High Performance Computing
Selected topics in shared-memory parallelization

G. Hager, G. Wellein
Regionales Rechenzentrum Erlangen

W. u. E. Heraeus Summerschool
on Computational Many Particle Physics
Sep 18-29, Greifswald, Germany
Another warning

Optimization of sequential code goes first!
Outline

- Architecture of shared memory computers
  - UMA/ccNUMA
  - Cache coherence

- Shared memory programming
  - Introduction to OpenMP
  - Common pitfalls
  - Parallelization of sparse MVM

- Programming for ccNUMA systems
  - Correct page placement
  - Optimization of parallel sparse MVM
  - C++ issues
Architecture of shared memory computers
Shared memory computers: Basic concepts

- Shared Memory Computer provides single, shared address space for all parallel processors

- Two basic categories of shared memory systems
  - Uniform Memory Access (UMA):
    - Flat Memory: Memory is equally accessible to all processors with the same performance (Bandwidth & Latency).
    - A.k.a Symmetric Multi Processor (SMP) system
  
  - Cache-Coherent Non Uniform Memory Access (ccNUMA):
    - Memory is physically distributed: Performance (Bandwidth & Latency) is different for local and remote memory access.

- Cache-Coherence protocols and/or hardware provide consistency between data in caches (multiple copies of same data!) and data in memory
Shared memory computers: UMA

UMA architecture

Simplest implementation: Dual-Core Processor (e.g. AMD Opteron dual-core or Intel Core)

Multi-Processor servers use bus or switch to connect CPUs with main memory

- Bus: Only one processor can access bus at a time!
- Switch: Cache-Coherency traffic can “pollute” switch
- Scalability beyond 2–8 CPUs is a problem
- Dual core chips, small Itanium servers, NEC SX8
Shared memory computers: ccNUMA

ccNUMA architecture

Proprietary hardware concepts (e.g. Hypertransport/Opteron or NUMALink /SGI) provide single address space & cache coherency for physically distributed memory

- **Advantages:**
  - Scalable concept (systems up to 1024 CPUs are available)

- **Disadvantages:**
  - Cache Coherence hard to implement / expensive
  - Performance depends on access to local or remote memory (no flat view of memory!)
Shared memory computers: Some examples

- Dual CPU Intel Xeon node
- Dual Intel “Core” node
- Dual AMD Opteron node
- SGI Altix (HLRB2 @ LRZ)
Shared memory computers

Cache coherence

- Data in cache is only a copy of data in memory
  - Multiple copies of same data on multiprocessor systems
  - Cache coherence protocol/hardware ensure consistent data view
  - Without cache coherence, shared cache lines can become clobbered:

```
P1
  C1
  A1, A2

P2
  C2
  A1, A2

Bus

Memory

A1, A2

P1
Load A1
Write A1=0

P2
Load A2
Write A2=0

Write-back to memory leads to incoherent data

A1, A2  A1, A2  A1, A2

C1 & C2 entry can not be merged to:

A1, A2
```
Cache coherence protocol must keep track of cache line (CL) status.

P1

1. Request exclusive access to CL
2. Invalidate CL in C2
3. Modify A1 in C1

P2

1. Load A2
2. CL write back + Invalidate
3. Load CL to C2
4. Modify A2 in C2

C2 is exclusive owner of CL
Cache coherence can cause substantial overhead
- may reduce available bandwidth

Different implementations
- Snoopy: On modifying a CL, a CPU must broadcast its address to the whole system
- Directory, “snoop filter”: Chipset (“network”) keeps track of which CLs are where and filters coherence traffic

Directory-based ccNUMA can reduce pain of additional coherence traffic

But always take care:

Multiple processors should never write frequently to the same cache line (“false sharing”)!
Shared-Memory Parallelization
Parallel Programming with OpenMP

- “Easy” and portable parallel programming of shared memory computers: **OpenMP**
    - FORTRAN, C and C++ interfaces are defined
    - Supported by most/all commercial compilers, GNU starting with 4.2
    - Few free tools are available
  
  - OpenMP program can be compiled and executed on a single-processor machine just by ignoring the directives
    - API calls must be masked out though
  
  - Supports data parallelism

- Central concept of OpenMP programming: **Threads**
Shared Memory Model used by OpenMP

- Threads access globally shared memory
- Data can be shared or private
  - shared data available to all threads (in principle)
  - private data only to thread that owns it
- Data transfer transparent to programmer
- Synchronization takes place, is mostly implicit
OpenMP Program Execution
Fork and Join

- Program start: only master thread runs
- Parallel region: team of worker threads is generated ("fork")
- Synchronize when leaving parallel region ("join")
- Only master executes sequential part
  - Worker threads persist, but are inactive
- Task and data distribution possible via directives
- Usually optimal: 1 Thread per Processor
Hybrid parallelization on clustered SMPs

Node Performance = **OpenMP** + Low-Level Optimization

- Parallelized by library call (MPI)
- Multi-Threading (**OpenMP**)
- Low-Level Optimization

Inter-Node

Node

Single CPU

Message Passing

DO \( l = 1, I \)  
Inter-node parallelization (MPI)

DO \( j = 1, m \)  
Intra-node **OpenMP** processing

DO \( k = 1, n \)  
single processor execution

25.09.06  
hpc@rrze.uni-erlangen.de  
Shared-memory parallelization  
16
Basic OpenMP functionality

About Directives and Clauses

About Data

About Parallel Regions and Work Sharing
First example: Numerical integration

Approximate by a discrete sum

\[ \int_{0}^{1} f(t) \, dt \approx \frac{1}{n} \sum_{i=1}^{n} f(x_i) \]

where

\[ x_i = \frac{i - 0.5}{n} \quad (i = 1, \ldots, n) \]

We want

\[ \int_{0}^{1} \frac{4}{1 + x^2} \, dx = \pi \]

→ solve this in OpenMP

```fortran
program compute_pi
... (declarations omitted)

! function to integrate
f(a)=4.0_8/(1.0_8+a*a)

w=1.0_8/n
sum=0.0_8

do i=1,n
    x=w*(i-0.5_8)
    sum=sum+f(x)
enddo
pi=w*sum

... (printout omitted)
end program compute_pi
```
First example:
Numerical integration

\[ \pi = 0.0_8 \]
\[ w = 1.0_8/n \]

```c
!$OMP parallel private(x,sum)
sum=0.0_8

!$OMP do
do i=1,n
   x=w*(i-0.5_8)
   sum=sum+f(x)
enddo
!$OMP end do

!$OMP critical
pi=pi+w*sum
!$OMP end critical

!$OMP end parallel
```

concurrent execution by “team of threads”
worksharing among threads
sequential execution
OpenMP Directives
Syntax in Fortran

- Each directive starts with **sentinel** in column 1:
  - fixed source: !$OMP or C$OMP or *$OMP
  - free source: !$OMP

  followed by a **directive** and, optionally, **clauses**.

- For function calls:
  - conditional compilation of lines starting with !$ or C$ or *

Example:

```fortran
myid = 0
!$ myid = omp_get_thread_num()
```

- use include file for API call prototypes (or Fortran 90 module `omp_lib` if available)
OpenMP Directives
Syntax in C/C++

- **Include file**
  ```
  #include <omp.h>
  ```

- **pragma preprocessor directive:**
  ```
  #pragma omp [directive [clause ...]]
  structured block
  ```

- **Conditional compilation: Compiler’s OpenMP switch sets preprocessor macro**
  ```
  #ifdef _OPENMP
  ...
  do something
  #endif
  ```
OpenMP Syntax: Clauses

- Many (but not all) OpenMP directives support clauses
- Clauses specify additional information with the directive
- Integration example:
  - `private(x,sum)` appears as clause to the `parallel` directive
- The specific clause(s) that can be used depend on the directive
- Another example: `schedule(…)` clause
  - `static[,chunksize]`: round-robin distribution of chunks across threads (no chunksize: max. chunk size – default!)
  - `dynamic[,chunksize]`: threads get assigned work chunks dynamically; used for load balancing
  - `guided[,chunksize]`: like dynamic, but with decreasing chunk size (minimal size = chunksize); used for load balancing when dynamic induces too much overhead
  - `runtime`: determine by OMP_SCHEDULE shell variable
OpenMP parallel regions
How to generate a team of threads

- !$OMP PARALLEL and !$OMP END PARALLEL
  - Encloses a parallel region: All code executed between start and end of this region is executed by all threads.
  - This includes subroutine calls within the region (unless explicitly sequentialized)
  - Both directives must appear in the same routine.

- C/C++:
  
  #pragma omp parallel
  structured block
  
  No END PARALLEL directive since block structure defines boundaries of parallel region
OpenMP work sharing for loops

Requires thread distribution directive

\texttt{
!$\text{OMP DO} / !$\text{OMP END DO} \text{ encloses a loop which is to be divided up if within a parallel region ("sliced").} \\
\quad \text{all threads synchronize at the end of the loop body} \\
\quad \text{this default behaviour can be changed ...}
}

- Only loop \textit{immediately following} the directive is sliced
- C/C++:
  \texttt{
  \#pragma omp for [clause] \\
  for ( \ldots ) \{ \\
    \ldots \\
  \}
  }
- restrictions on parallel loops (especially in C/C++)
  - trip count must be computable (no do while)
  - loop body with single entry and single exit point
  - Use integers, not iterators als loop variables
Directives for data scoping: shared and private

- **Remember the OpenMP memory model?**
  - Within a parallel region, data can either be
  - **private** to each executing thread
    - each thread has its own local copy of data
  - or be
  - **shared** between threads
    - there is only one instance of data available to all threads
    - this does not mean that the instance is always visible to all threads!

- **Integration example:**
  - shared scope not desirable for x and sum since values computed on one thread must not be interfered with by another thread.
  - Hence:
    - !$OMP parallel private(x,sum)
Defaults for data scoping

- All data in parallel region is **shared**
- This includes **global** data (Module, COMMON)
- **Exceptions:**
  1. **Local** data within enclosed subroutine calls are **private**
     (Note: Inlining must be treated correctly by compiler!)
     unless declared with **SAVE** attribute
  2. **Loop variables** of parallel (“sliced”) loops are **private**
- Due to stack size limits it may be necessary to give large arrays the SAVE attribute
  - This presupposes it is safe to do so!
  - If not: make data dynamically allocated
  - For Intel Compilers: `KMP_STACKSIZE` may be set at run time (increase thread-specific stack size)
Changing the scoping defaults

- Default value for data scoping can be changed by using the `default` clause on a parallel region:

  ```
  !$OMP parallel default(private)
  ```

- Beware side effects of data scoping:
  Incorrect `shared` attribute may lead to race conditions and/or performance issues (“false sharing”).
  - Use verification tools.

- Scoping of local subroutine data and global data
  - is not (hereby) changed
  - compiler cannot be assumed to have knowledge

- Recommendation: Use

  ```
  !$OMP parallel default(none)
  ```

  to not overlook anything

**Not in C/C++**
Storage association of private data

- Private variables: **undefined** on entry and upon exit of parallel region

- Original value of variable (before parallel region) is **undefined** after exit from parallel region

- To change this:
  - Replace `private` by `firstprivate` or `lastprivate`

- Private variable within parallel region has **no storage association** with same variable outside region
Running an OpenMP program

- **Number of threads**: Determined by shell variable
  
  `OMP_NUM_THREADS`

- **Loop scheduling**: Determined by shell variable
  
  `OMP_SCHEDULE`

- Some implementation-specific environment variables exist (here for Intel):
  - `KMP_STACKSIZE`: configure thread-local stack size
  - `KMP_LIBRARY`: specify the strategy for releasing threads that have nothing to do
Common OpenMP pitfalls

- **Correctness**
  - **Deadlock**: Thread waits for resources that never become available
    - Write correct programs (tools help to detect deadlocks)
  - **Race condition**: Uncontrolled writes to shared variable
    - Use `private` clause

- **Performance**
  - **False sharing**: Frequent writes from different threads to same cache line
    - Insert padding, choose appropriate OpenMP schedule
  - **Load imbalance**: Different workloads assigned to different threads leads to idling CPUs
    - Use dynamic or guided schedule, rearrange workload
  - **OpenMP loop overhead**: Loop is too short to amortize the cost of starting a team of threads
    - Use programming techniques to fatten loop body
OpenMP parallelization of sparse MVM
Data parallelism for sparse MVM

- Parallelize the loop that treats consecutive elements of result vector (or consecutive matrix rows)

- General idea:

  - RHS vector is accessed by all threads
    - ... but this is shared memory, so it does not have to be stored multiple times!
OpenMP parallelization of CRS MVM

- **Parallelized loop is outer loop**

```c
!$OMP parallel do
do i = 1,Nr
    do j = row_ptr(i), row_ptr(i+1) - 1
        c(i) = c(i) + val(j) * b(col_idx(j))
    enddo
endo
endo
!$OMP end parallel do
```

- **Features**
  - Long outer loop
    - small OpenMP overhead
  - Variable length of inner loop
  - possible load imbalance
OpenMP parallelization of JDS MVM

- Parallelized loop is inner loop

```c
!$OMP parallel private(diag,diagLen,offset,i)
  do diag=1, zmax
    diagLen = jd_ptr(diag+1) - jd_ptr(diag)
    offset = jd_ptr(diag)
    !$OMP do
    do i=1, diagLen
      c(i) = c(i) + val(offset+i) * b(col_idx(offset+i))
    enddo
    !$OMP end do
  enddo
!$OMP end parallel
```

- Features
  - Long inner loop
  - No load imbalance problems
OpenMP parallelization of blocked JDS MVM

- Parallelization can now be pulled to outer loop

```plaintext
!$OMP parallel do private(block_start, block_end, i, diag, 
!$OMP& diagLen, offset)
do ib=1, maxDiagLen, blocklen
  block_start = ib
  block_end = min(ib+blocklen-1, maxDiagLen)
do diag=1, zmax
    diagLen = jd_ptr(diag+1)-jd_ptr(diag)
    offset = jd_ptr(diag)
    if(diagLen .ge. block_start) then
      do i=block_start, min(block_end, diagLen)
        c(i) = c(i)+val(offset+i)*b(col_idx(offset+i))
      enddo
    endif
  enddo
$OMP end parallel do
```

- Features
  - Least OpenMP overhead
  - Some load imbalance possible
Parallel sparse MVM: Scalability

- Scalability data for OpenMP version

![Bar chart showing MFlop/s vs. number of threads for different platforms and thread numbers. The chart indicates that something is obviously wrong.](chart.png)
Data locality in ccNUMA systems
Memory Locality Problems

- ccNUMA:
  - whole memory is transparently accessible by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential congestion (shared memory paths)

- How do we make sure that memory access is always as "local" and "distributed" as possible?
Coding for Data Locality

- In OpenMP the programmer must ensure that memory pages get mapped locally, i.e. data that is accessed from CPU n should reside in a local memory block.
- Rigorously apply the "Golden Rule": A memory page gets mapped into the local memory of the processor that first touches (reads or writes to) it!
- i.e. we have to take a closer look at initialization code.
- Locality is always observed on the page level:
  - Page sizes: 4kB, 16kB, sometimes larger.
- Some false (page) sharing at domain boundaries may be unavoidable.
Coding for Data Locality

- Simplest case: explicit initialization

```fortran
Integer, parameter :: N = 1000000
Real*8 A(N), B(N)

A = 0.d0

 !$OMP parallel do
 Do I = 1, N
   B(i) = function ( A(i) )
 End do
```

```fortran
Integer, parameter :: N = 1000000
Real*8 A(N), B(N)

 !$OMP parallel do
 Do I = 1, N
   A(i) = 0.d0
 End do

 !$OMP parallel do
 Do I = 1, N
   B(i) = function ( A(i) )
 End do
```
Coding for Data Locality

- Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so "localize" arrays before I/O

```fortran
Integer, parameter :: N = 1000000
Real*8 A(N), B(N)

READ(1000) A
!$OMP parallel do
Do I = 1, N
    B(i) = function ( A(i) )
End do
```

```fortran
Integer, parameter :: N = 1000000
Real*8 A(N), B(N)

!$OMP parallel do
Do I = 1, N
    A(i) = 0.d0
End do
READ(1000) A
!$OMP parallel do
Do I = 1, N
    B(i) = function ( A(i) )
End do
```
Coding for Data Locality

- Required condition: **OpenMP loop schedule of initialization must be the same as in all computational loops**
  - **best choice:** *static!* Specify explicitly on all NUMA-sensitive loops, just to be sure...
  - imposes some constraints on possible optimizations (e.g. load balancing) → some sensibly large chunk size may be better than plain *static*

- How about **global objects**?
  - better not use them
  - if communication vs. computation is favorable, might consider *properly placed copies* of global data
  - in C++, **STL allocators** provide an elegant solution
Data locality in parallel sparse MVM

- No code change in MVM loop required (apart from static schedule)
- CRS
  - Initialization of arrays val[], c[], b[], row_ptr[] and col_idx[] must be done in parallel

```plaintext
!$OMP parallel do private(start,end,j)
!$OMP& schedule(static)
do i=1,Nr
    start = row_ptr(i)
    end = row_ptr(i+1)
    do j=start,end-1
        val(j) = 0.d0
        col_idx(j)= 0
    enddo
enddo
```

- Similar for JDS
Parallel sparse MVM
Doing it right on ccNUMA

- Correct placement leads to acceptable scalability

![Graph showing MFlop/s vs # Threads for different architectures]

JDS scalability worse than CRS – why?

No difference for Core architecture (UMA)
Coding for Data Locality:
C++ issues

- Bck to C++: Don't forget that constructors tend to touch the data members of an object. Example:

```cpp
class D {
    double d;
public:
    D(double _d=0.0) throw() : d(_d) {}  
    inline D operator+(const D& o) throw() {
        return D(d+o.d);
    }
    inline D operator*(const D& o) throw() {
        return D(d*o.d);
    }
    ...
};
```

→ placement problem with

```cpp
D* array = new D[1000000];
```
Coding for Data Locality:
C++ issues

- Solution: Provide overloaded `new` operator or special function that places the memory before constructors are called (``PAGE_BITS`` = base-2 log of pagesize)

```cpp
template <class T> T* pnew(size_t n) {
    size_t st = sizeof(T);
    int ofs, len=n*st;
    int i, pages = len >> PAGE_BITS;
    char *p = new char[len];
    #pragma omp parallel for schedule(static) private(ofs)
    for(i=0; i<pages; ++i) {
        ofs = static_cast<size_t>(i) << PAGE_BITS;
        p[ofs]=0;
    }
    #pragma omp parallel for schedule(static) private(ofs)
    for(ofs=0; ofs<n; ++ofs) {
        new(static_cast<void*>(p+ofs*st)) T;
    }
    return static_cast<T*>(m);
}
```

parallel first touch

placement
new!
Coding for Data Locality:
NUMA allocator for parallel first touch

```
template <class T> class NUMA_Allocator {
public:
    T* allocate(size_type numObjects, const void *
                localityHint=0) {
        size_type ofs,len = numObjects * sizeof(T);
        void *m = malloc(len);
        char *p = static_cast<char*>(m);
        int i,pages = len >> PAGE_BITS;
#pragma omp parallel for schedule(static) private(ofs)
        for(i=0; i<pages; ++i) {
            ofs = static_cast<size_t>(i) << PAGE_BITS;
            p[ofs]=0;
        }
        return static_cast<pointer>(m);
    }

...}

Application:
vector<double,NUMA_Allocator<double> > x(1000000)
```
References

- OpenMP Home: Specifications, resources, mailing list, events
  http://www.openmp.org/


- M. Austern: *What are allocators good for?*
  Dr Dobb’s Journal, April 2003
  http://www.ddj.com/dept/cpp/184403759
BACKUP
Application: DMRG – Parallelization of sparse MVM in superblock diagonalization

- **Sparse MVM: Sum over dense matrix-matrix multiplies!**

\[
\sum_{i',j'} H_{i,j;i',j'} \psi_{i',j'} = \sum_{\alpha} \sum_{i'} A_{ii'}^{\alpha} \sum_{j'} B_{jj'}^{\alpha} \psi_{i',j'}
\]

- However, \(A\) and \(B\) may contain only a few nonzero elements, e.g. if conservation laws (quantum numbers) have to be obeyed

- To minimize overhead, an additional loop (running over nonzero blocks only) is introduced

\[
H \psi = \sum_{\alpha} \sum_{k} (H \psi)^{\alpha}_{L(k)} = \sum_{\alpha} \sum_{k} A_{\alpha}^{\alpha} \psi_{R(k)} [B^T]_{k}^{\alpha}
\]
Sparse MVM in DMRG

Implementation of sparse MVM - pseudocode

\[ H\psi = \sum_{\alpha} \sum_{k} A_{k}^{\alpha} \psi_{R(k)} \left[ B^T \right]_{k}^{\alpha} \]

// W: wavevector ; R: result
for (\(\alpha=0\); \(\alpha < \text{number\_of\_hamiltonian\_terms}\); \(\alpha++\)) {
    \text{term} = \text{hamiltonian\_terms}[\alpha];
    for (k=0; k < \text{term.number\_of\_blocks}; k++) {
        li = \text{term}[k].left\_index;
        ri = \text{term}[k].right\_index;
        temp\_matrix = \text{term}[k].B.transpose() * W[ri];
        R[li] += \text{term}[k].A * temp\_matrix;
    }
}

Data dependency!

Matrix-matrix multiply

Parallel loop !?
DMRG: OpenMP Parallelization

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

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

Preparation

// 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

```
// 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 ($\alpha,k$) loop

Protect each block of result vector $R$ with locks
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%
DMRG : OpenMP Parallelization
Further improvements

- 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)
  - Observe vendor advice