First Experiences with Intel Cluster OpenMP

Georg Hager
Regionales Rechenzentrum Erlangen (RRZE)

19.05.2006
CLOMP Workshop, HLRS

Overview

- Systems used
  - EM64T (dual Nocona) with Gbit Ethernet and Infiniband, Debian 3.1 (Sarge)
  - Itanium2 (HP zx6000) with Gbit Ethernet, SLES9p3
  - Opteron would be a nice exercise, but CLOMP doesn’t work on AMD...

- Basic numbers: Triad tests

- Application: Lattice-Boltzmann code
  - influence of algorithmic details
  - data layout considerations

- Odds and ends

General Remarks

- CLOMP == "extreme" ccNUMA
  - very long latencies, expensive non-local access
  - page replications can lead to memory problems
  - but: placement is handled “automatically”

- Consequence: A well-optimized, ccNUMA-aware OMP code that scales well on Altix does not necessarily scale well with CLOMP
  - example: boundary code must be optimized for local access

- Good stability on all systems with latest CLOMP release
- No problems and good performance with IP over IB
  - native IB not working yet

Problems (RRZE-specific?)

- Memory footprint is about 2.5 times larger than expected from serial code (270MB instead of 61MB for vector triad)
  - Partially resolved by Intel (Jim C.)
- Huge core dumps even with small sharable heap and resident memory (2.4GB core with 200MB code)
- Reproducible hangs on entry to parallel region when OMP_NUM_THREADS smaller than number of hosts in hostfile (only for LBMKernel)
## CLOMP first experiences

### Parallel Triad $A(\cdot) = B(\cdot) + C(\cdot)D(\cdot)$

- **Three flavors**
  1. **Standard triad, OMP parallel**
     ```c
     #pragma omp parallel for
     for(i=0; i<N; i++)
     a[i] = b[i] + c[i]*d[i];
     ```
  2. **Throughput triad (separate local arrays on each thread)**
     ```c
     #pragma omp parallel
     sub_triad(N);
     ```
  3. **Padded triad**
     ```c
     #pragma omp parallel
     do_triad(N[myID], start[myID], a, b, c, d)
     ```

### Standard Triad on GBit Ethernet vs. IP over IB (1T/node)

- Report only on IP over IB in the following

### Filled vs. Half-filled nodes

- **2 ways to „fill the node“**
  1. Keep unique names in hostfile and use 2 „real“ OpenMP threads per node with `--process_threads=2`
  2. Duplicate names in hostfile and use `--process_threads=1`

- **Observations**
  - Breakdown of performance compared to the half-filled case for large $N$
  - Improvement with OpenMP for medium-sized arrays `--process_threads=2`: quite erratic performance data
  - Breakdown was actually expected (the same happens on single node with pure OpenMP)

- **Erratic behaviour**
  - Influence of „loaded“ switch? (improbable)
  - Threads losing CPU affinity?

### Threads vs. processes on node

- Erratic behaviour
  - Influence of „loaded“ switch? (improbable)
  - Threads losing CPU affinity?
Pinning of threads

- Performance results seem quite erratic when using all available CPUs on a node
- Possible remedy? → pin threads to CPUs
  - using PLPA for portability reasons

```c
#pragma omp parallel
#pragma omp critical
{
    if(PLPA_NAME(api_probe)() != PLPA_PROBE_OK) {
        cerr << "PLPA failed!" << endl;
    } else {
        plpa_cpu_set_t msk;
        PLPA_CPU_ZERO(&msk);
        PLPA_CPU_SET(omp_get_thread_num() & 1, &msk);
        PLPA_NAME(sched_setaffinity)((pid_t)0, (size_t)32, &msk);
    }
}
```

Results for pinned triad (4 and 8 threads)

- 4T: no change
- 8T Numbers get less erratic, but performance is worse
- Observation: IB completion thread (`ts_ib_completion`) frequently using CPU time

Lattice Boltzmann Method

- Numerical Method for Simulation of Fluids
  - Stream-Collide (Pull-Method)
    Get the distributions from the neighboring cells in the source array and store the relaxed values to one cell in the destination array
  - Collide-Stream (Push-Method)
    Take the distributions from one cell in the source array and store the relaxed values to the neighboring cells in the destination array

D3Q19 model:

Two Grids:

Compressed Grid

(not used here):

LBMKernel – Code Structure for Collide-Stream Step

```c
double precision f(0:xMax+1, 0:yMax+1, 0:zMax+1, 0:18, 0:1)
!$OMP PARALLEL DO
do z=1,zMax
    do y=1,yMax
        do x=1,xMax
            if( fluidcell(x,y,z) ) then
                LOAD f(x,y,z, 0:18, t)
                ...Relaxation (complex computations)...
                SAVE f(x  ,y  ,z  , 0,t+1)
                SAVE f(x+1,y+1,z , 1,t+1)
                SAVE f(x  ,y+1,z  , 2,t+1)
                SAVE f(x-1,y+1,z , 3,t+1)
                ...
                SAVE f(x  ,y-1,z-1,18,t+1)
            endif
        enddo
    enddo
enddo
```
LBMKernel

- Scalability beyond 2 nodes was very bad with standard code
- proper choice of geometry (long thin channel) can restore scalability
  - not a general solution
- Solution: bounceback (boundary) routine was not properly optimized for local access
  - on ccNUMA, this is a negligible effect for small obstacle density (n²)
  - on CLOMP, it is devastating
- Still: indexing has significant impact on performance
  - "push" vs. "pull" algorithm
  - parallelized dimension should be the outermost one to minimize false sharing: (i,j,v,t,k) better than (i,j,k,v,t)
- Might profit from ghost layers, but is this still OpenMP???

DMRG

- Large C++ code, OpenMP parallelized
  - good scalability not really expected, but a good example for porting
  - cache-bound, so not optimized for ccNUMA
- Important issues:
  - use new (kmp_sharable) for dynamic objects used in parallel regions
  - derive classes from kmp_sharable_base if dynamic objects are used in parallel regions
- Possible problem with global objects (still under investigation)

Conclusions

- Cluster OpenMP is an interesting programming experience
- Imagine a ccNUMA machine with automatic page migration (wow!) and an awfully slow network
  - If something strange happens (performance-wise), use profiler by all means
  - Otherwise (with OMP) negligible boundary effects may become dominant with CLOMP
- With CLOMP, performance results tend to be more scattered than usual
- Looking forward to AMD-enabled versions…