LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-426239

A Proposal for User-defined Reductions in OpenMP

A. Duran, R. Ferrer, M. Klemm,
B. R. de Supinski, E. Ayguade

March 23, 2010

6th International Workshop on OpenMP
Tsukuba, Japan
June 14, 2010 through June 16, 2010

Disclaimer

A Proposal for User-defined Reductions in OpenMP

Alejandro Duran,[1] Roger Ferrer,[1] Michael Klemm,[2]
Bronis R. de Supinski,[3] and Eduard Ayguade[1]

[1] Barcelona Supercomputing Center [2] Intel Corporation
{alex.duran,roger.ferrer,eduard}@bsc.es michael.klemm@intel.com

[3] Lawrence Livermore National Laboratory
bronis@llnl.gov

Abstract. Reductions are commonly used in parallel programs to produce a global result from partial results computed in parallel. Currently, OpenMP only supports reductions for primitive data types and a limited set of base language operators. This is a significant limitation for those applications that employ user-defined data types (e. g., objects). Implementing manual reduction algorithms makes software development more complex and error-prone. Additionally, an OpenMP runtime system cannot optimize a manual reduction algorithm in ways typically applied to reductions on primitive types. In this paper, we propose new mechanisms to allow the use of most pre-existing binary functions on user-defined data types as User-Defined Reduction (UDR) operators. Our measurements show that our UDR prototype implementation provides consistently good performance across a range of thread counts without increasing general runtime overheads.

1 Introduction

OpenMP [16] is a well-known and widespread programming model for the development of parallel applications on shared-memory platforms. It allows parallel and sequential implementations to co-exist in a single code base by using directives that tell the compiler which parts of the code to parallelize. Non-OpenMP compilers safely ignore the parallelization hints and emit an executable for sequential execution. In practice, OpenMP does not always keep its single-source promise since programmers often must modify their sequential code to overcome OpenMP's limitations.

We focus on OpenMP's lack of support for arbitrary reduction operators on arbitrary data types in this paper. Programmers use reductions to produce a global result from partial results computed in parallel. OpenMP currently only supports reductions for primitive data types and a limited set of base language operators. If the program computes a reduction on a user-defined data type or with a more complex operator, the programmer must implement the reduction algorithm manually. This limitation makes errors likely and complicates program maintenance by requiring repeated implementation of this common design pattern. Performance may also suffer since the OpenMP implementation can no longer adapt the reduction algorithm to the specific aspects of the execution (e. g., thread count or architecture).

In this paper, we propose extensions to OpenMP that eliminate this limitation. Our solution provides an additional OpenMP declarative directive `declare reduction` that specifies that a binary function (a *UDR function*) can be used as a *User-defined Reduction* (UDR). We also extend OpenMP's `reduction` clause to accept UDR operators in addition to built-in OpenMP operators.

The remainder of the paper is organized as follows. The following section discusses UDR mechanisms in other parallel programming languages. We then elaborate on the limitations of the current OpenMP specification in Section 3. Section 4 then details our proposal for UDR support in OpenMP. Finally, we evaluate performance of our proposal in Section 5. Overall, we show that our prototype implementation outperforms many hand-coded UDRs, particularly with large thread counts, without implying other overheads.

2 Related Work

Parallel programming frequently requires aggregation of local (partial) results into a global result. While low-level threading APIs such as POSIX threads [7], Windows Threads [13], Java Threads [15], or C# Threads [12] allow programmers to implement UDRs manually, other parallel programming languages provide a better, higher level approach.

Google's MapReduce API [4] provides a parallelization API that distributes work and accepts user-supplied reducers. Although MapReduce supports object-oriented languages, it only processes key-value data. Our OpenMP UDR proposal reflects the philosophy of OpenMP by using compiler directives to define UDRs, which supports all OpenMP base languages. Further, we directly support the variety of data types available in those languages.

MPI [14] includes UDR support. While our approach accepts a variety of binary functions, MPI UDRs are restricted to a special function signature. The programmer must provide corresponding wrapper functions to reuse existing functions. Additionally, we use a declarative syntax to define UDRs, whereas MPI requires special allocate and free function calls to inform the runtime about UDRs.

ZPL [5] relies on overloading for the specification of associative and commutative UDRs. One signature of the function returns the identity element while a second signature implements the actual reduction operator. OpenMP/Java [10] extends OpenMP with a `Reducer` interface to define UDRs. Similarly to ZPL, the interface requires separate methods to return the identity and to reduce two values. Cilk++ [6] follows a similar approach that defines special classes that implement the reduction semantics. Unlike these UDR approaches, we use explicit clauses of the UDR declaration to specify the identity element and the reduction operator and, thus, support all OpenMP base languages.

TBB [17] parallelizes C++ programs through templates to which reductions can be added as methods. PPL [11] also supports UDRs through `combinable` objects. While TBB and PPL provide UDRs, they only cover C++ programs; our approach is more generic in that it also targets C and Fortran. Our declaration syntax also provides better separation of concerns as a UDR can be reused in any parallel region, whereas

```
1 void example(double *array, size_t N) {
2     double sum = 0.0;
3     double prd = 1.0;
4 #pragma omp parallel for reduction(+:sum) reduction(*:prd)
5     for (size_t i = 0; i < N; i++) {
6         sum += array[i];
7         prd *= array[i];
8     }
9 }
```

Fig. 1. Simple reduction example with two reduction variables

TBB and PPL programmers must re-implement the reduction method in all functors, effectively adding redundant code to the application.

3 Costs of the Lack of UDR Support in OpenMP 3.0

OpenMP 3.0 only provides a standard set of reduction operators (e. g., addition) that operate on built-in primitive data types (e. g., `double`) of the corresponding base language. Fig. 1 shows an example of an OpenMP `parallel` construct (line 4) that performs simple sum and product reductions on the variables `sum` and `prd`.

In the example, the variables are of the primitive data type `double`. If we change them to a user-defined data type such as `complex_t`[1] we can no longer use OpenMP reductions. Instead, we must manually implement the reduction algorithms using one of the many ways to write a parallel reduction. Fig. 2 presents an efficient reduction algorithm that has little overhead for small thread counts.

The code in Fig. 2 first determines an upper bound of the possible number of participating threads (line 9). We then declare temporary arrays for each reduction variable to hold the intermediate local results of each thread (lines 11–12)[2].

Each thread has a private copy of the reduction variables `sum` and `prd`. Since these private copies are in different stacks, they cannot cause false sharing. Each thread must initialize its copies with the appropriate identity values (lines 16 and 17). Each thread then computes a local reduction for the array elements corresponding to its iterations of the `for` loop (lines 18–22). After executing the loop, each thread stores its partial results in the temporary arrays (line 23 and 24). After the parallel region, the master thread iterates over all partial results in the temporary arrays and produces the final result of the computation (lines 26–29).

Although this implementation performs well for small thread counts, larger thread counts might benefit from a tree-based reduction.Tree-based reductions are significantly more complicated and require a much higher coding effort. Switching between these implementations would require even more complex code, which must be repeated for every UDR in the OpenMP application. Although the pattern could be provided by a

[1] We use this type for explanatory purposes despite the `_Complex` primitive type in C99.

[2] For simplicity, the code shown could allocate more space than necessary since some OpenMP threads may not participate in the `parallel` region.

```
1 complex_t complex_add(complex_t a, complex_t b);
2 complex_t complex_mul(complex_t a, complex_t b);
3
4 void example(complex_t *array, size_t N) {
5     int nthreads;
6     complex_t sum = {0.0, 0.0}, prd = {1.0, 0.0};
7     complex_t *part_sum, *part_prd;
8
9     nthreads = omp_get_max_threads();
10
11    complex_t part_sum[nthreads];
12    complex_t part_prd[nthreads];
13
14 #pragma omp parallel shared(part_sum,part_prd) private(sum,prd)
15    {
16        sum = {0.0, 0.0};
17        prd = {1.0, 0.0};
18 #pragma omp for
19        for (size_t i = 0; i < N; i++) {
20            sum = complex_add(sum, array[i]);
21            prd = complex_mul(prd, array[i]);
22        }
23        part_sum[omp_get_thread_num()] = sum;
24        part_prd[omp_get_thread_num()] = prd;
25    }
26    for (int thr = 0; thr < nthreads; thr++) {
27        sum = complex_add(sum, part_sum[thr]);
28        prd = complex_mul(prd, part_prd[thr]);
29    }
30 }
```

Fig. 2. Programming pattern for user-defined reductions in OpenMP 3.0

parametrized library function, direct OpenMP support for UDRs would be less error-prone and more efficient.

Fig. 3 shows how our proposal simplifies this example. In lines 4 and 5, declaration pragmas inform the OpenMP compiler about UDR operators on the type `complex_t` and supply the corresponding identity values. After definition, the `reduction` clause can use these UDR operators (line 9), resulting in code almost identical to that for the `double` primitive type of Fig. 1.

4 User-defined Reductions for OpenMP

This section explores the design space for user-defined reductions and presents our `declare reduction` directive and the modifications to the current `reduction` clause. We then discuss extensions that support UDRs on array types and that more tightly integrate them with object-oriented languages.

4.1 Design rationale

The UDR language extension is subject to several crucial design requirements. First, it must follow the OpenMP directive-based philosophy. Second, the UDR feature must blend well with all OpenMP base languages and reflect their specifics while maintaining

```
1 complex_t complex_add(complex_t a, complex_t b);
2 complex_t complex_mul(complex_t a, complex_t b);
3
4 #pragma omp declare reduction(complex_add:complex_t) identity({0.0,0.0})
5 #pragma omp declare reduction(complex_mul:complex_t) identity({1.0,0.0})
6
7 void example(complex_t *array, size_t N) {
8     complex_t sum = {0.0, 0.0}, prd = {1.0, 0.0};
9 #pragma omp parallel for reduction(complex_add:sum) reduction(complex_mul:prd)
10    for (size_t i = 0; i < N; i++) {
11        sum = complex_add(sum, array[i]);
12        prd = complex_mul(prd, array[i]);
13    }
14 }
```

Fig. 3. Example of Fig. 2 rewritten with user-defined reductions

a common syntax across them. Third, the mechanism must express UDRs without any unnecessary syntax bloat. Fourth, the definition should allow for efficient implementations when used with any parallel loop schedule and support common optimizations.

The OpenMP compiler needs two pieces of information to implement a reduction: the identity element and the implementation of the operator. It needs the operator's identity value to initialize temporary variables that hold intermediate results. The implementation must combine two input values into one output value. The compiler generates code that invokes the reduction operator whenever the reduction algorithm aggregates values from different threads. The OpenMP specification provides this information for the reductions supported in OpenMP 3.0, while programmers must supply it for UDRs.

We could simply extend the existing `reduction` clause, which would not add any idioms to OpenMP. However, programmers would have to supply the above information at every `reduction` clause that works on a user-defined data type. This unnecessary repetition would increase the likelihood of errors at the `reduction` clauses.

Thus, we split the UDR definition into two parts: UDR *declaration* and UDR *usage*. At the declaration, programmers describe UDRs by specifying the UDR operator and the identity value. OpenMP-enabled libraries can safely incorporate UDR declarations for their data types in C or C++ header files or Fortran modules. At UDR usage, programmers supply the declared UDR name in a `reduction` clause as the operator.

In contrast to designs for UDRs in other programming models, one of our main design principles is code reuse. We explicitly allow programmers to reuse existing binary functions without the need for any wrapper mechanisms that adapt existing code interfaces to UDR requirements. Most OpenMP programs stem from a sequential code base with a set of operators on user-defined data types. These operators often include functions with two input values and one output value and that UDRs can reuse. Thus, our mechanism blends well with OpenMP's principle of incremental parallelization.

A sequential loop, such as that corresponding to the example in Fig. 1, combines array elements in the sequential iteration order. If the OpenMP implementation assigns a single chunk of iterations to each thread, then the reductions performed within each thread will be subsequences of the sequential iteration order. If the compiler combines these temporary values in the chunk order, the overall reduction order is simply a reas-

sociation of the original computation. Thus, we require UDR operator to be *associative* so that the implementation can compute the reduction using multiple threads without sequentializing (e.g., using a critical region).

OpenMP schedules allow different distributions of iterations to threads besides a single chunk of consecutive iterations. These schedules reorder the operations. The original OpenMP intrinsic reduction operators are all associative and *commutative*,[3] which allows the implementation to combine iterations into a partial reduction in each thread and then combine them into the global result in any order, such as the order in which the threads complete. While we could restrict the loop schedules or require the use of the `ordered` clause, we require the UDR operators to be commutative in order to maximize parallelism and to simplify the implementation of UDRs.

4.2 The `declare reduction` directive

UDRs must be declared prior to their use in a `reduction` clause. We use the `declare reduction` directive with the following syntax:[4]

```
1 #pragma omp declare reduction(operator-list:type-list) [clause]
```

where clause can only be an `identity` clause.

Basic Syntax The `declare reduction` directive instructs the compiler that the *operators* in the list are valid *UDR operators* for the types specified in the *type-list*. The directive can be specified for any type (primitive types and user-defined data types) except functions and array types. Reductions on function types do not have any useful semantics; arrays are handled differently (see Section 4.4).

A valid UDR operator *op* must exist for each type. As we strive to maximize code reuse we define a set of minimum requirements for a possible UDR operator to be valid instead of defining a fixed prototype to which all operators must conform. These requirements are the following:

- *op* must be a *binary* function with both arguments of a type compatible with the type in the UDR declaration;[5]
- *op* must be a commutative function;
- *op* must be an associative function;
- *op* must produce a result in its function return value or an argument; if *op* could produce multiple results (e. g., both arguments are pointer types) then the leftmost result is used (i. e., the precedence is return value, left argument, right argument).

For C++, all standard and (correctly implemented) overloaded operators are valid UDR operators; function members of that type are valid if they have a single argument compatible with the type. In any case, the function must be accessible in the scope where the reduction takes place as well as in the scope where the UDR is declared.

[3] Although the subtraction operator is non-commutative, it is mapped to the commutative addition operator by many OpenMP implementations.

[4] We only present the C/C++ syntax and requirements, which are similar to those of Fortran.

[5] With *T* being the type in the UDR declaration, compatible types in C or C++ include *T*, *const T*, *T **, *T &*, *const T ** and *const T &*.

```
1 struct T {
2     void alpha ( const T & );
3     const T & operator+ ( const T & );
4 };
5
6 T& alpha ( T &, T & );
7 T  beta  ( T *, T );
8 void  gamma ( T *, T );
9 void  delta ( T *, T *);
10 void  epsilon ( T, T *);
11
12 const T & operator* ( const T & , const T & );
13
14 #pragma omp declare reduction(+,*,T::alpha,alpha,beta,gamma,delta : T)
```

Fig. 4. C++ examples of valid UDR operators for data type *T*

```
1 #pragma omp declare reduction(fixed_mul:fixed_t) identity(1)
2 #pragma omp declare reduction(complex_mul:complex_t) identity({1.0,0.0})
3 #pragma omp declare reduction(*:Complex) identity(constructor(1.0,0.0))
```

Fig. 5. Examples of valid `identity` clauses

Fig. 3 provided valid UDR declarations for our simple C example used in Section 3. Fig. 4 provides additional examples of valid UDR declarations in C++ for a user-defined data type *T*. The operators + and * are overloaded C++ operators for *T* that are visible in the scope that contains the UDR declaration. The UDR operator *T::alpha* refers to the member function of *T*; *alpha* refers to the global function. The functions *gamma* and *delta* are valid UDR operators since they take two input values of type *T*. Both *gamma* and *delta* must store the reduction result in the left argument since they do not have return values. Similarly, *epsilon* must store it in the right argument.

The `identity` clause By default, we perform *zero initialization* for *non-object types* and invoke the default constructor for *object types* in C++. The `identity` clause overrides the default with user-defined values. It takes either a constant expression, a brace initializer, or the special keyword `constructor` and a list of constant expressions of the form *(expr1,...,exprN)*. In the first two cases, all temporaries are assigned the identity value initially. In the last case, the constructor for the specified type is invoked with the listed arguments. Fig. 5 shows examples of these different cases.

4.3 Extensions to the `reduction` clause

Our UDR proposal does not change the well-known syntax of the `reduction` clause. We only require that it accepts declared UDR operators as well as the built-in reduction operators (and intrinsic functions in Fortran). When a UDR operator is specified in a `reduction` clause, the OpenMP compiler must determine the UDR declaration that applies to the scope of that particular `reduction` clause. It then uses the information from the UDR declaration to implement the reduction. First, the compiler initializes

```
1 #pragma omp declare reduction(matrix_add:int[][])
2
3 void example() {
4    int M[n][n];
5
6 #pragma omp for reduction(matrix_add:M)
7    for(...) {...}
8 }
```

Fig. 6. Example of a UDR on a two-dimensional array

```
1 #pragma omp declare reduction(vector_add:int[])
2
3 void example(int *a, int n) {
4    int (*v)[n] = (int (*)[n]) a;
5
6 #pragma omp for reduction(vector_add:v)
7    for(...) {...}
8 }
```

Fig. 7. Example of UDR for arrays with pointer reshaping

any temporaries with the identity value. Second, it replaces occurrences of the original variable with the corresponding private temporary variable. Finally, it uses the UDR operator in its reduction algorithm to combine the temporaries into the overall result. As all necessary information is specified at the UDR declaration, the compiler can implement more sophisticated reduction approaches as well.

4.4 Array reductions

OpenMP 3.0 supports array reductions on primitive types for Fortran but not for C or C++ because the number of dimensions (and their size) may not be available to the compiler at the `reduction` clause in those languages.

If the reduction variable is strictly an array, the compiler could infer the number of dimensions and the size from the reduction variable. But, we require the programmer to add square brackets to the data type in the UDR *declaration* to specify that the operator will work for array types. This allows the compiler to check at the UDR *declaration* that the operator is valid for the type. The actual size of the dimensions, which are needed to create the correct private variables, is deduced by the compiler from the type of the variable of the `reduction` clause.

Fig. 6 declares a UDR on a two-dimensional matrix of `int` values by specifying `int[][]` as the UDR's data type. A compiler allocates private arrays of $n \times n$ elements (see line 4) and initializes each element with the identity value (i. e., in this case as no `identity` clause is specified, with the value 0) for the UDR usage in line 6.

While this works well for variables that are strictly arrays, an issue in C and C++ is that arrays are implicitly converted to pointers across call boundaries [8, 9]. To improve the support of UDRs for arrays, our proposal considers *pointers to arrays* as if they were arrays for the purpose of finding the corresponding UDR. As Fig. 7 shows,

```
#pragma omp declare reduction(template <typename T_> + : std::vector<T_>)
```

Fig. 8. UDR for adding `std::vector` objects

```
namespace A { class T; }
namespace B { class S; }

// declares UDRs A::T::foo, B::S::foo, A::T::bar, and B::S::bar
// with the same identity
#pragma omp declare reduction(.foo,.bar: A::T,B::S)

...

A::T t;
B::S s;

// Uses UDR A::T::foo for t and B::S::foo for s
#pragma omp parallel for reduction(.foo:t,s)
...
```

Fig. 9. Example of the dot syntax to shorten UDR declarations of qualified identifiers

programmers must often convert *pointer to types* to *pointers to arrays* that contain the needed dimensionality information. Although this solution requires some modifications to the sequential code, the sequential code remains valid and we avoid more extensive shaping expression support for the UDR.

4.5 C++-specific extensions

Although our UDR design provides a common syntax for C, C++, and Fortran, we extend the UDR syntax to make UDR declarations more concise and easier to use in C++. The first extension targets C++ templates (i. e., partially instantiated types):

```
#pragma omp declare reduction(template<template-header> op-list:type-list) [clause]
```

After specifying a *template-header*, the different template parameters defined in the header may appear in the *operator-list*, *type-list*, or the `identity` clause.

Template support is crucial for C++ to define UDRs for template types in a generic way. For instance, the template `std::vector<T_>` of Fig. 8 defines reductions on any possible vector. Otherwise, possible instantiations (e. g., `std::vector<int>`, `std::vector<float>`) would require separate UDR declarations.

In addition to template support, we use *dot syntax* to omit class qualifiers in both the `declare reduction` directive and the `reduction` clause. Fig. 9 shows how it simplifies the use of qualified C++ identifiers in UDRs. Qualifiers for identifiers can be omitted if a dot prefixes UDR operator names. The compiler then automatically qualifies the operator with the type(s) being declared (in UDR declaration directives) or of variables of `reduction` clauses. This syntax can greatly simplify the declaration and usage of UDR operators that have the same *name* on unrelated types but that implement the same kind of reduction.

5 Evaluation

Although our proposed extensions to OpenMP simplify reduction operations for non-basic types it remains to be seen if they can be implemented as efficiently as hand-made reductions.

To this purpose we implemented a prototype of our proposal for user-defined reductions in the Mercurium source-to-source compiler [2]. The code generated for standard OpenMP reductions and UDRs is the same except that it uses the identity and operator that the UDR declaration specifies.

We have implemented five kinds of reductions that capture typical OpenMP reduction scenarios:

- Our *standard reduction* tests the existing reduction support with an `int` sum;
- Our *manual critical* version stores partial values of each thread in a temporary variable, which it sums into a shared variable in a `critical` region at the end of the parallel region;
- Our *manual atomic* variant is identical to our *manual critical* version except that it performs the sum in an `atomic` construct, which generic UDRs cannot use—we evaluated this reduction strategy for completeness;
- Our *manual shared arrays* test uses the reduction algorithm that Fig. 2 shows;
- Our *UDR* version uses a UDR operator that adds two `int` variables and returns the result as the return value.

We use the statistical analysis of the EPCC OpenMP benchmark suite [1] but determine the overhead of reductions directly. Specifically, we profile the time that each thread spends in the reduction operation rather than indirectly measuring the overhead by subtracting the overhead of a `parallel` region from the overhead of a `parallel` region with a `reduction` clause. The indirect method has high variance since the reduction overhead is relatively small compared to that of any parallel region.

We measure reduction execution times with the Itanium interval timer facilities [3] of an SGI Altix 4700 (1.67 GHz). We vary the thread count from one to 64 and compute the average of 2,000 overhead measurements for each kind of reduction. The maximum deviation of the time measured is around 2%, which indicates that our methodology provides a consistent measurement. Fig. 10 summarizes our microbenchmark results.

The *manual critical* test incurs higher reduction overheads with increasing thread counts. As the thread count increases, more threads compete for the central lock that protects the computation of the global result. With two threads, the overhead to enter and to exit the `critical` region already exceeds the time spent within it. As the threads reach the `critical` region at about the same time, the overhead increases linearly with the thread count.

The *manual atomic* variant improves performance over the *manual critical* version by roughly 50% but we still observe linear increases in overhead with thread count. The *manual atomic* variant incurs smaller overhead since `atomic` uses atomic instructions instead of acquiring and releasing a lock. The machine's memory subsystem ensures mutual exclusion for accesses to the global result, which still incurs increasing overhead as the thread count increases. Each step of the reduction also incurs a cache fault.

The *standard reduction*, *manual shared array*, and *UDR* tests exhibit roughly the same overhead. Mercurium implements standard OpenMP reductions with private tem-

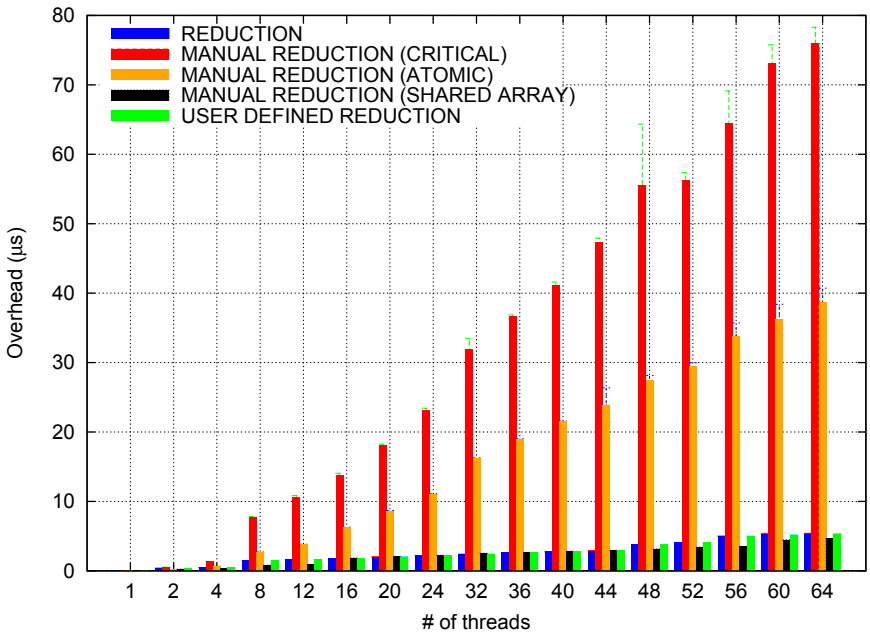

Fig. 10. Performance comparison of reduction patterns and UDRs

porary variables on each threads' stack. A `for` loop then retrieves each thread's private variable and adds them to the global result at the end of the `parallel` region. Our *manual shared arrays* implementation uses the same approach except that it stores the private variables in a shared array. Our UDR implementation extends Mercurium's standard reduction algorithm such that it invokes the UDR function when computing the global result. The native compiler inlines the UDR function so it incurs almost no additional overhead compared to the *standard reduction* and *manual shared array* tests.

In summary, our performance evaluation shows that the UDR implementation of our prototype exhibits the same level of performance as standard reductions and the efficient manual UDR implementation of Fig. 2. Thus, we demonstrate that OpenMP can remove the burden of error-prone and cumbersome manual idioms for reductions of user-defined types from the programmer while providing high performance through our UDR mechanism.

6 Conclusions and Future Work

OpenMP applications, like their sequential counterparts, often employ user-defined data types. Typically, programmers must overcome OpenMP's lack of support for reductions on these types. Our new mechanism overcomes this limitation by concisely specifying user-defined reductions in OpenMP programs. Our solution uses a declarative directive that is consistent with existing OpenMP syntax and allows existing binary functions on

user-defined data types to serve as UDRs. UDRs support all OpenMP base languages and blend well with potential future OpenMP base languages.

Our UDR mechanism allows the OpenMP runtime system to choose the most efficient reduction algorithm for a parallel region. For example, the runtime can adapt the reduction algorithm to the thread count, which would otherwise require complex user programming. Our measurements have shown that our proposal introduces no additional overhead compared to manually implemented reductions (or regular OpenMP reductions) while avoiding copy-and-paste duplication of reduction algorithms and hard-to-find errors that stem from user-level reduction implementations.

References

1. J.M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *Proc. of 1st European Workshop on OpenMP*, pages 99–105, Lund, Sweden, October 1999.
2. Barcelona Supercomputing Center. The NANOS Group Site: The Mercurium Compiler. `http://nanos.ac.upc.edu/mcxx`.
3. Intel Corporation. Intel Itanium 2 Processor Reference Manual for Software Development and Optimization, May 2004. Order number 251110-003.
4. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of ACM*, 51(1):107–113, January 2008.
5. S.J. Deitz, B.L. Chamberlain, and L. Snyder. High-level Language Support for User-defined Reductions. *Journal of Supercomputing*, 23(1), August 2002.
6. M. Frigo, P. Halpern, C.E. Leiserson, and S. Lewin-Berlin. Reducers and Other Cilk++ Hyperobjects. In *Proc. of the 21st Ann. Symp. on Parallelism in Algorithms and Architectures*, pages 79–90, Calgary, AB, Canada, August 2009.
7. IEEE. *Threads Extension for Portable Operating Systems (Draft 6)*, February 1992. Document P1003.4a/D6.
8. ISO/IEC. Programming Languages – C, 1999. ISO/IEC 9899:1999.
9. ISO/IEC. Programming Languages – C++, 2003. ISO/IEC 14882:2003.
10. M. Klemm, R. Veldema, M. Bezold, and M. Philippsen. A Proposal for OpenMP for Java. In M. S. Mueller, B.M. Chapman, B.R. de Supinski, A.D. Malony, and M. Voss, editors, *OpenMP Shared Memory Parallel Programming (International Workshops IWOMP 2005 and IWOMP 2006)*, pages 409–421, Berlin, Germany, 2008. Springer.
11. D. McGrady. Avoiding Contention using Combinable Objects. online, September 2008. `http://blogs.msdn.com/nativeconcurrency/archive/2008/09/25/avoiding-contention-using-combinable-objects.aspx`.
12. M. Michaelis. *Essential C# 3.0: For .NET Framework 3.5 (Microsoft .Net Development)*. Addison-Wesley Longman, Amsterdam, The Netherlands, 2nd edition, September 2008.
13. Microsoft Developer Network. Process and Thread Functions (Windows). Available at `http://msdn.microsoft.com/en-us/library/ms684847\%28VS.85\%29.aspx`.
14. MPI Forum. MPI: Extensions to the Message-passing Interface, Version 2.2. Technical report, MPI Forum, September 2009.
15. S. Oaks and H. Wong. *Java Threads*. O'Reilly, Sebastopol, CA, USA, 3rd edition, 2004.
16. OpenMP ARB. OpenMP Application Program Interface, v. 3.0, May 2008.
17. J. Reinders. *Intel Threading Building Blocks*. O'Reilly, Sebastopol, CA, USA, July 2007.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.