C++ programming techniques for High Performance Computing on systems with non-uniform memory access using OpenMP

Holger Stengel, diploma thesis supervised by Dr. Georg Hager, RRZE

Abstract

This work develops programming methodologies for C++ that respect the need for optimal NUMA page placement in OpenMP. An overloaded new[] operator is presented that guarantees proper placement for arrays of objects. Along the same lines, the STL vector<T> class can be endowed with an allocator class argument that serves the same purpose.

Page Placement

Distributing data across locality domains (LDs) in a way that enables concurrent, local access makes a huge difference for memory-bound codes.

Fig. 2: Performance penalty for vector triad: Locality and bandwidth problems (HP DL585)

How can proper placement be accomplished?

The Golden Rule of ccNUMA: A memory page is mapped to the locality domain of the processor core that touches it, i.e. writes to it first (first touch policy).

Solution in standard languages (Fortran, C): Exploit first-touch policy on data initialization:

double *a = new double[N];
#pragma omp schedule(static)
for(int i = 0; i < N; ++i)
    a[i] = sin(i);
#pragma omp schedule(static)
for(int i = 0; i < N; ++i)
    if(obscure) dummy(a[i], b[i], c[i], d[i]);

The static schedule is vital to controlling the map of threads to iterations. A possible chunk size should encompass whole pages if possible.

Is there a problem with NUMA placement in C++?

NUMA-Unfriendly C++

Arrays of objects are constructed sequentially by design, leading to page placement in a single LD if the ctor initializes member data.

Class D

double d;
public:
    D() : d(0) {}

D *array[1000000];

STL vector<T> containers initialize data by calling uninitialized_fill() or similar:

std::vector<double> v[1000000];

In both cases, there is no way to influence the construction of pages in a similar way as with standard C Arrays, i.e. by inserting parallelization pragmas.

Possible solutions:

- Overload operator new[] for each class
- Use optional allocator template argument for std::vector<> [2]
- Design high-performance, configurable NUMA-aware container
- Account for locality constraints via segmented data structures

Overloading operator new[]

Responsible for allocating raw dynamic storage; objects are constructed elsewhere using placement new. Example for class D:

void D::operator new[](size_t n) throw(std::bad_alloc) {
    void *m;
    if(alocloc(m))
        throw std::bad_alloc;
    char *p = static_cast<char*>(m);
    #pragma omp schedule(static)
    for(int i = 0; i < n; ++i) {
        // non-destructive f.t.
        char a = p[i];
        p[i] = a;
    }
    return m;
}

A Fully NUMA-aware Container

Memory is naturally segmented on NUMA and multicore machines. Segmented memory creates memory blocks shared between threads.

- Solution: Segmentation-aware container with configurable padding prevents boundary effects.
- Pot: Final performance of overloaded operator new[]

Fig. 5: .new vector<> provides high speed operator new() and proper page placement

Benefits of numa_vector<>:

- Supports allocator concept
- More efficient operator new [] compared to std::vector<> [1]
- Supports iterator concept for compatibility with STL algorithms
- Includes valarray<> features
- Provides NUMA-aware resize() function
- Operators can take arguments with different allocators

A Segmented Container

Memory is naturally segmented on NUMA and multicore machines. Segmented memory creates memory blocks shared between threads.

- Solution: Segmentation-aware container with configurable padding prevents boundary effects.
- Pot: Final performance of overloaded operator new[]

Fig. 6: Data layout of seg_array<>:

- Introduction of segmented iterator (local and segment iterator)
- Traits class supports dispatching algorithms [5]

Fig. 7: seg_array<> allows low level algorithms with optimal performance

Disadvantage: Issues with alignment, prefetching and memory consumption.

Fig. 8: Data layout of segment iterator

Conclusion

Correct page placement is essential for the performance of memory-bound parallel algorithms on ccNUMA architectures. We have presented different correct methods to achieve NUMA placement semantically in a C++ context. Optimized containers were provided that outperform std::vector<> in several ways.


