Issues In Implementing The Primal-Dual Method for SDP

Brian Borchers
Department of Mathematics
New Mexico Tech
Socorro, NM 87801
borchers@nmt.edu

Outline

1. Cache and shared memory parallel computing concepts.
2. Key steps in the the primal-dual interior point method.
3. Parallelizing the primal-dual interior point method.
4. Computational results.
5. Conclusions.

Memory Latency And Caches

- Latency is simply the time that it takes to retrieve a number or other data from a computer's memory.
- With today's machines, a processor can typically execute one hundred instructions in the time that it takes to retrieve a floating point number from main memory.
- To overcome this mismatch of processor speed and memory latency, modern microprocessors incorporate small low latency memories called caches. Data are prefetched into cache memory so that they will be available when the processor needs the data. There may be two or even three levels of cache memory in a system.
- A typical level one cache has a latency of one nanosecond, while a level two cache might have a latency of five nanoseconds, and main memory might have a latency of 50 to 100 nanoseconds.

Memory Bandwidth

- Bandwidth is the rate at which data can be transferred from main memory into a cache or from one level of cache to a lower level of cache.
- On today's machines it's quite common for a processor to exhaust the available bandwidth while working on very large data sets, because the rate at which instructions are executed is faster than the rate at which data can be moved from main memory into the cache and to the processor.
- For problems with smaller data sets that reside in cache, main memory bandwidth is typically not a limiting factor.

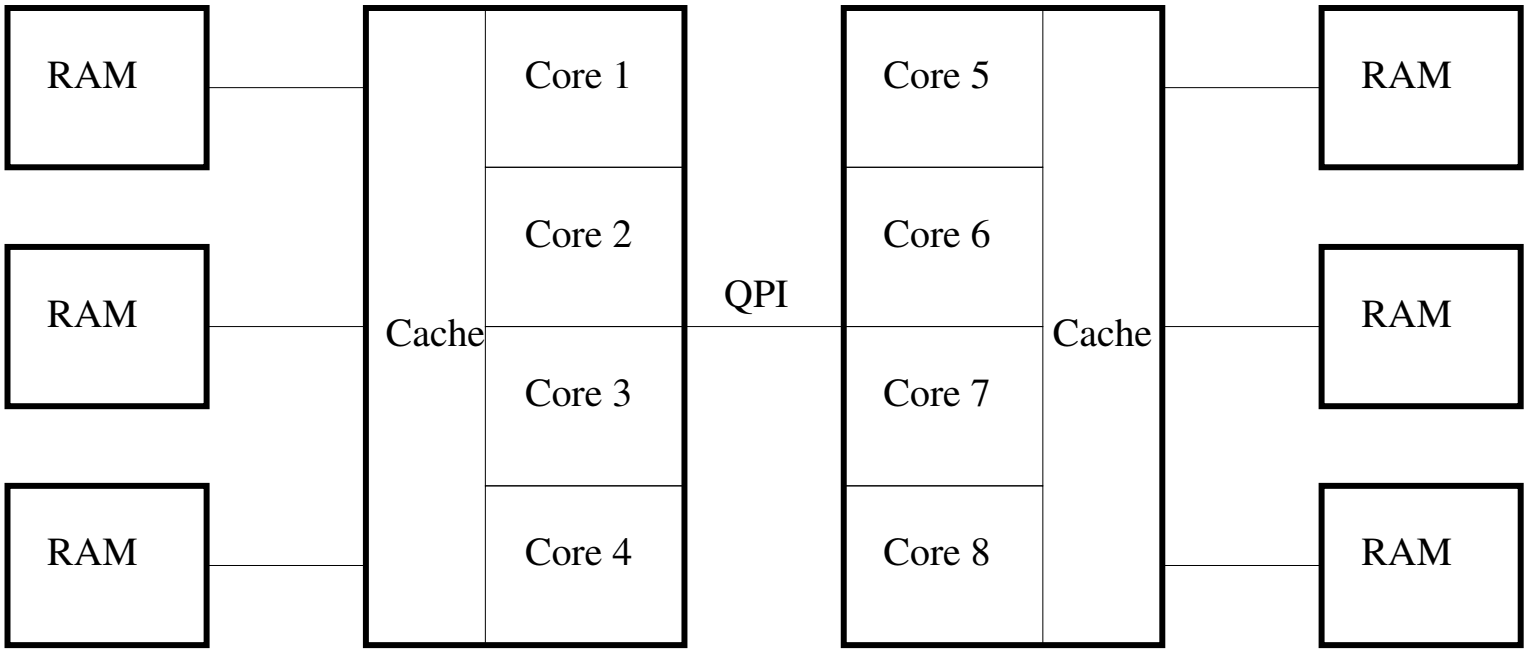
The Importance Of Higher Level Operations

- In numerical linear algebra with n element vectors most operations take $O(n)$ time and operate on each element of the vectors exactly once. For these level one operations it's not possible to reuse data in the cache.
- For operations on n by n matrices such as matrix addition or matrix vector multiplication the operations take $O(n^2)$ time and operate on $O(n^2)$ elements of the matrices. For these level two operations there is also no opportunity to reuse data in the cache.
- For $O(n^3)$ operations on n by n matrices such as matrix multiplication or Cholesky factorization, we access $O(n^2)$ data and perform $O(n^3)$ operations, so there are opportunities to reuse data in the cache, particularly if we use algorithms that operate on blocks of the matrices.

Shared Memory Architecture

- In recent years Intel and AMD have begun producing microprocessor chips that have two or four microprocessor cores on a single chip. Computers may have 1, 2, or more of these multicore processor chips.
- Each processor core has shared access to all of the memory. Processes cooperate in performing operations on data in the shared memory.
- Programs for these shared memory systems can be written using C or Fortran with parallel processing extensions such as POSIX Threads (pthreads) and OpenMP.
- Contrast this with distributed memory Beowulf clusters, in which programs running on separate computers communicate with each other by sending messages across a network.

Intel's Shared Memory Architecture



Parallel Efficiency And Amdahl's Law

- A classic measure of parallel performance is the speedup, obtained by dividing the running time on a single processor by the running time on N processors.
- Ideally, we'd get speedup of $S = N$ by using N processors.
- The parallel efficiency of a code is the percentage of this N fold speedup that we actually attain.
- If P is the fraction of the run time of a single processor program that can be parallelized with speedup S, then the overall speedup is

$$\frac{1}{(1-P) + \frac{P}{S}}.$$

- Speedups beyond $1/(1-P)$ simply aren't possible no matter how many processors we use.

The SDP Problem

$$\begin{array}{lrcl} \max & \text{tr}\,(CX) & & \\ (P) & A(X) & = & b \\ & X & \succeq & 0 \end{array}$$

where

$$A(X) = \begin{bmatrix} \text{tr}\,(A_1X) \\ \text{tr}\,(A_2X) \\ \cdots \\ \text{tr}\,(A_mX) \end{bmatrix}.$$

Note that

$$\text{tr}(CX) = \sum_{i=1}^{n}\sum_{j=1}^{n} C_{i,j}X_{j,i} = \sum_{i=1}^{n}\sum_{j=1}^{n} C_{i,j}X_{i,j}.$$

The Dual Problem

$$
\begin{array}{lrcl}
\min & b^T y & & \\
(D) & A^T(y) - Z & = & C \\
 & Z & \succeq & 0
\end{array}
$$

where

$$A^T(y) = \sum_{i=1}^{m} y_i A_i.$$

The Algorithm

- CSDP implements a predictor–corrector variant of the primal-dual interior point method of Helmberg, Rendl, Vanderbei, and Wolkowicz (1996.) This method is also known as the HKM method, since the same algorithm was discovered by two other groups of authors (Kojima et al. 1997; Monteiro and Zhang, 1997.)
- CSDP uses an infeasible interior point version of the HKM method.
- The basic idea is to apply Newton's method to a system of equations that can be thought of as a perturbed version of the KKT conditions for the primal/dual SDP's or the KKT conditions for a pair of primal and dual barrier problems.

The Algorithm

- The equations for the Newton's method step can be reduced to

$$O\Delta y = A(Z^{-1}(C - A^T(y) + Z)X) + A(\mu Z^{-1}) - b.$$

  Here

$$O = \left[\begin{array}{ccc} A(Z^{-1}A_1X) & \ldots & A(Z^{-1}A_mX) \end{array} \right]$$

  is a symmetric and positive definite matrix.

- This is similar to the fully reduced system of equations in interior point methods for linear programming, except that O is typically fully dense.

The Algorithm

- We can also write $O_{i,j}$ as

$$O_{i,j} = \text{tr}\,(A_i Z^{-1} A_j X).$$

- If a constraint matrix A_j is dense, then it may be worthwhile to compute $Z^{-1}A_jX$ using a dense matrix multiplication routine and then apply the A operator to get the jth column of O.
- However, if the A_i and A_j matrices are both sparse, it can be less expensive to compute $O_{i,j}$ by directly adding up the contributions to the trace from the nonzero entries of A_i and A_j.

Computational Complexity

- Multiplying matrices of size n takes $O(n^3)$ time.
- Factoring matrices of size n takes $O(n^3)$ time.
- For dense constraint matrices, constructing the Schur complement matrix takes $O(mn^3 + m^2n^2)$ time.
- For sparse constraint matrices with $O(1)$ entries, constructing the Schur complement matrix takes $O(mn^2 + m^2)$ time.
- In practice, most problems have $m > n$ and sparse constraint matrices.
- Thus we would expect the most time consuming steps in the algorithm to be the computation of the elements of the Schur complement matrix and the Cholesky factorization of this matrix.

Computational Results

- Our tests were performed on a Dell 5500 workstation with two 2.4 GHz quad core Intel Xeon E5530 processors.
- The four cores in each Xeon processor share an 8 megabyte level 3 cache.
- Each core has its own 256K byte level 2 cache and separate 32K byte level 1 code and data caches.
- The system has 12 gigabytes (6x2 Gbyte DIMMS) of DDR3-1066 RAM.
- The QPI between the two processor chips runs at 5.86 Gtrans/sec.

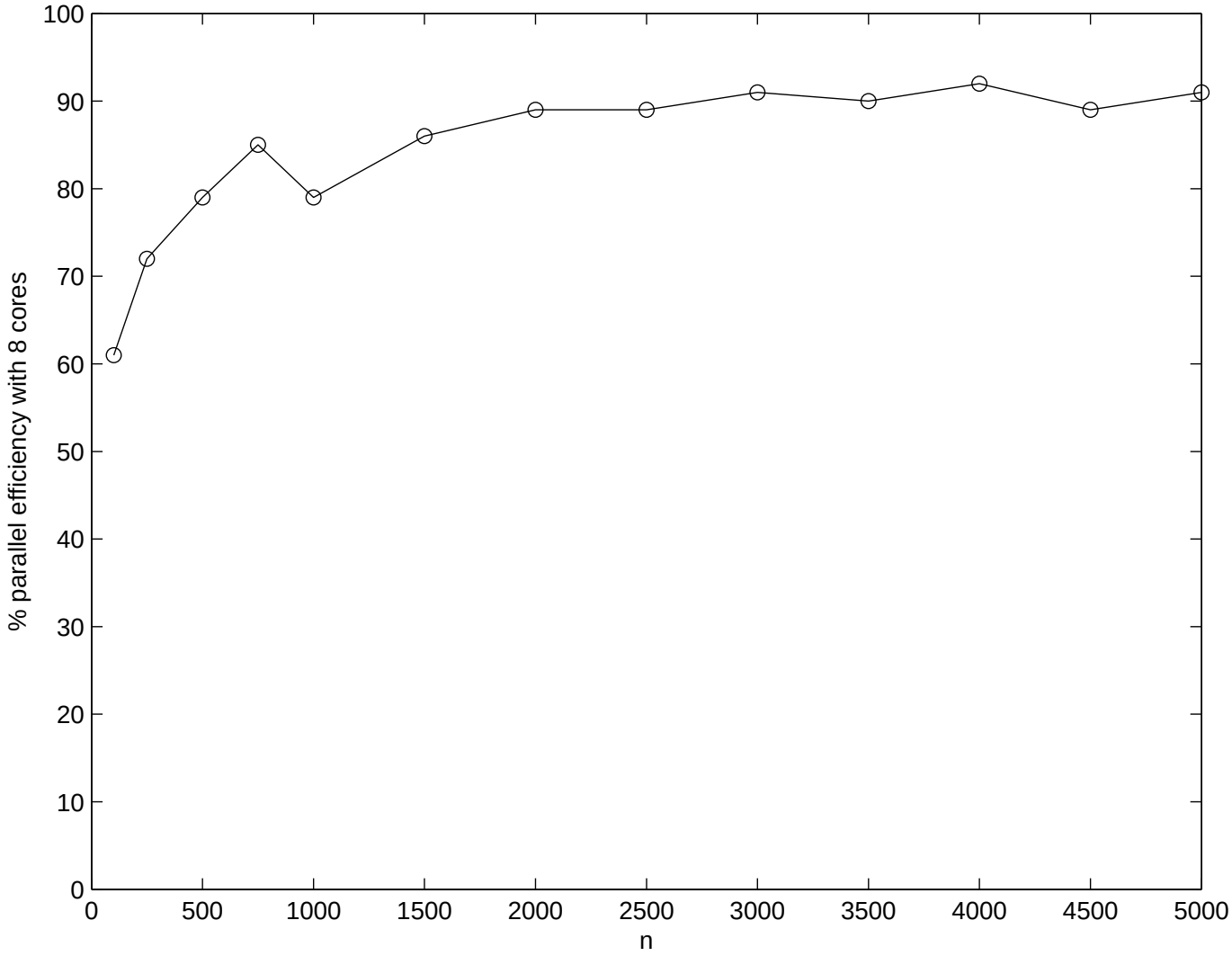
Computational Results

- The following tests were run under Ubuntu Linux. The codes were compiled using Intel's icc 11.1 with OpenMP. BLAS/LAPACK routines from Intel's MKL were used.
- Each code was tested with one, two, four, and eight processor cores.
- Wall clock times and parallel speedups and efficiencies were measured for each code.
- For the SDP test problems we computed CPU times and efficiencies for the major steps of the algorithms as well as the total run time.

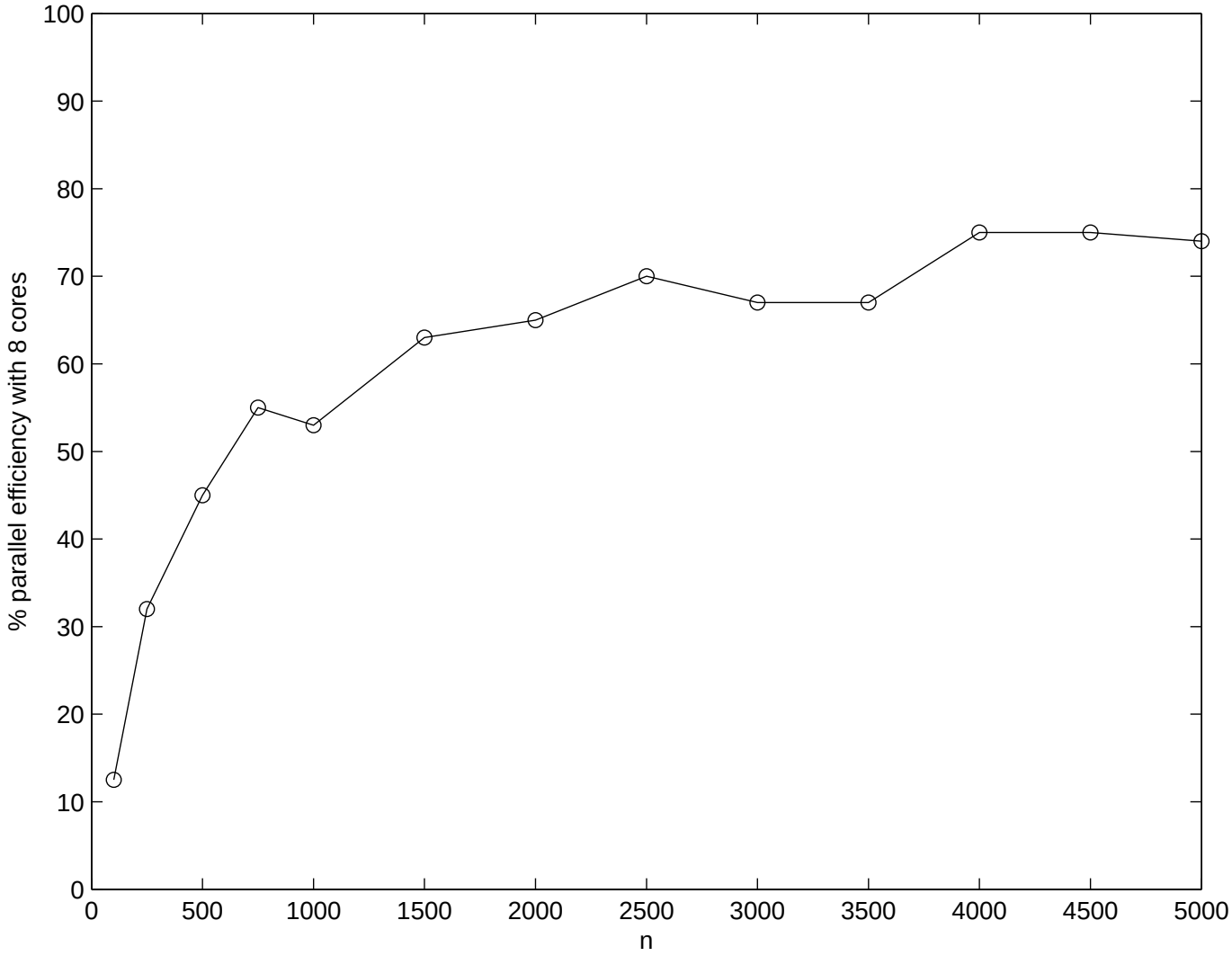
Memory Bandwidth

- The shared memory limits parallel performance in situations where there isn't enough memory bandwidth to keep the processors busy.
- The stream benchmark is a good way to demonstrate this. This benchmark simply copies data from one area of main memory to another as fast as possible.
- Using a single core, we can copy data at a rate of 6,000 megabytes per second (actually 3,000 megabytes per second read and 3,000 megabytes per second written.)
- Using two cores we can copy data at a rate of 11,000 megabytes per second. This increases to 16,000 megabytes per second with four cores and 18,000 megabytes per second using all eight cores.

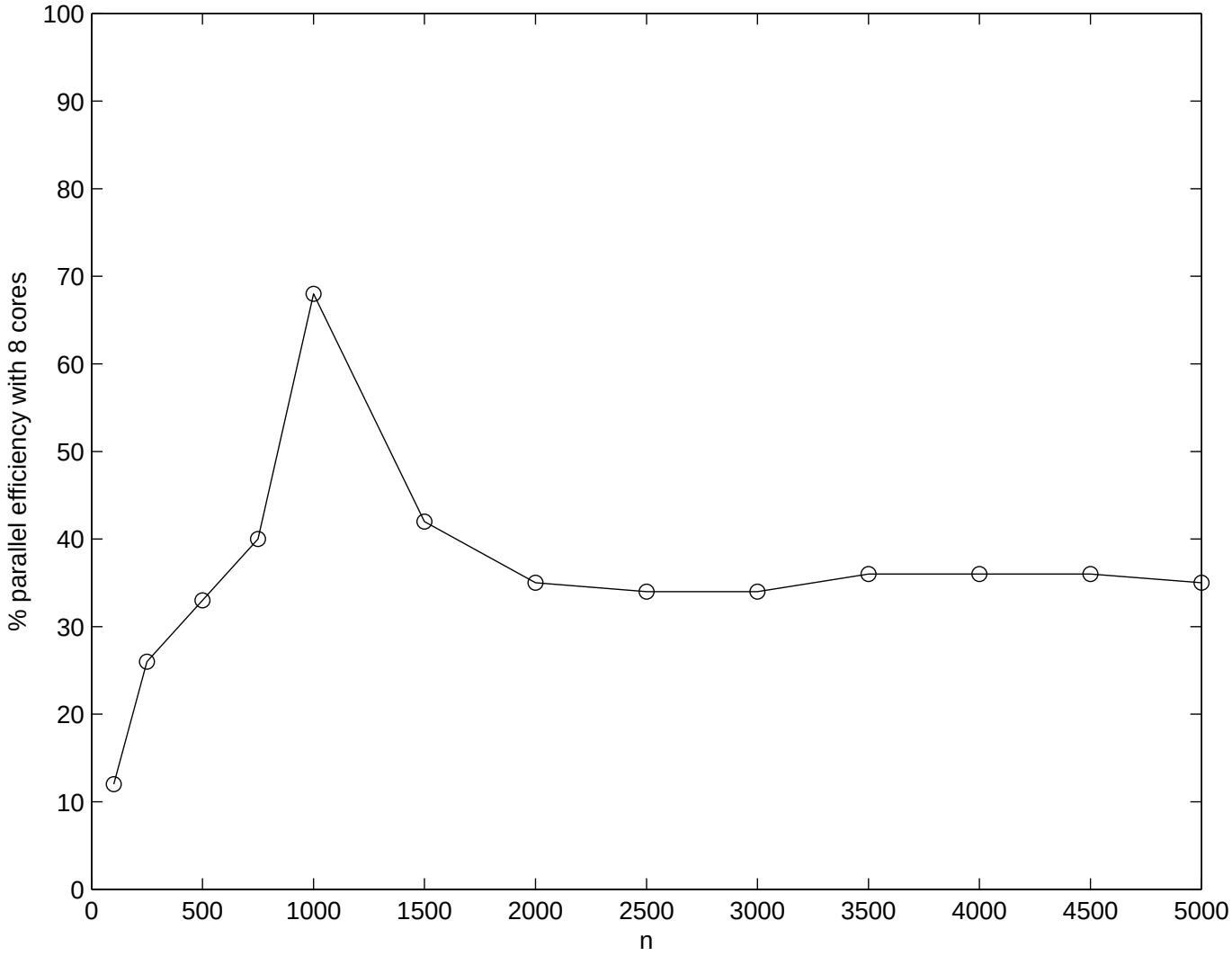
Matrix–Matrix Multiplication (DGEMM)



Cholesky Factorization (DPOTRF)



Matrix–Vector Multiplication (DGEMV)



A Parallel Version Of CSDP

- A 64-bit parallel version of CSDP has been developed to run on a shared memory multiprocessor using OpenMP.
- The code makes use of parallelized BLAS and LAPACK routines such as ATLAS, Intel's MKL, IBM's ESSL, or Sun's performance library.
- Our first attempt used automatic compiler parallelization. However, the performance of code was unsatisfactory.
- Although the Cholesky factorization was reasonably well parallelized, the computation of the elements of the Schur complement matrix was a significant bottleneck.

A Parallel Version Of CSDP

- The routine that computes the elements of the Schur complement matrix was rewritten using OpenMP directives.
- The matrix is split into strips, with each processor working on one strip at a time.
- With this change, the computation of the elements of the Schur complement matrix became much more efficient.
- On single processor systems that don't support OpenMP the same C code is used in a single processor version. The OpenMP directives are simply ignored by compilers that don't support OpenMP.

A Parallel Version Of CSDP

For example, the following bit of code uses an OpenMP directive to parallelize a loop.

```
#pragma omp parallel for shared(O,ldam,k) private(i,j)
   for (j=1; j<=k; j++)
     for (i=1; i<=k; i++)
       O[ijtok(i,j,ldam)]=...
```

control10

The control10 problem has $m = 1326$ and $n_{\max} = 100$.

Run Times	1 Core	2 Core	4 Core	8 Core
Elements	44.7	20.1	12.7	6.5
Cholesky	4.3	2.0	1.2	0.6
Other	7.1	5.3	5.1	5.7
Total	56.2	27.4	18.9	12.9

Efficiency	1 Core	2 Core	4 Core	8 Core
Elements	100%	111%	88%	86%
Cholesky	100%	108%	90%	90%
Other	100%	67%	35%	16%
Total	100%	103%	74%	54%

theta6

The theta6 problem has $m = 4375$ and $n = 300$.

Run Times	1 Core	2 Core	4 Core	8 Core
Elements	8.7	4.5	2.3	1.4
Cholesky	49.9	25.5	13.1	7.7
Other	4.8	4.2	3.8	4.2
Total	63.4	34.2	19.3	13.3

Efficiency	1 Core	2 Core	4 Core	8 Core
Elements	100%	97%	95%	78%
Cholesky	100%	98%	95%	81%
Other	100%	57%	32%	14%
Total	100%	93%	82%	60%

maxG32

The maxG32 problem has $m = 2000$ and $n = 2000$.

Run Times	1 Core	2 Core	4 Core	8 Core
Elements	1.5	0.8	0.4	0.3
Cholesky	4.9	2.5	1.3	0.8
Other	154.5	88.5	56.3	45.1
Total	161.0	91.9	58.1	46.2

Efficiency	1 Core	2 Core	4 Core	8 Core
Elements	100%	94%	94%	63%
Cholesky	100%	98%	94%	77%
Other	100%	87%	69%	43%
Total	100%	88%	69%	44%

maxG32

The majority of the "other" time is spent in BLAS/LAPACK routines operating on the X and Z matrices.

Run Times	1 Core	2 Core	4 Core	8 Core
DGEMM	109.5	55.2	28.0	17.0
DGEMV	7.3	4.5	3.5	3.0
DPOTRF	12.6	6.5	3.4	2.1
DTRTRI	11.4	6.6	4.2	4.5

Efficiency	1 Core	2 Core	4 Core	8 Core
DGEMM	100%	99%	98%	81%
DGEMV	100%	81%	52%	30%
DPOTRF	100%	97%	93%	75%
DTRTRI	100%	86%	68%	32%

Conclusions

- The computation of the elements of the Schur complement matrix is efficiently parallelized.
- The efficiency of the Cholesky factorization of the Schur complement matrix improves as m increases but seems to be limited to about 75%. This could reflect fundamental architectural limitations as well as problems with the efficiency of the MKL BLAS.
- On some problems other operations on the X and Z matrices dominate the total execution time and are not as efficiently parallelized.
- Further improvements in the parallel efficiency of CSDP will depend primarily on increased memory bandwidth and improvements in the efficiency of the BLAS/LAPACK library routines.

Conclusions

- Modern computer architecture is a very complicated subject, but getting good performance out of your code requires some understanding of these issues.
- Limited memory bandwidth can really slow down certain operations and make high parallel efficiency unachievable.
- It's a good idea to make use of library routines that have been optimized for your machine. Let someone else do the hard work!
- Programming with OpenMP is probably the easiest approach to parallel computing for a novice.

Getting CSDP

The current stable version of CSDP is version 6.0.1. You can download the code and a user's guide from

http://projects.coin-or.org/Csdp

The software is available under the Common Public License (CPL).

Hans Mittelmann's benchmarks comparing SDP solvers can be found at

http://plato.la.asu.edu/bench.html