Writing Efficient Programs in Fortran, C and C++: Selected Case Studies

Georg Hager
Frank Deserno
Dr. Frank BrechTFeld
Dr. Gerhard Wellein

Regionales Rechenzentrum Erlangen
HPC Services

Agenda

· “Common sense“ optimizations
  · Case Study: Optimization of a Monte Carlo spin system simulation

· Classic data access optimizations
  · Case Study: Optimization of kernel loops
  · Case Study: Optimization and parallelization of a Strongly Implicit Solver

· Advanced Parallelization
  · Case Study: Parallelization of a C++ sparse matrix-vector multiplication
Case Study: Optimization of a Monte Carlo Spin System Calculation

Optimization of a Spin System Simulation: Model

- 3-D cubic lattice
- One variable ("spin") per grid point with values +1 or -1
- "Interaction": Variables on neighbouring grid points prefer to have the same values
Optimization of a Spin System Simulation:

- **Systems under consideration**
  - $50 \times 50 \times 50 = 125000$ lattice sites
  - $2^{125000}$ different configurations
  - Computer time: $2^{125000} \cdot 1 \text{ ns} \approx 10^{37000} \text{ years}$

- **Loophole: Monte Carlo simulation!**
  - Random choice of a subset of all configurations

- **Memory requirement of original program $\approx 1 \text{ MByte}$

Optimization of a Spin System Simulation:

**Original Code**

- **Program Kernel:**
  
  \[
  \begin{align*}
  IA &= IZ(KL,KM,KN) \\
  IL &= IZ(KLL,KM,KN) \\
  IR &= IZ(KLR,KM,KN) \\
  IO &= IZ(KL,KMO,KN) \\
  IU &= IZ(KL,KMU,KN) \\
  IS &= IZ(KL,KM,KNS) \\
  IN &= IZ(KL,KM,KNN)
  \end{align*}
  \]

  \[
  edelz = iL + iR + iU + iO + iS + iN
  \]

  \[
  BF = 0.5d0*(1.d0 + \tanh(edelz/tt))
  \]

  \[
  \text{IF}(YHE \LE BF) \text{ then} \\
  iZ(kl,km,kn) = 1 \\
  \text{else} \\
  iZ(kl,km,kn) = -1
  \]

  decide about spin orientation

- **C CRITERION FOR FLIPPING THE SPIN**

  - Load neighbours of a random spin
  - Calculate magnetic field
Optimization of a Spin System Simulation:
Code Analysis

- Profiling shows that
  - 30% of computing time is spent in the $\tanh$ function
  - Rest is spent in the line calculating $\text{edelz}$
- Why?
  - $\tanh$ is expensive by itself
  - Compiler fuses the spin loads and calculation of $\text{edelz}$ into a single line

- What can we do?
  - Try to reduce the „strength“ of calculations (here $\tanh$)
  - Try to make the CPU move less data
- How do we do it?
  - Observation: argument of $\tanh$ is always integer in the range -6..6 ($tt$ is always 1)
  - Observation: Spin variables only hold values +1 or -1

Optimization of a Spin System Simulation:
Making it Faster

- Strength reduction by tabulation of $\tanh$ function
  
  $$BF = 0.5d0*(1.d0+tanh\_table(edelz))$$

  - Performance increases by 30% as table lookup is done with „lightspeed“ compared to $\tanh$ calculation
  - By declaring spin variables with INTEGER*1 instead of INTEGER*4 the memory requirement is reduced to about ¼
    - Better cache reuse
    - Factor 2–4 in performance depending on platform
    - Why don’t we use just one bit per spin?
      - Bit operations (mask, shift, add) too expensive $\rightarrow$ no benefit
  - Potential for a variety of data access optimizations
    - But: choice of spin must be absolutely random!
Optimization of a Spin System Simulation: Performance Results

- Pentium 4 (2.4 GHz)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Runtime [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table + 1Bit/Spin</td>
<td>150</td>
</tr>
<tr>
<td>Table + 1Byte/Spin</td>
<td>200</td>
</tr>
<tr>
<td>Original code</td>
<td>220</td>
</tr>
</tbody>
</table>

Case Study: Optimization of Kernel Loops
Optimization of Kernel Loops

- Code from theoretical nuclear physics (three-nucleon interaction)
  - MPI code, typically 64 CPUs (8 nodes) on SR8000
- Original program performance on SR8000 (1 CPU): 26 MFlops
- Major part (98%) of compute time is attributed to code fragment with two simple loops:

  ```
  do M = 1, IQM
     do K = KZHX(M), KZAHL
       F(K) = F(K) * S(MVK(K,M))
     enddo
  enddo
  do K = 1, KZAHL
     WERTT(KVK(K)) = WERTT(KVK(K)) + F(K)
  enddo
  ```

  1st loop: \( \approx \frac{3}{4} \) of time
  2nd loop: \( \approx \frac{1}{4} \) of time

Optimization of Kernel Loops: Helping the Compiler

- SR8000 compiler with highest optimization chooses the following pseudo-vectorization strategy:
  - Prefetch for \( \text{MVK()} \), \( \text{F()} \) and \( \text{KVK()} \)
  - Preload for \( \text{S()} \) and \( \text{WERTT()} \)
  - Outer loop unrolling of first loop impossible due to dependencies
  - Unrolling of second loop useless due to possible dependencies

- Important facts about the data structures:
  - \( \text{IQM} \) is small (typically 9)
  - Entries in \( \text{KZHX()} \) are sorted in ascending order
  - Length of \( \text{S()} \) is small (between 100 and 200), \text{array fits in cache}
  - \( \text{KZAHL} \) is typically a couple of 1000s
  - Length of \( \text{WERTT()} \) is very small (1 in the worst case), \text{fits in cache}

- First aid: disable pseudo-vectorization for \( \text{S()} \) and \( \text{WERTT()} \)
  - \( \rightarrow \) acceleration to 77 MFlops!
Optimization of Kernel Loops:
Why preload is not always beneficial

- Preload must be issued...
  - for every input stream in the loop that is eligible for it
  - for every iteration of the unrolled loop
- Significant overhead for data that is already in cache
- Why is prefetch not as bad for in-cache data?
  - Prefetch only necessary for each 16th data element in each stream (cache line size is 128 bytes)
  - This rate is achieved by the appropriate amount of unrolling
    - unrolling avoids unnecessary prefetches
- Preload might be better for strided access
  - The larger the stride, the less efficient is prefetch

---

Example: Vector product

\[ A(1:N) = B(1:N) \times C(1:N) \]

- In-cache preload penalty: factor 3
  - No cache reuse!
  - One preload per iteration
- In-cache prefetch penalty: maybe 10%
  - Just one prefetch every 16 iterations
- Out-of-cache preload: better than nothing, but much worse than (consecutive) prefetch
Optimization of Kernel Loops: Why preload is not always beneficial

- Strided access (stride 8): Bad reuse of prefetched data
  - Effective cache size is only 1/8 of real size
  - One prefetch every other iteration
    - CPU running out of memory references!
  - Stride does not affect performance of preload streams

Optimization of Kernel Loops: Data Access Transformations, First Loop

- Is there more potential for optimization?
  - Try to enable unrolling of outer loop!
  - Original access pattern:

```plaintext
F(KZAH) is loaded many times from memory (or cache)
```

- Initially: no outer loop unrolling possible, i.e. no potential for register reuse
Optimization of Kernel Loops: Data Access Transformations, First Loop

- Visualization of data access

\[ F'(K) \]
\[ F'(K) \]
\[ F(K) \]

\[ F(K) = F(K) \times S(MVK(K, M)) \]

\[ S(MVK(K, M)) \]
\[ MVK(K, M) \]

- Naive optimization: "pseudo-loop-interchange"
  - New access pattern: introduce new outer loop level (blocking), interchange middle and inner loops

- Now full register reuse for \( F() \) possible
- \( F() \) is loaded only once from memory
- Downside: small inner loop length
Optimization of Kernel Loops:
Data Access Transformations, First Loop

- Naïve optimization does not pay off with SR8000 and all Intel systems
  - Inner loop length too small
  - Even manual unrolling of middle ($k$) loop does not help
- Remedy: Retain a moderately large inner loop length but enable unrolling to improve Flop/Load quotient
  - Access pattern:

  ![Access Pattern Diagram]

  - Unrolling of middle loop now possible

Optimization of Kernel Loops:
Data Access Transformations, First Loop

- Final access pattern:

  ![Final Access Pattern Diagram]

  - special treatment of odd middle loop lengths necessary
  - SR8000 compiler now unrolls the middle loop further
    - overall unrolling factor of 48
    - moderate integer register spill
  - Performance: $\approx 87$ MFlops
Optimization of Kernel Loops:
Optimal SR8000 Code for First Loop

```fortran
do M=1,IQM
    ISTART=KZHX(M)
    if(M.NE.IQM) then
        IEND=KZHX(M+1)-1
    else
        IEND=KZAHL
    endif
    IS=1
    if(mod(M,2).NE.0) then
        do MM=1,mod(M,2)
            *voption nopreload(S)
            *voption noprefetch(S)
            do K=ISTART,IEND
                F(K)=F(K)*S(MVK(K,IS))
            enddo
            IS=IS+mod(M,2)
        enddo
    endif
    do MM=IS,M,2
        *voption nopreload(S)
        *voption noprefetch(S)
        do K=ISTART,IEND
            F(K)=F(K)*S(MVK(K,MM))
        enddo
    enddo
enddo
```

middle loop

remainder loop

Optimization of Kernel Loops:
Data Access Transformations, Second Loop

- **Problem:** Data dependency prevents compiler from unrolling the loop (no improvement expected)
- **Remedy:** Unrolling pays off when the instances of the loop body write to different targets

```fortran
do K=IS,KZAHL,2
    WERTT(KVK(K)) = WERTT(KVK(K)) + F(K)
    if(IM.lt.KVK(K)) IM=KVK(K)
    WERTT2(KVK(K+1)) = WERTT2(KVK(K+1)) + F(K+1)
    if(IN.lt.KVK(K+1)) IN=KVK(K+1)
enddo
IQ=max(IM,IN)
do K=1,IQ
    WERTT(K)=WERTT(K)+WERTT2(K)
enddo
```

calculation of length

for reduction loop

remainder loop omitted!

Final subroutine performance: \( \approx 94 \) MFlops
- Whole program: 90 MFlops; MPI code performance doubled
Optimization of Kernel Loops:
Architectural Issues

- **MIPS R14000**: Optimal strategy is the naïve optimization!
  - Original code performs about as well as fully optimized version on SR8000
  - no unnecessary preload attempts because there is no provision for preload
  - Good performance of short loops due to short pipelines
  - Compiler unrolls the middle loop automatically to make the loop body fatter
  - 2 instructions/cycle (very good!)
  - Final code on O3400 is about 50% faster than optimal version on SR8000

- **IA32**: Optimal strategy is the same as for SR8000
  - Very limited FP register set, stack-oriented
  - Few integer registers
  - Long P4 pipeline, but good performance with short loops
  - due to special L1-ICache (decoded instructions)?

- **IA64**: Optimal strategy is the same as for SR8000
  - Very bad performance for naïve strategy
  - Further unrolling (by 4) of middle loop helps
  - But: Naïve optimization with middle loop unrolling (16-fold) is also very close to optimum
  - Also some benefit on IA32, but not that much
Optimization of Kernel Loops: Conclusions

- SR8000 is a RISC architecture, but has some particular features
  - Vectorization abilities
    - 16 outstanding prefetches
    - 128 outstanding preloads
  - Large bandwidth
  - Long FP pipelines
- Careful data stream analysis is more important on SR8000 than on other RISC systems
  - Sometimes PVP gets in the way of performance
- MIPS behaviour is as expected for typical RISC machine
- IA32/IA64 is still a mystery
  - Complicated architecture (CISC+RISC/EPIC), maybe compiler deficiencies

Case Study: Optimization and Parallelization of a Strongly Implicit Solver
CFD kernel: Strongly Implicit Solver

- CFD: Solving $Ax = b$
  - for finite volume methods can be done by Strongly Implicit Procedure (SIP) according to Stone

- SIP-solver is widely used:
  - LESOCC, FASTEST, FLOWSI (Institute of Fluid Mechanics, Erlangen)
  - STHAMAS3D (Crystal Growth Laboratory, Erlangen)
  - CADiP (Theoretical Thermodynamics and Transport Processes, Bayreuth)
  - ...

- SIP-Solver:
  1) Incomplete LU-factorization
  2) Series of forward/backward substitutions

- Toy program available at:
  ftp.springer.de:/pub/technik/peric (M. Peric)

SIP-solver: Data Dependencies & Implementations

Basic data dependency:

(i,j,k) ← {(i-1,j,k);(i,j-1,k);(i,j,k-1)}

Dominant part: Forward (and backward) Substitution!

Naive 3D version:

```plaintext
do k = 2 , kMax
  do j = 2 , jMax
    do i = 2 , iMax
      \[ RES(i,j,k) = (RES(i,j,k) - LB(i,j,k) \times RES(i,j,k-1) - \]
      \[ LW(i,j,k) \times RES(i-1,j,k) - LS(i,j,k) \times RES(i,j-1,k) ) \times \]
      \[ LP(i,j,k) \]
    enddo
  enddo
endo
```
SIP-solver: Data Dependencies & Implementations

size = 91^3 (100 MB); naive implementation/compiler switches

- IBM Power4 (p690)
- IBM Power4 (p655)
- I1 (SGI 750)
- I2 (900MHz; efc7.1)
- I2 (1 GHz; efc7.1)
- Opteron 1,4 GHz (ifc)
- Opteron 1,4 GHz (pgf90)
- Xeon 2,6 GHz (ifc)
SIP-solver:
Implementations & Single Processor Performance

- size = 91^3 (100 MB)
- IBM improvements:
  - split single loop in 4 separate loops; use large pages

\[ \text{Basic data dependency:} \quad (i,j,k) \leftarrow \{(i-1,j,k),(i,j-1,k),(i,j,k-1)\} \]

Define Hyperplane: \( i+j+k = \text{const} \)
- non-contiguous memory access
- shared memory parallelization / vectorization of innermost loop

\[
\begin{align*}
\text{do } l=1, \text{hyperplanes} \\
\quad n=\text{ICL}(l) \\
\quad \text{do } m=n+1,n+\text{LM}(l) \\
\quad \quad ijk=\text{IJKV}(m) \\
\quad \quad \text{RES}(ijk)=(\text{RES}(ijk)-\$ LB(ijk)*\text{RES}(ijk-ijMax)-\$ LW(ijk)*\text{RES}(ijk-1)-\$ LS(ijk)*\text{RES}(ijk-iMax)) \\
\quad \quad \quad * LP(ijk) \\
\quad \quad \text{enddo} \\
\quad \text{enddo}
\end{align*}
\]
**SIP Solver: SMP scalability (Hyperplane)**

![Graph showing MPlop/s vs. # of Threads for different systems]

- NEC SX6 (hp)
- NEC SX5e (hp)
- I2 (1 GHz; efc7.1) (hp)
- I2 (1 GHz; efc7.1) (3D)

* !DIR$IVDEP

**SIP-solver: Data Dependencies & Implementations**

**Basic data dependency:**

\[(i, j, k) \leftrightarrow \{(i-1, j, k); (i, j-1, k); (i, j, k-1)\}\]

3-fold nested loop (3D): \((i, j, k)\)

- Data locality, but recurrences
- No automatic shared memory parallelization by compiler, but OpenMP (except Hitachi SR8000: Pipeline parallel processing)

**j-loop is being distributed:**

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Barrier
**SIP-solver: SMP scalability (Pipeline Parallel Processing)**

- **NEC SX6 (hp)**
- **NEC SX5e (hp)**
- **I2 (1 GHz; efc7.1) (ppp) 91**
- **I2 (1 GHz; efc7.1) (ppp) 201**
- **SGI O3400 (500 MHz)**

**Benchmark:**
- **Lattice**: $91^3$
- **100 MB**
- **1 ILU**
- **500 iterations**

**HSR8k-F1:**
- unrolling up to 32 times

**IBM Power4 (p690):**
- **128 MB L3 cache accessible for 1 CPU**
IBM wants to improve performance

SIP-solver: Implementations & Single Processor Performance

SIP-solver: SMP scalability (Hyperplane & Hyperline)
SMP scalability:
Pipeline Parallel Processing (3D)

Be careful on vector systems when choosing memory layout!

SIP-solver: Problem Sizes & Performance

Varying problem size

Be careful with cache effects on IBM p690!
Case Study: Parallelization of a Sparse MVM in C++

Sparse MVM Procedure in DMRG

- DMRG
  - Density-Matrix Renormalization Group Algorithm
  - Used for solving quantum problems in solid state physics and theoretical chemistry
  - Alternative to expensive (resource-consuming) Exact Diagonalization
- Core of DMRG: Sparse matrix-vector multiplication (in Davidson diagonalization)
  - Dense matrices as matrix and vector components
  - Dominant operation at lowest level: dense matrix-matrix multiply (use optimized Level 3 BLAS!)
- Parallelization approaches:
  - Use parallel BLAS (no code changes)
  - Parallelize sparse MVM using OpenMP
DMRG: Potential Parallelization approaches

Implementation of sparse MVM - pseudocode

\[ H \psi = \sum_{\alpha} \sum_{k} A^\alpha_k \psi_R(k) \left[ B^T_k \right]^\alpha \]

// W: wavevector ; R: result
for (α=0; α < number_of_hamiltonian_terms; α++) {
    term = hamiltonian_terms[α];
    for (k=0 ; k < term.number_of_blocks; k++) {
        li = term[k].left_index;
        ri = term[k].right_index;
        temp_matrix = term[k].B.transpose() * W[ri];
        R[li] += term[k].A * temp_matrix;
    }
}

Matrix-Matrix-Multiply (Parallel DGEMM ?!)

Data dependency!

1. Linking with parallel BLAS (DGEMM)
   - Does not require restructuring of code
   - Significant speed-up only for large (transformation) matrices (A, B)

2. Shared-Memory parallelization of outer loops
   - Chose OpenMP for portability reasons
   - Requires some restructuring & directives
   - Speed-Up should not depend on size of (transformation) matrices

Expected maximum speed-up for total program:
   - if MVM is parallelized only: \( \sim 6 - 8 \)
   - if also Davidson algorithm is parallelized: \( \sim 10 \)

MPI parallelization
   - Requires complete restructuring of algorithm
DMRG: Linking With Parallel BLAS

- Useless on IBM for #CPU > 4
- Best scalability on SGI (Network, BLAS implementation)
- Dual processor nodes can reduce elapsed runtime by about 30 %
- Speedup is also strongly dependent on problem parameters

DMRG: OpenMP Parallelization

- Parallelization of innermost k loop: Scales badly
  - loop too short
  - collective thread operations within outer loop
- Parallelization of outer α loop: Scales badly
  - even shorter
  - load imbalance (trip count of k loop depends on α)

Solution:
- “Fuse” both loops (α & k)
- Protect write operation R[li] with lock mechanism
- Use list of OpenMP locks for each block li
DMRG: OpenMP Parallelization

Implementation of parallel sparse MVM – pseudocode (prologue loops)

// store all block references in block_array
ics=0;
for (α=0; α < number_of_hamiltonian_terms; α++) {
    term = hamiltonian_terms[α];
    for (k=0; k < term.number_of_blocks; k++) {
        block_array[ics]=&term[q];
        ics++;
    }
}
icsmax=ics;

// set up lock lists
for(i=0; i < MAX_NUMBER_OF_THREADS; i++)
    mm[i] = new Matrix // temp.matrix
for (i=0; i < MAX_NUMBER_OF_LOCKS; i++) {
    locks[i]= new omp_lock_t;
    omp_init_lock(locks[i]);
}

DMRG: OpenMP Parallelization

Implementation of parallel sparse MVM – pseudocode (main loop)

// W: wavevector ; R: result
#pragma omp parallel private(mymat, li, ri, myid, ics)
{
    myid = omp_get_thread_num();
    mytmat = mm[myid]; // temp thread local matrix

#pragma omp for
    for (ics=0; ics< icsmax; ics++) {
        li = block_array[ics]->left_index;
        ri = block_array[ics]->right_index;
        mytmat = block_array[ics]->B.transpose() * W[ri];
        omp_set_lock(locks[li]);
        R[li] += block_array[ics]->A * mytmat;
        omp_unset_lock(locks[li]);
    }
}

Fused (α,k) loop

Protect each block of result vector R with locks
DMRG: OpenMP Parallelization

- The parallel code is compliant to the OpenMP standard
- However: NO system did compile and produce correct results with the initial MVM implementation!

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Problem Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM xIC V6.0</td>
<td>OpenMP locks prevent <code>omp for</code> parallelization</td>
<td>Fixed by IBM</td>
</tr>
<tr>
<td>Intel ifc V7</td>
<td>Severe problems with orphaned <code>omp critical</code> directives in class constructors</td>
<td>Does not work</td>
</tr>
<tr>
<td>SUN forte7</td>
<td>Does not allow <code>omp critical</code> inside C++ classes!</td>
<td>Does not work (Forte 8 EA does work)</td>
</tr>
<tr>
<td>SGI MIPSpro 7.3.1.3m</td>
<td>Complex data structures can not be allocated inside <code>omp parallel regions</code></td>
<td>Allocate everything outside loop</td>
</tr>
</tbody>
</table>

DMRG: OpenMP Parallelization

Scalability on SGI Origin

- `OMP_SCHEDULE=STATIC`
- OpenMP scales significantly better than parallel DGEMM
- Serial overhead in parallel MVM is only about 5%
- Still some parallelization potential left in program
DMRG: OpenMP Parallelization

Scalability & Performance: SGI Origin vs. IBM p690

- Scalability is pretty much the same on both systems
- Single processor run and OMP_NUM_THREADS=1 differ by approx. 5% on IBM
  - Hardly any difference on SGI
- Total performance
  1 * Power4 = 8 * MIPS
  8 * Power4 > 12 GFlop/s sustained!

Further improvement of total performance/scalability

- Chose best distribution strategy for parallel for loop:
  OMP_SCHEDULE="dynamic,2"
  (reduces serial overhead in MVM to 2%)
- Re-Link with parallel LAPACK /BLAS to speed-up density-matrix diagonalization (DSYEV).
  Good thing to do:
  OMP_NESTED=FALSE
Thank You!