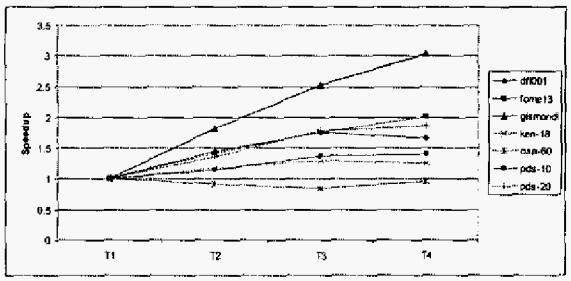

Figure 2: Scalability of a single iteration of the IPS for 1, 2, 3, and 4 processors.

priate control registers. The performance monitoring system on Itanium 2 is non-invasive and does not affect the execution of the workload. We use the Vtune performance analyzer to setup and configure the performance counters. The Vtune analyzer is a system wide measurement tool that can gather performance data at thread granularity.

We use Intel C and Fortran compilers version 8.0 to compile our workloads. For all experiments, we run IPS for two iterations prior to taking measurements. This warm up period allows the workload to warm-up the data caches and enter into steady-state mode. All datasets were run for ten iterations during the measurement phase. The results were averaged and reported for a single iteration of IPS. We instrumented IPS with the Vtune API to characterize all parallel regions in our workload.

5.2 Scalability of Parallel Regions

Figure 2 reports the speedup of IPS for the test datasets on 2, 3, and 4 processors. The scalability appears to be correlated with the amount of work required to factor the constraint matrix **M** (see the discussion for Table 2 in Section 4.1). This is encouraging as it seems to suggest that parallel computation improves the performance on harder problems.

Figure 3 shows the breakdown of total execution time into the four parallel regions discussed earlier and the remaining serial code. For each dataset, we show four bars corresponding to 1, 2, 3, and 4 processors, respectively. Each bar is broken into five parts, one for each execution region.

The factorization routine achieves a good scalability on many datasets, which correlates well with the scalability results of PARDISO reported in [21]. Unfortunately, the solver scales poorly compared to the factorization and it emerges as the main obstacle in scalability of IPS on a large number of processors. We plan to address the performance of the solver in the near future.

Contrary to the expectations, MVM takes more time than MMM in almost all cases. We conjecture that the thread creation/termination and the synchronization overhead are the culprit. After the solver, the MVM routine is the next candidate for performance improvement.

5.3 CPU Cycle and Instruction Breakdown

CPU Cycle Breakdown

This section focuses on analyzing the micro-architectural behavior of our workload. We begin by decomposing the available CPU cycles into two primary contributing components, the idle time and the runtime. We define and measure the idle time as the difference between the total execution time and the CPU time. A processor is idle when a thread is blocked on synchronization or a barrier. Itanium 2 performance counters can accurately decompose the runtime stalls into five categories, *Flush*, *RSE*, *L1D*, *FE*, and *EXE*. The time spent retiring instructions is categorized as *Work*. Table 4 defines the components. A more detailed description can be found in [11].

Itanium Processor CPU Cycle Component	Description
Work	The time spent executing instructions
Flush	The stall time due to flushing the pipeline after branch misprediction
RSE	The stalls due to the register stack engine
L1D	The stalls due to the L1 data cache and TLB
FE	The stalls in the backend pipeline caused by stalls in the front end. Stall sources include I-cache miss stalls and TLB stalls
EXE	The stalls due to L2 cache misses, L3 cache misses

Table 4: CPU cycle component definitions.

Figure 4 shows the CPU cycle time breakdown. For each dataset, three stacked bars are shown corresponding to 1, 2 and 4 processors. Each bar segment shows the contribution of that component toward the total CPU time. We see that on average 50% of the cycle time is *Work*, i.e., at least one instruction is retired during that cycle. Of all the stall events the *EXE* stalls, which occur primarily due to L3 misses, account for more than 50% of all stall cycles on average.

For most datasets, the *EXE* stalls decrease as we go from 2 to 4 processors despite an increased ratio of coherence misses. We speculate that this is because for 4 processors the reduction in capacity misses overcomes the increase in coherence misses. In the future, we plan to do additional analysis to verify this claim.

Unfortunately, we were not able to separate 'busy waiting' from *Work* using VTune analizer because a substantial part of IPS is implemented using OpenMP directives. However, we found that looking at the increase in the number of executed instructions when the number of processors is increased, as reported at the top of Figure 5, gives a good estimate of the amount of busy waiting.

We also see that the pipeline flush stalls due to branches increase with the number of processors for all datasets. For example, for *pds-10* this number doubles when we go from 2 to 4 processors. This behavior could be attributed to the dynamic load balancing in the factorization routine. As mentioned in Section 4.3 and Section 4.4, load balancing is implemented using a centralized queue that supplies dynamically created tasks to each processor. This dynamic change in the computation on each iteration must be adversely affecting the training of the branch predictor.

As the number of processors increases, the idle time increases. For example, for *osa-60* the idle time constitutes almost 20% of the total execution time on 4 processors. Since in this case the MVM routine dominates the execution time, we conclude that the idle time is due to the thread creation/termination and the synchronization overhead, as explained in Section 5.2. (There is almost no idle time in the factorization and the solver: each thread performs a large

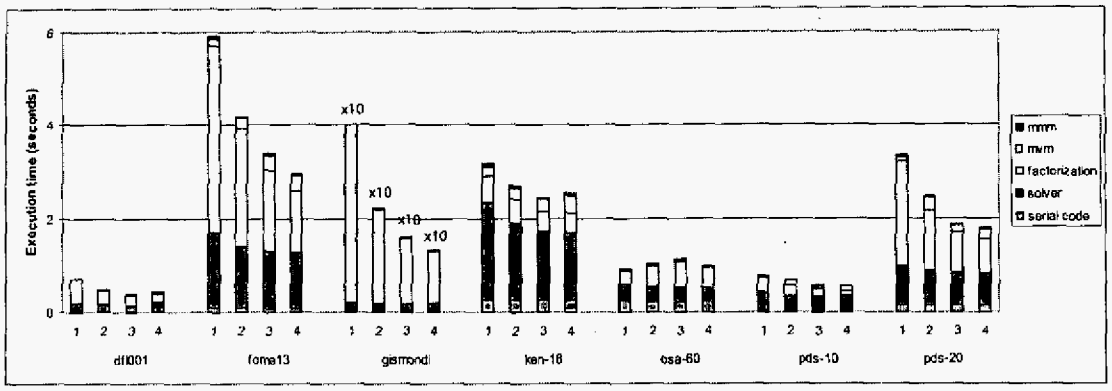

Figure 3: Execution time breakdown into individual parallel regions.

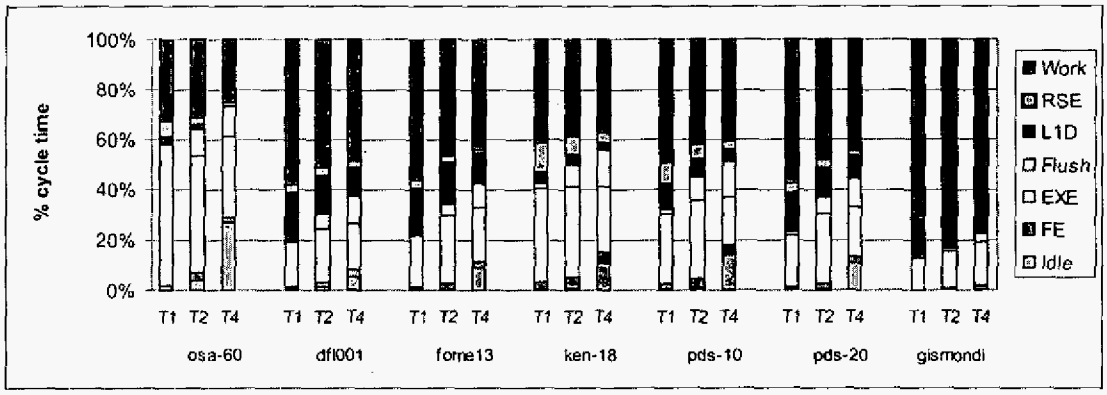

Figure 4: Breakdown of CPU cycles.

amount of computation and all synchronization in these two routines is performed using busy-waiting.)

Instruction Breakdown

Figure 5 shows instruction breakdown for each dataset. We split the instructions into integers, loads, stores, floating-points and branches. For each dataset we show three bars corresponding to 1, 2, and 4 processors. Above each bar we report the total executed instructions (in billions).

Using the existing Itanium 2 performance counters, we cannot separate FP and integer loads. However, given that our application is FP intensive and assuming that each FP operation requires 2 FP loads for its two source operands, the percentage of FP loads can be approximated from the number of FP operations.

We see that the number of instructions increases with the number of processors. For example, in the case of *pds-10*, the total number of instructions more than doubles when we move from 1 to 4 processors. This increase is largely due to busy-waiting in the two time-dominating routines—the factorizer and the solver.

5.4 Memory System Characterization

Cache Misses Behavior

Figure 6 shows the data miss rates per memory reference for the 3 levels of caches on Itanium 2: L1D, L2 and L3. For all the datasets, the L2 miss rate is higher than the L1 miss rate because the floating point loads on Itanium 2 system bypass the L1 cache and get data directly from L2. The L2 miss rate is the highest for *gismondi* because, as is shown in Figure 5, this dataset has the largest number of loads.

Although the L3 miss rate is small in absolute terms, it has the highest impact on the overall performance of our application, since an average L3 miss costs 210 cycles [10].[1]

Also, the ratio of L3 misses to L2 misses seems quite high—on average, the L3 cache is catching only about 25% of the L2 misses. In the future, we plan to collect the L3-cache-miss statistics per routine to get a better idea how these are distributed in the application and if there is a way of reducing them through prefetching.

Coherence Traffic

Figure 7 shows the breakdown of L3 cache misses based on where the miss was serviced according to MESI cache coherence protocol. Each bar represents the breakdown of L3 misses into five distinct categories. R-MEM represents the L3 read misses serviced from memory. R-S/E represents the read misses serviced from a remote cache with the block in

[1]Take for example *ken-18 running on two processors*. A simple calculation reveals that an L3 miss occurs every 500 instructions. The impact is even higher if we consider that Itanium 2 can issue up to 6 instructions in parallel.

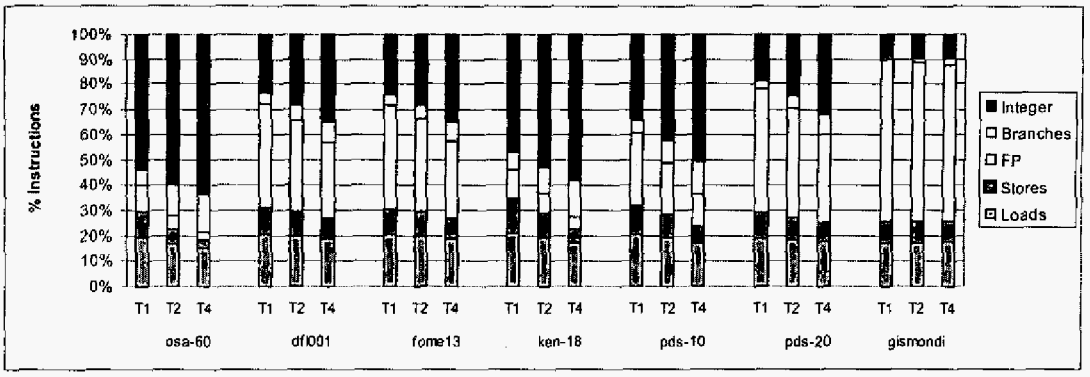

Figure 5: Instruction Breakdown. Above each bar is the total number of executed instructions (in billions).

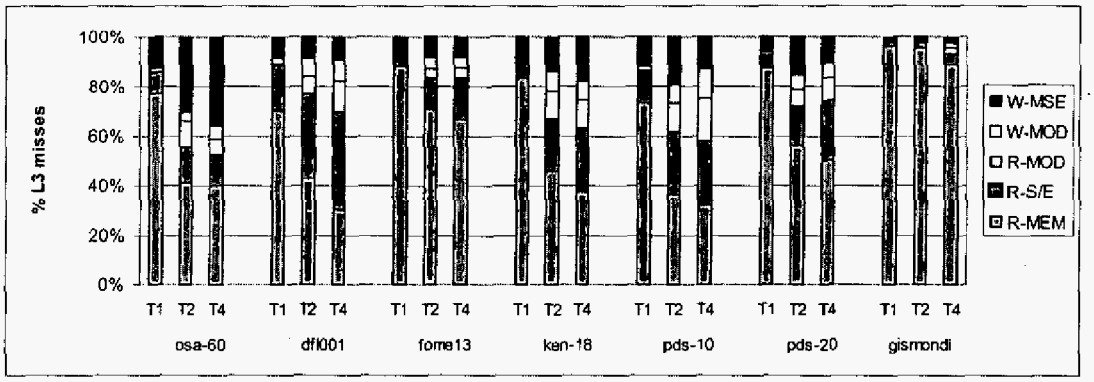

Figure 7: Breakdown of L3 cache misses based on the source servicing the miss.

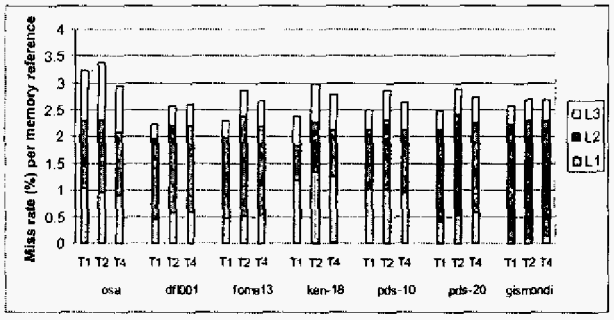

Figure 6: L1, L2 and L3 data miss rates.

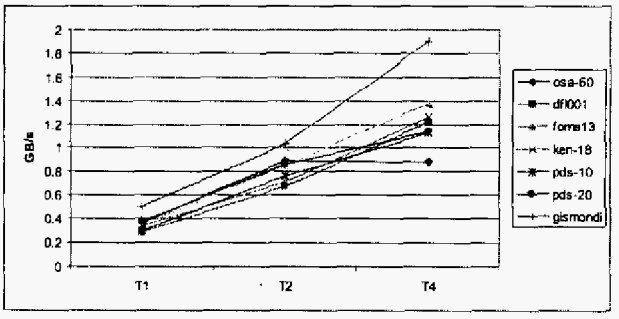

Figure 8: Sustained Memory Bandwidth

either shared or exclusive state. R-MOD represents the read misses from a remote cache with the block in the modified state. W-MOD represents the L3 write misses serviced from a remote cache with the block in a modified state. W-MSE represents the L3 write misses serviced either from memory or a remote cache with the block in either exclusive or shared state. Currently, we cannot separate these latter misses into more distinct categories.

Figure 7 indicates that memory traffic decreases (R-MEM) and coherence traffic increases (sum of R/SE, R-MOD, W-MOD, and part of W-MSE) with the number of processors. However, for most datasets, this change is less dramatic when we go from 2 to 4 processors, indicating that 2 processors have enough aggregate L3 cache capacity to fit most of their working set.

Note that memory traffic does not decrease for *gismondi*. Most likely, the working set for its supernodes is too big to fit into the L3 cache, hence it is evicted from the L3 cache after each iteration, increasing memory, but not coherence traffic.

Memory Bandwidth

Figure 8 shows the memory bandwidth usage of IPS for each dataset on 1, 2 and 4 processors. As expected, the bandwidth increases with the number of processors. However, the bus remains under-utilized: even on four processors only less than 25% of the maximum available bandwidth (6.4 GB/s) is used.

This observation combined with cache-miss data (Figure 6) might suggest that IPS is latency-bound, rather than bandwidth-limited. However, in the future we would like to compute the bandwidth usage over time, since we suspect that some parallel regions might be using much higher bandwidth than others. Only then will we be able to definitively establish the impact of bandwidth on performance.

6. CONCLUSIONS

In this work we have described our implementation of a scalable parallel interior point method and studied its performance on a 4-way Intel Itanium 2 shared-memory system.

We performed a detailed scalability study of the application and shown that, for certain datasets, the application achieves up to 3X speedup on 4 processors. Scalability seems to correlate well with problem sizes—the harder problems scale better.

Additionally, we performed a detailed micro-architectural analysis of the application. Our results indicate that a good implementation of interior-point method is latency-bound rather than bandwidth-limited. This suggests that the future performance improvements to interior-point methods are likely to come from larger caches and hardware/software latency hiding techniques, such as prefetching, multi-threading and code scheduling optimizations.

7. ACKNOWLEDGMENTS

We would like to thank our colleagues: Skip Macy for help studying scalability of CPLEX, and Stephen Skedzielewski for editing the paper. We also thank Murali Annavaram, Richard Hankins from Intel Research and Richard Greco from Enterprise Product Group for their help with VTune performance analysis. We are greatly indebted to Edward Rothberg from ILOG, Inc. for his help and guidance throughout this work. Finally, we thank the reviewers, and other members of the Architecture Research Lab for their useful insights and suggestions.

8. REFERENCES

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

[2] C. Ashcraft and R. Grimes. The Influence of Relaxed Supernode Partitions on the Multifrontal Method. *ACM Trans. on Math. Software*, 15:291–309, 1989.

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

[4] 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.

[5] D.M. Gay. *Electronic Mail Distribution of Linear Programming Test Problems*. Mathematical Programming Society COAL Newsletters, 1988.

[6] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of Reordering on the Memory of a Multifrontal Solver. *Parallel Comput.*, 29(9):1191–1218, 2003.

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

[8] ILOG CPLEX. `http://www.ilog.com/products/cplex`.

[9] Intel Corporation. Math Kernel Library version 7.0. `http://www.intel.com/software/products/mkl/index.htm`.

[10] Intel Corporation. *Introduction to Microarchitectural Optimization for Itanium 2 Processors Reference Manual*. Intel Reference Manual, Document Number 251464-001, 2002.

[11] S. Jarp. *A Methodology for using the Intel Itanium 2 performance counters for bottleneck analysis*. HP Laboratories Technical Report, August 2002.

[12] G. Karypis, A. Gupta, and V. Kumar. A Parallel Formulation of Interior Point Algorithms. In *Proceedings Supercomputing '94 (Washington, D.C., USA, November 1994)*, pages 204–213. IEEE Computer Society, New York, NY, USA, 1994.

[13] G. Karypis and V. Kumar. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95, 1998.

[14] G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[15] J. W. H. Liu. Modification of the Minimum-Degree Algorithm by Multiple Elimination. *ACM Trans. Math. Softw.*, 11(2):141–153, 1985.

[16] J. W. H. Liu. The Role of Elimination Tree in Sparse Factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[17] S. Mehrotra. On The Implementation of a Primal-Dual Interior Point Method. *SIAM Journal on Optimization*, 2(4):575–601, November 1992.

[18] Optimization Technology Center, Argonne National Laboratory and Northwestern University. PCx LP Solver. `http://www-fp.mcs.anl.gov/otc/Tools/PCx`.

[19] E. Rothberg and A. Gupta. *Techniques for Improving the Performance of Sparse Matrix Factorization on Multiprocessor Workstations*. Department of Computer Science Technical Report STAN-CS-90-1318, Stanford University Stanford, California 94305, June 1990.

[20] E. Rothberg and B. Hendrickson. Sparse Matrix Ordering Methods for Interior Point Linear Programming. *INFORMS J. on Computing*, 10(1):107–113, 1998.

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

[22] O. Schenk and K. Gärtner. Two-Level Dynamic Scheduling in Pardiso: Improved Scalability on Shared Memory Multiprocessing Systems. *Parallel Computing*, 28(2):187–197, 2002.

[23] University of Basel. PARDISO Direct Sparse Solver. `http://www.computational.unibas.ch/cs/scicomp`.

[24] S. J. Wright. *Primal-Dual Interior-Point Methods*. SIAM, 1997.