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

University of Applied Sciences
GEORG-SIMON-OHM FACHHOCHSCHULE NÜRNBERG

Abstract

This work develops programming methodologies for C++ that respect the need for optimal NUMA page placement in OpenMP code. An overloaded `new[]` operator is presented that guarantees proper placement for arrays of objects. Along the same lines, the STL `vector<>` class can be endowed with an allocator class argument that serves the same purpose. The disadvantages of `std::vector<>` in terms of performance and usability in a NUMA setting are circumvented by developing a special `numa_vector<>` container which is compatible with all STL algorithms. Finally, a container with a segmented, padded data structure, including appropriate iterators, allows one to make generic algorithms aware of data segmentation of any kind (including NUMA) without sacrificing performance.

ccNUMA

In High Performance Computing (HPC), shared-memory systems with cache coherent non-uniform memory access (ccNUMA) characteristics are becoming more common:

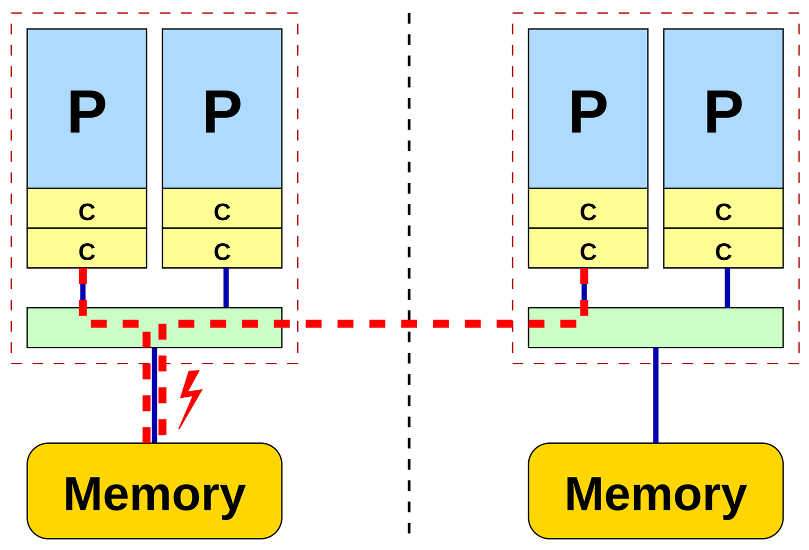

Fig. 1: *Two-socket Opteron node with ccNUMA via HyperTransport and two locality domains*

Advantages:

- Memory bandwidth scales with number of locality domains
- Low cost

Challenges:

- Cache coherence traffic has larger latencies than on UMA
- Non-local access has lower bandwidth and larger latency
- Improper page placement can lead to bandwidth bottlenecks

Vector Triad

The performance of the parallel vector triad is used to pinpoint bandwidth-related issues [3]:

```
for(int j = 1; j < NITER; ++j) {
#pragma omp parallel for
   for(int i = 0; i < N; ++i) {
      a[i] = b[i] + c[i] * d[i];
   }
   if(obscure) dummy(a,b,c,d);
}
```

Properties:

- Code balance is 2 Words/Flop without RFO and 2.5 Words/Flop with RFO. This is well beyond the machine balance of any current microprocessor (0.05–0.2 Words/Flop).
- Triad shows a rich set of performance features on different architectures.

The current OpenMP [1] standard has no elements to implement locality constraints. Moreover, OS activities can fill LDs (e. g. with buffer space) and prevent applications from using local memory.

Does the vector triad performance scale with core count for large N?

[1] `http://www.openmp.org`

Page Placement

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

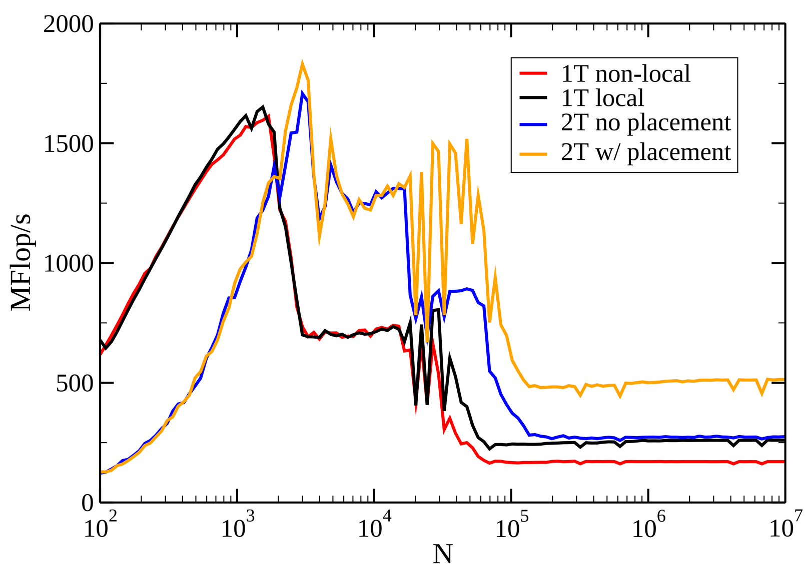

Fig. 2: *Performance penalty for vector triad: Locality and bandwidth problems (HP DL585)*

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], *b = ...;
#pragma omp parallel for \
                schedule(static)
for(int i = 0; i < N; ++i)
   a[i]=b[i]=c[i]=d[i]=1.0;
for(int j = 1; j < NITER; ++j) {
#pragma omp parallel for \
                schedule(static)
   for(int i = 0; i < N; ++i)
      a[i]=b[i]+c[i]*d[i];
   if(obscure) dummy(a,b,c,d);
}
```

The static schedule is vital to control the mapping 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 = new D[10000000];
```

STL `vector<>` containers

initialize data by calling `uninitialized_fill()` or similar:

```
std::vector<double> v(10000000);
```

In both cases, there is no way to influence the construction process 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<>` [4]
- Design high-performance, configurable NUMA-aware container
- Account for locality constraints via segmented data structures

[2] G. Hager, G. Wellein: *Concepts of High Performance Computing* (Regionales Rechenzentrum, Erlangen), 2007.

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(!(m=malloc(n)))
    throw std::bad_alloc;
  char *p = static_cast<char*>(m);
#pragma omp parallel for \
            schedule(static)
  for(int i=0; i < n ; ++i) {
    // non-destructive f.t.
    char a = p[i];
    p[i]=a;
  }
  return m;
}
```

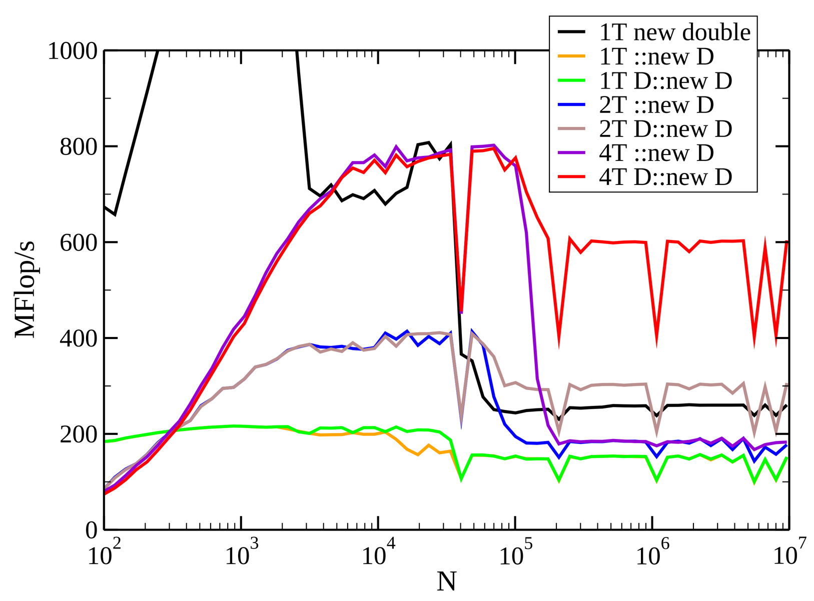

Fig. 3: *Benefits of overloaded `operator new[]` for parallel vector triad performance using `class D`*

Disadvantage: Dynamic (heap) storage referenced by objects is not first-touched correctly — placement `new` call is not under programmer's influence.

Allocator Template for `std::vector<>`

Allocator template arguments for STL containers provide a way of customizing raw memory allocation:

```
std::vector<Type,Allocator<Type> > v(N);
```

Most important methods of custom allocators:

- `allocate()` allocates raw memory, including NUMA placement (see above)
- `construct()` uses placement `new` to construct one object at a certain address
- `destroy()` calls the dtor of an object at certain address
- `deallocate()` frees raw memory

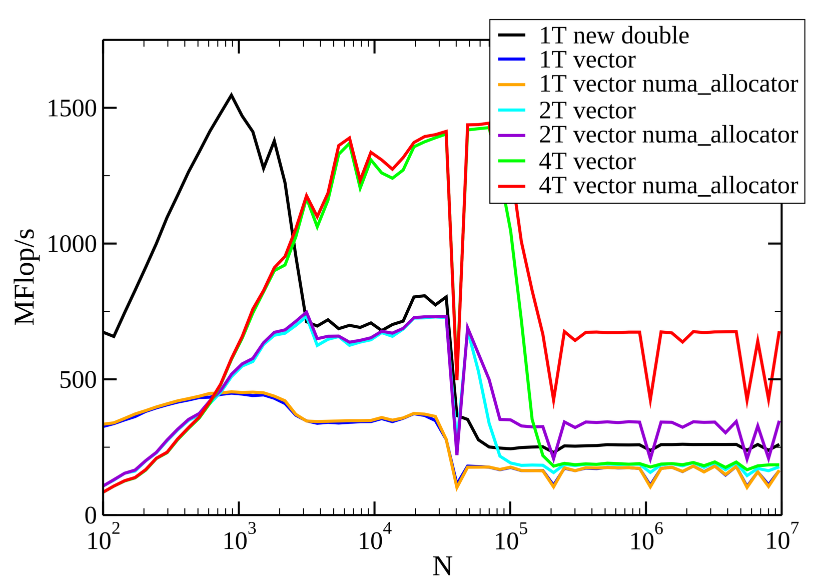

Fig. 4: *Benefits of customized NUMA allocator for `std::vector<>`*

Disadvantages

- As with overloaded `operator new[]`, objects with dynamic data are problematic because the loop that calls `numa_allocator::construct()` is inaccessible.
- `std::vector<>` has too many NUMA-unsuitable features like capacity vs. size

[3] W. Schönauer: *Scientific Supercomputing - Architecture and Use of Shared and Distributed Memory Parallel Computers* (self-edition, Karlsruhe), 2000.

A Fully NUMA-aware Container

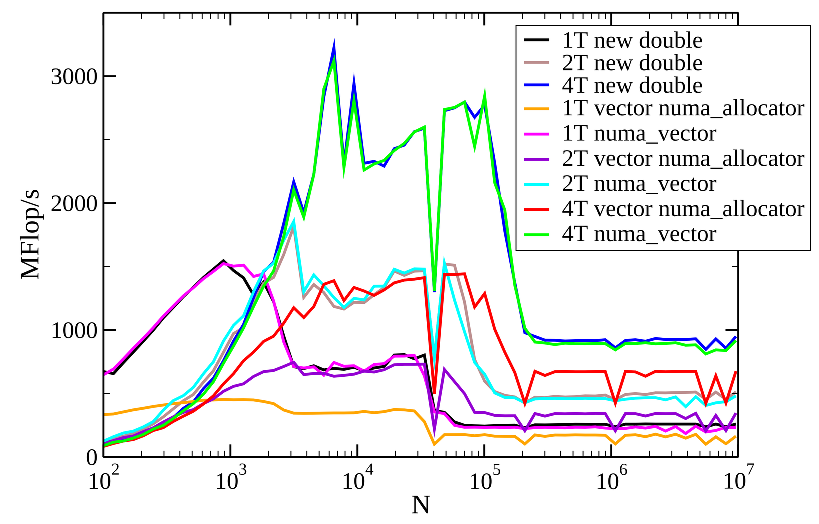

Fig. 5: *`numa_vector<>` provides high speed `operator[]` and proper page placement*

Benefits of `numa_vector<>`

- Supports allocator concept
- More efficient `operator[]` (compared to `std::vector<>`)
- 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
- But: Bad performance of overloaded `operator++`

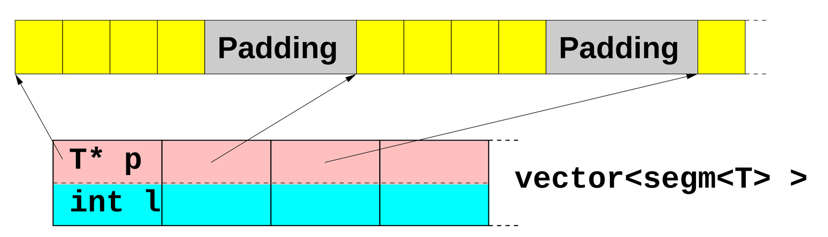

Fig. 6: *Data layout of `seg_array<>`*

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

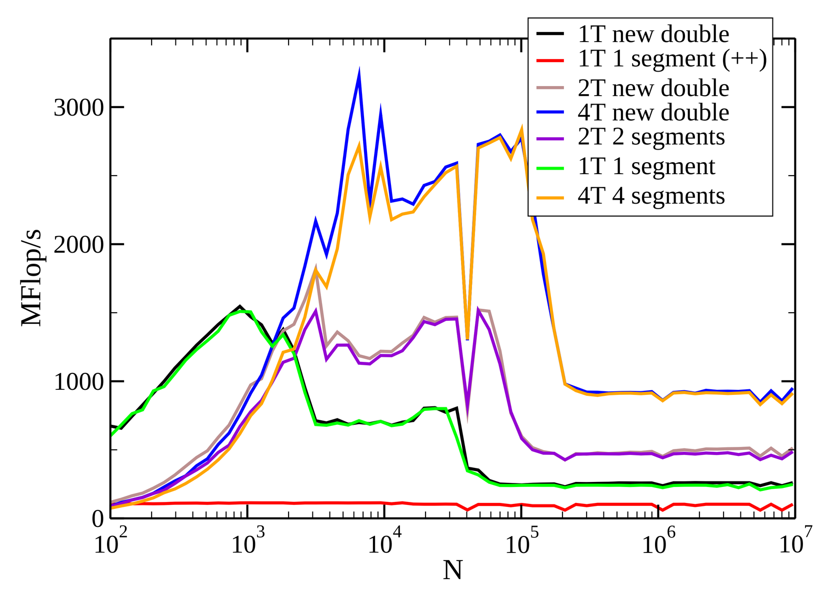

Fig. 7: *`seg_array<>` allows low level algorithms with optimal performance*

Disadvantage: Issues with alignment, prefetching and memory consumption.

Conclusion

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

[4] C. Terboven, D. an Mey: *OpenMP and C++* Proceedings of IWOMP2006 - International Workshop on OpenMP, Reims, France, June 12-15, 2006.

[5] M. H. Austern: *Segmented Iterators and Hierarchical Algorithms* (in M. Jazayeri, R. G. K. Loos, and D. R. Musser (ed.), Generic programming: International Seminar on Generic Programming, Castle Dagstuhl, Springer), 2001.