

The PC GAMESS project at MSU: optimization tutorial

Alex A. Granovsky

**Laboratory of Chemical Cybernetics, M.V. Lomonosov
Moscow State University, Moscow, Russia**

November 1999, Intel Oregon

Outline

- The PC GAMESS project.
- Optimization techniques.
- The PC GAMESS performance samples.

The PC GAMESS project

What is Quantum Chemistry?

- Quantum Chemistry (QC) is the science based on applications of the first principles of quantum mechanics to studies of chemical systems.
- All chemical systems are treated as sets of electrons and nuclei. Solutions of the *Schrodinger Equation* contain information on all molecular properties.
- The molecular Schrodinger Equation ought to be solved approximately to get the properties of the molecular system of interest.

What is GAMESS?

- GAMESS means General Atomical and Molecular Electronic Structure System.
- GAMESS (US) is being developed and maintained by the members of the Gordon's research group at Iowa State University.
- Today it is the most popular non-commercial QC package.

How GAMESS is used in chemical research?

- To predict structures of both equilibrium and transition states of molecules in various electronic states.
- To calculate various molecular properties like dipole moments, polarizabilities, atomic charges, and so forth.
- To predict and interpret molecular spectra.
- To calculate sections of molecular Potential Energy Surfaces (PES) and to get various dynamical parameters like lifetimes, reaction rates, and so forth.
- . . .

Why PC?

- Fast.
- Cheap.
- Best price/performance ratio.
- Hundreds of millions PCs over the world.

Why (PC) GAMESS?

- Non-commercial.
- Program sources are available.
- Well-known and trustworthy.
- Broad functionality.
- Variety of available calculation methods.

The PC GAMESS project initial goal:

- To create GAMESS version which will run as fast as possible on Intel-based systems.

What is the PC GAMESS?

The **PC GAMESS** is our freely-available Intel-specific version of the GAMESS (US) program.

By now, approximately 400-600 users (10-15% of all GAMESS users) over the world.

The PC GAMESS key features:

- **Strongly modified** to achieve the maximum possible performance on Intel-based platforms;
- **Functionally extended** to provide QC methods which are not currently present in the regular GAMESS version;
- **Written to support both shared memory** (via **multithreading** on **SMP** systems) and **distributed memory** (via **MPI** on **LANs** and **PC clusters**) **parallel models of execution**;
- **Runs on all popular PC Operating Systems:**
 - ◆ **Win32: NT** (the base OS for PC GAMESS) & Win9x
 - ◆ **Linux** (only partial support at present)
 - ◆ **OS/2**
- **Different executables tuned for Pentium, Pentium Pro, Pentium II and Pentium III CPUs.**

Current goals of the PC GAMESS project:

- Support of modern high-level highly-correlated calculation techniques.
- Better SMP support.
- Better distributed memory parallel algorithms.
- Better performance on new Intel's CPUs.
- Better Linux support.

The PC GAMESS on the Web:

- <http://classic.chem.msu.su/gran/gamess/index.html>

Optimization techniques

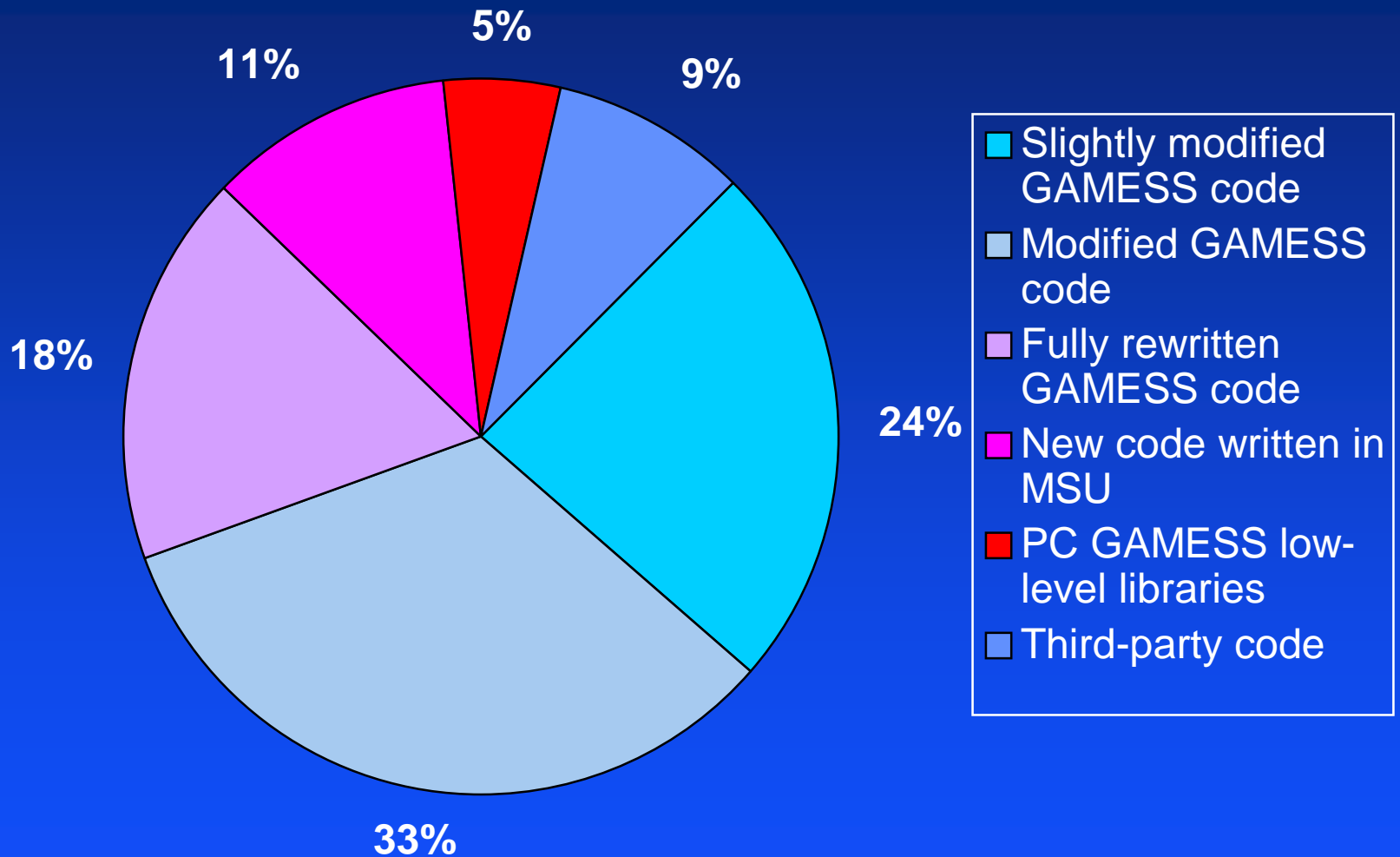
Common problems of all QC packages:

- Non-uniform quality of program sources.
- Variety of algorithms and data structures.
- Both sparse and dense data.
- Huge CPU, memory, and disk space requirements.

What has been done to GAMESS to make it PC GAMESS?

- **Source-level changes.**
- **Intel-specific optimization.**
- **Support of parallel execution (both SMP and distributed memory systems).**
- **Fast I/O and memory management.**
- **Development of our own QC code.**

Structure of the PC GAMESS code:



Source-level changes

Source-level changes:

- Multiple bug fixes.
- Multiple source-level changes to improve performance.
- Multiple changes in the internal data structures.
- Multiple modules have been entirely rewritten to speed up the program.

Source-level changes: key ideas

■ Basic rules:

- ◆ Choice of optimal calculation strategy with minimal number of operations, memory and disk requirements.
- ◆ Memory access optimization by changing data layout.
- ◆ Loop simplification.
- ◆ Loop splitting. Avoiding multiple data streams at a time.
- ◆ Divide removal.
- ◆ Complex code simplification. Data dependence removal.

Source-level changes: key ideas

■ Dense data case:

- ◆ Reformulation of algorithms in terms of linear algebra objects, if possible
- ◆ Extensive use of BLAS routines.
- ◆ BLAS level 3 usage is highly preferred.

Source-level changes: key ideas

■ Sparse data case:

- ◆ Moving from unstructured sparse data to dense data with block structure, if possible.
- ◆ Use of BLAS and sparse BLAS extensions, when appropriate.
- ◆ Use of efficient assembly-written routines.

Source level changes: Example #1

Divide removal:

- $S = \sum a_i/b_i$

- Idea: $a_1/b_1 + a_2/b_2 = (a_1b_2 + b_1a_2)/b_1b_2$

- $a = a(1)$

- $b = b(1)$

 - do $i=2,n$

- $a = a \bullet b(i) + b \bullet a(i)$

- $b = b \bullet b(i)$

 - end do

- $s = a/b$

- One divide → three multiply.

- Potential problem: FP overflow/underflow.

Source level changes: Example #2

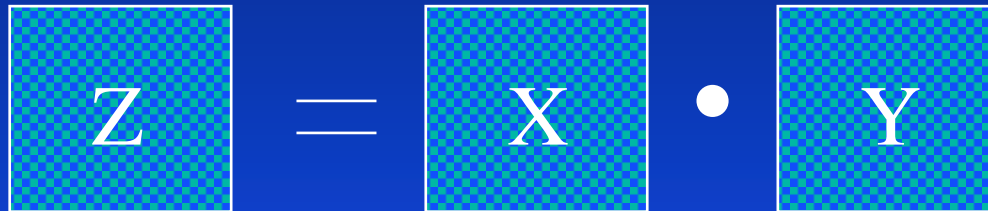
Matrix-matrix multiplication:

- $Y = C_1 \cdot X \cdot C_2 = (C_1 \cdot X) \cdot C_2 = C_1 \cdot (X \cdot C_2)$
- Dimensions: C_1 m by n, X n by n, C_2 n by k, and Y m by k.
- Number of FP operations:
 - ◆ First way: $2 \cdot m \cdot n \cdot n + \underline{2 \cdot m \cdot n \cdot k}$
 - ◆ Second way: $2 \cdot k \cdot n \cdot n + \underline{2 \cdot m \cdot n \cdot k}$
 - ◆ Difference: $2 \cdot (m - k) \cdot n^2$
- Conclusion: the order of multiplications can be very important.

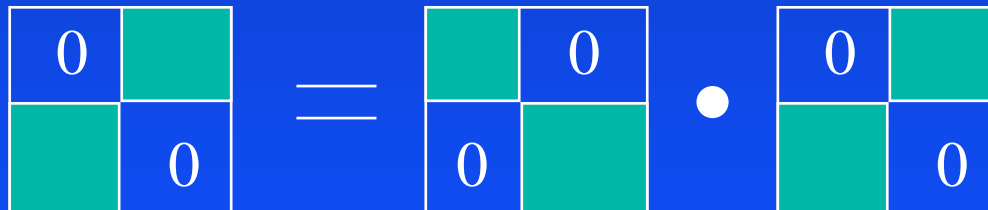
Source level changes: Example #3

Sparse data reordering:

- $Z = X \cdot Y$, matrices X and Y have many zero elements (e.g., due to symmetry).


$$Z = X \cdot Y$$

- After reordering of lines and columns of matrices:


$$Z = X \cdot Y$$

- Only nonzero blocks should be multiplied.

Source level changes: Example #4

Fock matrix update: changing memory layout.

Initial code version:

```
DIMENSION D(*), F(*), IA(*)
DO M=1,NINT
  GET NEXT V AND CORRESPONDING INDICES I,J,K,L
  NIJ = IA(I) + J
  NIK = IA(I) + K
  NIL = IA(I) + L
  NKL = IA(K) + L
  NJK = IA(MAX(J,K)) + MIN(J,K)
  NJL = IA(MAX(J,L)) + MIN(J,L)

  V4 = V*4.0D0

  [ F(NIJ) = F(NIJ) + V4*D(NKL)
    F(NKL) = F(NKL) + V4*D(NIJ)
  [ F(NIK) = F(NIK) - V *D(NJL)
    F(NJL) = F(NJL) - V *D(NIK)
  [ F(NIL) = F(NIL) - V *D(NJK)
    F(NJK) = F(NJK) - V *D(NIL)
END DO
```

Source level changes: Example #4

■ Problem: Why the code above is slow?

- ◆ The consecutive values of indices **NIJ**, **NIK**, **NIL**, **NKL**, **NJK**, and **NJL** usually show no regular patterns.
- ◆ On each iteration, 12 cache lines are fetched, and 6 of them are modified.

■ Solution:

- ◆ Use of different memory layout. Convert arrays **D** and **F** into one structure, aligned on the cache line boundary.
- ◆ In this case, only 6 cache lines are fetched and modified on each iteration.

Source level changes: Example #4

Fock matrix update: changing memory layout.

Code with data locality improved:

```
STRUCTURE /D_F/  
  DOUBLE PRECISION D,F  
END STRUCTURE  
RECORD /D_F/ DF(*)  
  
DO M=1,NINT  
  ...  
  
  [ DF(NIJ).F = DF(NIJ).F + V4*DF(NKL).D  
    DF(NKL).F = DF(NKL).F + V4*DF(NIJ).D  
  
  [ DF(NIK).F = DF(NIK).F - V *DF(NJL).D  
    DF(NJL).F = DF(NJL).F - V *DF(NIK).D  
  
  [ DF(NIL).F = DF(NIL).F - V *DF(NJK).D  
    DF(NJK).F = DF(NJK).F - V *DF(NIL).D  
END DO
```

Source level changes: Example #4

Fock matrix update: data dependence removal.

- Compiler-generated code is still slow because statements #1-6 should be executed in order (some of indices can occasionally coincide):

```
DO M=1,NINT
  ...
  DF(NIJ).F = DF(NIJ).F + V4*DF(NKL).D ! (1)
  DF(NKL).F = DF(NKL).F + V4*DF(NIJ).D ! (2)
  DF(NIK).F = DF(NIK).F - V *DF(NJL).D ! (3)
  DF(NJL).F = DF(NJL).F - V *DF(NIK).D ! (4)
  DF(NIL).F = DF(NIL).F - V *DF(NJK).D ! (5)
  DF(NJK).F = DF(NJK).F - V *DF(NIL).D ! (6)
END DO
```

Source level changes: Example #4

Fock matrix update: data dependence removal.

- In most cases (95-99%), all indices are different.
- Possible solutions:
 - ① Check for coincided indices and handle this case separately, otherwise ignore data dependence.
 - ② Separation of data with coincided indices into special arrays or records. In general case, ignore data dependence. Handle special cases separately.
 - ③ Removal of data dependence by using several temporary data records (next slide).
 - ★ ④ Use of special highly-optimized assembly-written routine.

Source level changes: Example #4

Fock matrix update: data dependence removal.

Code with partially removed data dependence:

```
RECORD /D_F/ DF1(*), DF2(*), DF3(*)
DO M=1,NINT
  ...
  DF1(NIJ).F = DF1(NIJ).F + V4*DF1(NKL).D           ! (1)
  DF1(NKL).F = DF1(NKL).F + V4*DF1(NIJ).D
  DF2(NIK).F = DF2(NIK).F - V *DF2(NJL).D           ! (2)
  DF2(NJL).F = DF2(NJL).F - V *DF2(NIK).D
  DF3(NIL).F = DF3(NIL).F - V *DF3(NJK).D           ! (3)
  DF3(NJK).F = DF3(NJK).F - V *DF3(NIL).D
END DO
```

Source level changes: Example #5

External exchange contributions: loop splitting.

Initial code (simplified model):

```
DO M = 1,NINT
  GET NEXT V12P,V13P,V23P, AND CORRESPONDING INDICES I,J,K,L

  IAI = IA(I)
  IAJ = IA(J)
  IAK = IA(K)

  IJ  = IAI + J
  KL  = IAK + L
  EMP3P = EMP3P + V23P*DDOT(NPAIRS,CijAO(1,IJ),1,CijAO(1,KL),1)

  IL = IAI + L
  JK = IAJ + K
  EMP3P = EMP3P + V12P*DDOT(NPAIRS,CijAO(1,IL),1,CijAO(1,JK),1)

  IK = IAI + K
  JL = IAJ + L
  EMP3P = EMP3P + V13P*DDOT(NPAIRS,CijAO(1,IK),1,CijAO(1,JL),1)
END DO
```

■ Comment: all data reside in L2 cache.

Source level changes: Example #5

External exchange contributions: loop splitting.

New code (up to 50-80% faster):

```
DO M = 1,NINT
  ...
  EMP3P = EMP3P +
          V23P*DDOT(NPAIRS,CijAO(1,IJ),1,CijAO(1,KL),1)
END DO
DO M = 1,NINT
  ...
  EMP3P = EMP3P +
          V12P*DDOT(NPAIRS,CijAO(1,IL),1,CijAO(1,JK),1)
END DO
DO M = 1,NINT
  ...
  EMP3P = EMP3P +
          V13P*DDOT(NPAIRS,CijAO(1,IK),1,CijAO(1,JL),1)
END DO
```

■ Note: all data still in L2 (not L1) cache.

Source level changes: Example #5

External exchange contributions: loop splitting.

■ Why new code is faster?

- ◆ Each loop iteration uses only two data streams!

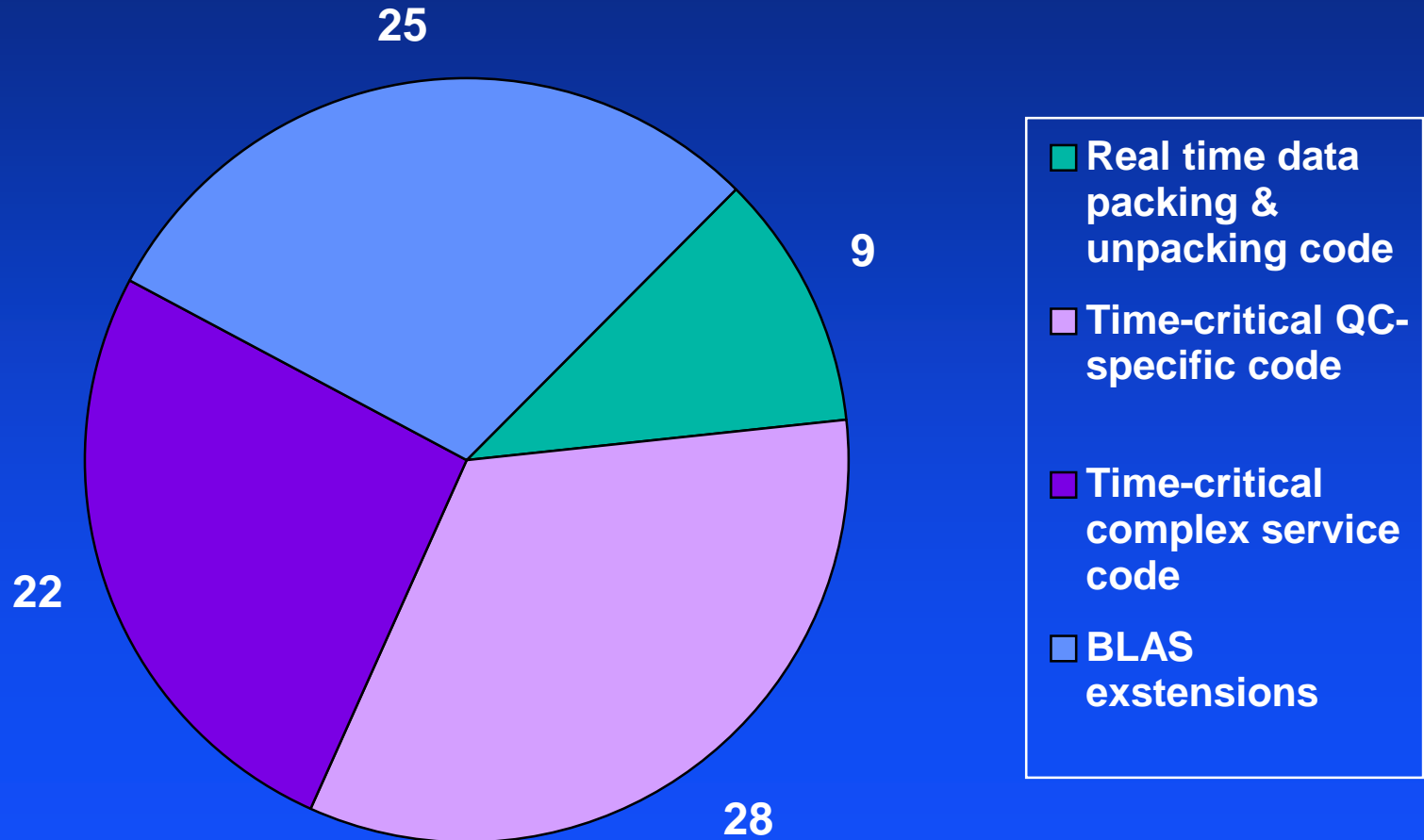
Intel-specific optimization

Intel-specific optimization:

- Intel-specific source-level optimization.
- Creation and use of highly-optimized low-level library of the QC primitives (LQCP).
- Extensive usage of BLAS level 3 (MKL).
- CPU type, L1, and L2 cache size autodetection. This information is used for automatic fine-tuning by several time-critical parts of the PC GAMESS.

Where assembly code (LQCP) was introduced?

Contents of the LQCP library



Why assembly code (LQCP) was introduced?

- Assembly-written code is the fastest.
- Different versions of library are fine-tuned for different Intel's CPUs.
- Assembly-written library reduces the dependence on compiler's quality and reliability.
- Assembly-written code allows one to use new CPU instructions (e.g., cache-manipulation).
- Assembly-written code allows creation of fast OS-independent SMP synchronization primitives.

How to improve performance of assembly code?

- **Typical problem:** not enough integer registers.
- **Idea:** esp can be used as an additional base pointer.
- **How does it work:**
 - ◆ Additional space for the temporary stack should be reserved as the part of the data to be processed.
 - ◆ On entry, routine switches to this temporary stack.
 - ◆ It is now possible to use esp to address data because the offset to the data block is known and it is fixed.
 - ◆ On exit, old stack is restored.

Use of optimized libraries

Use of optimized libraries.

- **Two basic libraries: MKL and LQCP.**
- **Goal:** optimized libraries should be used as extensively as possible.
- **Tools:** code and data structural changes to allow usage of optimized libraries.

Use of optimized libraries: Example.

Original code sequence:

```
DO MB=1,NVIR
  DO MJ=1,NOC
    DO MK=1,NOC
      DO MA=1,NVIR
        MAI = IA(MA+NOC)
        TERM = T(MA,MJ,MK)
        DO MI=1,NOC
          MAI = MAI+1
          DIKAB = E(MI) + E(MK) - E(MA+NOC) - E(MB+NOC)
          P(MI,MJ) = P(MI,MJ) - TERM*X(MK,MAI,MB+NOC)/DIKAB
        END DO
      END DO
    END DO
  END DO
END DO
```

Use of optimized libraries: Example.

First step:

```
DO MB=1,NVIR
  DO MK=1,NOC
    DO MA=1,NVIR
      MAI = IA(MA+NOC)
      DO MI=1,NOC
        MAI = MAI+1
        DIKAB = E(MA+NOC) + E(MB+NOC) - E(MI) - E(MK)
        X(MK,MAI,MB+NOC) = X(MK,MAI,MB+NOC)/DIKAB
        DO MJ=1,NOC
          P(MI,MJ) = P(MI,MJ) + T(MA,MJ,MK)*X(MK,MAI,MB+NOC)
        END DO
      END DO
    END DO
  END DO
END DO
```

Use of optimized libraries: Example.

Second step, loop #1:

```
DO MB=1,NVIR
  DO MA=1,NVIR
    MAI = IA(MA+NOC)
    DO MI=1,NOC
      MAI = MAI+1
      DO MK=1,NOC
        DIKAB = E(MA+NOC) + E(MB+NOC) - E(MI) - E(MK)
        X(MK,MAI,MB+NOC) = X(MK,MAI,MB+NOC)/DIKAB
      END DO
    END DO
  END DO
END DO
```

Use of optimized libraries: Example.

Second step, loop #2:

```
DO MB=1,NVIR
  DO MK=1,NOC
    DO MJ=1,NOC
      DO MA=1,NVIR
        MAI = IA(MA+NOC)
        DO MI=1,NOC
          MAI = MAI+1
          P(MI,MJ) = P(MI,MJ) +
            T(MA,MJ,MK)*X(MK,MAI,MB+NOC)
        END DO
      END DO
    END DO
  END DO
END DO
```

Use of optimized libraries: Example.

Third step, loop #2:

```
DO MB=1,NVIR
  DO MK=1,NOC
    DO MJ=1,NOC
      DO MI=1,NOC
        DO MA=1,NVIR
          MAI = IA(MA+NOC) + MI
          P(MI,MJ) = P(MI,MJ) +
            T(MA,MJ,MK)*X(MK,MAI,MB+NOC)
        END DO
      END DO
    END DO
  END DO
END DO
```

Use of optimized libraries: Example.

Fourth step, loop #2:

```
DO MB=1,NVIR
  DO MK=1,NOC
    DO MI=1,NOC
      DO MA=1,NVIR
        MAI = IA(MA+NOC) + MI
        Y(MA,MI) = X(MK,MAI,MB+NOC)
      END DO
    END DO
  DO MI=1,NOC
    DO MJ=1,NOC
      DO MA=1,NVIR
        P(MI,MJ) = P(MI,MJ) + Y(MA,MI)*T(MA,MJ,MK)
      END DO
    END DO
  END DO
END DO
END DO
```

Use of optimized libraries: Example.

Fifth step, loop #2:

```
DO MB=1,NVIR
  DO MK=1,NOC
    DO MI=1,NOC
      DO MA=1,NVIR
        MAI = IA(MA+NOC) + MI
        Y(MA,MI) = X(MK,MAI,MB+NOC)
      END DO
    END DO
  END DO

  CALL DGEMM('T','N',NOC,NOC,NVIR,1.0D0,Y,NVIR,
            T(1,1,MK),NVIR,1.0D0,P,NOC)
END DO
END DO
```


Use of optimized libraries: Example.

Finally, eliminating loop #1:

```
DO MB=1,NVIR
  DO MK=1,NOC
    DO MI=1,NOC
      DO MA=1,NVIR
        MAI = IA(MA+NOC) + MI
        DIKAB = E(MA+NOC) + E(MB+NOC) - E(MI) - E(MK)
        Y(MA,MI) = X(MK,MAI,MB+NOC) / DIKAB
      END DO
    END DO
    CALL DGEMM('T','N',NOC,NOC,NVIR,1.0D0,Y,NVIR,
              T(1,1,MK),NVIR,1.0D0,P,NOC)
  END DO
END DO
```

Parallelization

Support of parallel execution:

- SMP is supported via multithreading.
- Parallel (MPI-based) PC GAMESS version for Win32-based LANs and clusters.

SMP parallelization

SMP parallelization.

- Multithreading is optimal parallelization strategy on shared memory parallel systems.
- **Benefits:**
 - ◆ More efficient.
 - ◆ Uses less system resources.
 - ◆ No unnecessary code and data duplication.
 - ◆ Simple I/O control and I/O optimization.
- **Drawbacks:**
 - ◆ Multithreaded code is more complex.
 - ◆ Multithreading requires significant changes in data layout. No calculations in COMMONs, only in automatic and dynamic data structures.

SMP parallelization: different ways.

The simplest way:

Use of MKL built in multithreading.

■ Benefits:

- ◆ Takes no additional efforts.
- ◆ Fully transparent.
- ◆ Good scaling if large matrices are used.

■ Drawbacks:

- ◆ Win32-specific solution.
- ◆ Many QC methods do not allow efficient formulation in terms of matrix-matrix multiplications or LAPACK routines.
- ◆ Matrix-formulated QC methods usually deal with relatively small matrices (e.g., from 100x100 to 500x500). Hence, the scaling is usually not very good on four- and eight-CPU systems.

SMP parallelization: different ways.

The best way:

Native support of multithreading.

■ Benefits:

- ◆ Wider applicability.
- ◆ Better performance.
- ◆ Better scaling.

■ Drawbacks:

- ◆ Requires development of new algorithms.
- ◆ Requires data structural changes.
- ◆ Takes additional programming efforts.

SMP parallelization: Real Life.

■ Combination of both MKL-level and native multithreading models.

◆ Use of MKL-level multithreading:

- ◆ Large matrices.
- ◆ Complex matrix-based algorithms which are still to be rewritten to use native multithreading.

◆ Use of native multithreading:

- ◆ QC algorithms which cannot be formulated in terms of matrix-matrix multiplication.
- ◆ Matrix-based QC algorithms which were already rewritten to use native multithreading.
- ◆ Asynchronous I/O.

■ Our priority: purely native multithreading.

SMP parallelization: OpenMP vs. manual multithreading.

■ Use of OpenMP

◆ Benefits:

- ◆ Easy to use.
- ◆ Industry standard.
- ◆ Portability across OpenMP-aware Fortran compilers.

◆ Drawbacks:

- ◆ Requires use of OpenMP-aware Fortran compiler.

SMP parallelization: OpenMP vs. manual multithreading.

■ Use of manual multithreading

◆ Benefits:

- ◆ Potentially better performance.
- ◆ Simpler memory usage control.
- ◆ Flexibility.
- ◆ Wider portability across different OS and Fortran compilers.

◆ Drawbacks:

- ◆ Requires much more programming efforts.

OpenMP vs. manual multithreading: Real Life.

■ Current status:

- ◆ Use of manual multithreading exclusively.

■ Main Reasons:

- ◆ Watcom compilers do not support OpenMP.
- ◆ Simpler memory usage control.

■ Year 2000 plans:

- ◆ Moving to PGI compilers.
- ◆ Test OpenMP-parallelized code versions.
- ◆ Switch to OpenMP if no or little (e.g. <5%) performance degradation.

Manual multithreading and GAMESS legacy code.

■ Key problem:

- ◆ old GAMESS code uses common blocks to pass parameters and to perform calculations (e.g., 2-electron integral code, 2-electron gradient and hessian code).

■ SMP-capable code should:

- ◆ Be reentrant.
- ◆ Receive all parameters as routine arguments.
- ◆ Receive some arguments by value.
- ◆ Perform all calculations using dynamic and automatic data structures only.

■ Solution:

- ◆ Code and data changes to meet these requirements (work in progress).

Manual multithreading and GAMESS legacy code.

■ Comments:

- ◆ Some performance penalty due to use of dynamically allocated data.
- ◆ Code change requires large amount of time.
- ◆ Use of mixed SMP/MPI strategy on SMP systems as a temporary solution.
- ◆ OpenMP usage will probably greatly simplify this transition.

SMP parallelization: Threads synchronization objects.

■ OS-level synchronization objects.

◆ Benefits:

- ◆ No dummy wait loops consuming CPU resources.
- ◆ More CPU resources for other threads, processes, and OS itself.

◆ Drawbacks:

- ◆ Slow due to large system overhead.
- ◆ Different API and functionality on different OSES.

SMP parallelization: Threads synchronization objects.

■ Application-level synchronization objects.

◆ Benefits:

- ◆ Fast.

- ◆ Portable across different OSes.

◆ Drawbacks:

- ◆ Dummy wait loops consume CPU resources.

- ◆ Less CPU resources for other threads, processes, and OS itself.

Threads synchronization objects: Real Life.

■ **Mixed approach.**

- ◆ Use of OS-level synchronization if:
 - ◆ Long delays.
 - ◆ Serious impact on program or system performance.
- ◆ Use of application-level synchronization if:
 - ◆ Short delays.
 - ◆ No or little impact on program or system performance.

Distributed memory parallelization

Distributed memory parallelization: Current status.

- Mainly inherited from the original GAMESS code.
- MPI-based.
- Static load balancing.
- Supported by Win32-based PC GAMESS versions (using WMPI v. 1.2).
- Compatible with most of the new code which is PC GAMESS specific.
- Compatible with SMP parallelization.

MPI and SMP parallelization: Basic concepts for new code development.

- Thread-safe programming style.
- MPI parallelization over outermost loops, SMP parallelization over inter- and innermost loops.
- Reduce communications costs as much as possible, duplicate data if necessary.
- If SMP parallelization of some computational stage is impossible or multithreaded code is still to be developed, use MPI-based code to perform this step on SMP system. Then, switch back to SMP mode, and so forth.

Parallelization sample: MP4(T) energy calculation.

Skeleton of the simplified MP4(T) energy code

```
DO I=1,NOC
DO J=I,NOC
DO K=J,NOC
  GET NECESSARY DATA
  DO MC=1,NVIR
    CALL DGEMM()
    CALL DGEMM()
    REORDER RESULTS
  END DO
  DO MC=1,NVIR
    CALL DGEMM()
    CALL DGEMM()
    REORDER RESULTS
  END DO
  DO MC=1,NVIR
    CALL DGEMM()
    CALL DGEMM()
  END DO
  EVALUATE CONTRIBUTION TO MP4(T) ENERGY
END DO
END DO
END DO
```

These loops are distributed
over different nodes

These calculations
are distributed over
CPUs on one node.
Each node has all
necessary data.

I/O optimization

Fast I/O and memory management:

- Fast non-Fortran file I/O with large files and asynchronous I/O support.
- Real time data packing/unpacking technology.
- Advanced memory management technology.

How QC programs use I/O?

- Both sequential and random I/O.
- Both fixed and variable-size records.
- Both small and large records.
- Typical strategy: write once, read multiple.
- I/O operations are usually intermixed with data processing.
- Large file sizes.

I/O optimization.

Fortran I/O vs non-Fortran I/O.

■ Fortran I/O:

- ◆ Slow (multi-buffered).
- ◆ Limits maximum file size (2 or 4 GB).
- ◆ Synchronous.

■ non-Fortran I/O:

- ◆ Fast (uses OS-level API directly).
- ◆ Uses OS advanced I/O features.
- ◆ Supports large files.
- ◆ Allows transparent use of asynchronous I/O.
- ◆ Flexible.

How non-Fortran I/O is implemented?

- In GAMESS, all unformatted I/O operations are always performed as calls of the dedicated I/O routines (Fortran written).
- These routines check for non-Fortran I/O usage. If enabled, they call high-level functions from the non-Fortran I/O module (C written).
- High-level non-Fortran I/O functions calls low-level I/O functions.
- Low-level I/O functions call Operating System I/O API functions (Win32 and OS/2 are currently supported).

I/O optimization.

Why asynchronous I/O is important?

- Increases overall I/O throughput.
- Hides I/O latencies.
- Improves performance allowing simultaneous data processing and I/O.

I/O optimization.

Where asynchronous I/O is important?

■ Sequential I/O:

- ◆ Write operations are asynchronous on OS level.
- ◆ Read operations are used more frequently.
- ◆ Intensive reads are usually synchronous.
- ◆ Conclusion: asynchronous reads are important.

■ Random I/O:

- ◆ Random writes are often a big problem for OS.
- ◆ Huge latencies due to disk mechanics.
- ◆ Conclusion: both asynchronous reads and writes are important.

Asynchronous I/O implementation.

■ Use of OS-level API:

- ◆ Slow.
- ◆ Unportable.
- ◆ Difficult to implement transparently.

★ Use of dedicated I/O server threads:

- ◆ Faster.
- ◆ Portable.
- ◆ Transparent.

Asynchronous I/O implementation.

■ Sequential (fully predictable) I/O:

- ◆ Allows fully transparent implementation.

■ Random (unpredictable) I/O:

- ◆ Fully transparent implementation is impossible.
- ◆ Each I/O request is handled separately.
- ◆ Explicit synchronization is usually required.
- ◆ More difficult to program and use.

I/O optimization. Additional hints.

- Use of higher priority for asynchronous I/O server threads.
- Use of OS-specific I/O optimization hints (like *FILE_FLAG_SEQUENTIAL_SCAN*).
- Record size alignment on cluster or disk sector boundary.
- File truncation:
 - ◆ Sequential access files: truncate at zero length before reusing for writing.
 - ◆ Random access files: never truncate before reusing for writing.
- Renewal of OS-level file handles.

New QC code

Development of our own QC codes which are the PC GAMESS specific:

- Fast MP2 energy/energy gradient modules.
- Fast MP3/MP4 modules with SMP and parallel mode support.
- New modules for high-level calculations based on coupled cluster approach (work in progress).

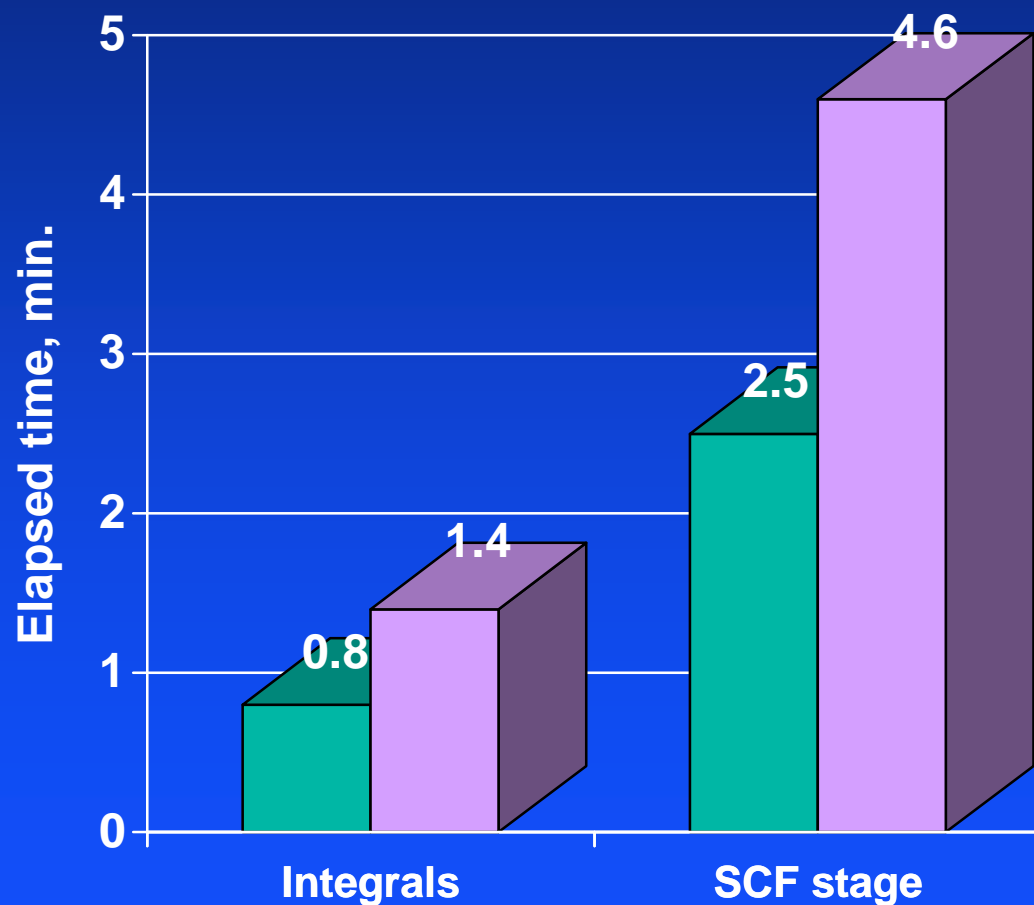
The PC GAMESS performance samples

The PC GAMESS performance.

Model chemical system:

- ◆ 38 atoms (C, N, O, H, S, Zn)
 - ◆ 214 electrons
-
- ◆ SCF calculation
 - ◆ Number of basis functions (N) **216**
 - ◆ Number of atomic integrals **150 millions**
 - ◆ Number of SCF iterations **19**

SCF calculation running on four CPUs.



■ PC GAMESS on two dual-CPU Pentium III Xeon (500MHz, 1MB L2 cache)-based workstations, 512 MB RAM each

■ GAMESS on Origin 2000 SGI. 64 195&250 MHz MIPS R10000 processors, 17 GB main memory

The PC GAMESS performance.

Model chemical system:

- ◆ 11 atoms (H, F, Cl)
 - ◆ 68 electrons
-

- ◆ MP4(full) calculation
- ◆ Number of basis functions (N) **227**
- ◆ Number of FP operations $\sim 43 \cdot 10^{12}$

MP4(full) calculation running on cluster of four P3XP (500 MHz, 1 MB L2 cache, 512 MB RAM).
PC GAMESS runs in SMP mode on each box.

MP4(full) parallel scalability testcase

$N_{\text{core}} = 10$, $N_{\text{occ}} = 34$, $N_{\text{virt}} = 193$, $N = 227$, C1 symmetry group

