Language Extensions in Support of Compiler Parallelization

Jun Shirako[†,††], Hironori Kasahara[†,‡], and Vivek Sarkar[‡‡]

[†]Dept. of Computer Science, Waseda University
[††]Japan Society for the Promotion of Science, Research Fellow
[‡]Advanced Chip Multiprocessor Research Institute, Waseda University
[‡‡]Department of Computer Science, Rice University
{shirako,kasahara}@oscar.elec.waseda.ac.jp, vsarkar@rice.edu

Abstract. In this paper, we propose an approach to automatic compiler parallelization based on language extensions that is applicable to a broader range of program structures and application domains than in past work. As a complement to ongoing work on high productivity languages for explicit parallelism, the basic idea in this paper is to make sequential languages more amenable to compiler parallelization by adding enforceable declarations and annotations. Specifically, we propose the addition of annotations and declarations related to multidimensional arrays, points, regions, array views, parameter intents, array and object privatization, pure methods, absence of exceptions, and gather/reduce computations. In many cases, these extensions are also motivated by best practices in software engineering, and can also contribute to performance improvements in sequential code. A detailed case study of the Java Grande Forum benchmark suite illustrates the obstacles to compiler parallelization in current object-oriented languages, and shows that the extensions proposed in this paper can be effective in enabling compiler parallelization. The results in this paper motivate future work on building an automatically parallelizing compiler for the language extensions proposed in this paper.

1 Introduction

It is now well established that parallel computing is moving into the mainstream with a rapid increase in the adoption of multicore processors. Unlike previous generations of mainstream hardware evolution, this shift will have a major impact on existing and future software. A highly desirable solution to the multicore software productivity problem is to automatically parallelize sequential programs. Past work on automatic parallelization has focused on Fortran and C programs with a large body of work on data dependence tests [1, 25, 18, 9] and research compilers such as Polaris [7, 19], SUIF [10], PTRAN [21] and the D System [12]. However, it is widely acknowledged that these techniques have limited effectiveness for programs written in modern object-oriented languages such as Java.

In this paper, we propose an approach to compiler parallelization based on language extensions that is applicable to a broader range of program structures and application domains than in past work. As a complement to ongoing work on high productivity languages for explicit parallelism, the basic idea in this paper is to make sequential languages more amenable to compiler parallelization by adding enforceable declarations and annotations. In many cases, these extensions are also motivated by best practices in software engineering, and can also contribute to performance improvements in sequential code.

A detailed case study of the Java Grande Forum benchmarks [22, 13] confirms that the extensions proposed in this paper can be effective in enabling compiler parallelization. Experimental results were obtained on a 16-way Power6 SMP to compare the performance of four versions of each benchmark: 1) sequential Java, 2) sequential X10, 3) hand-parallelized X10, 4) parallel Java. Averaged over ten JGF Section 2 and 3 benchmarks, the parallel X10 version was 11.9× faster than the sequential X10 version, which in turn was 1.2× faster than the sequential Java version (Figure 1). An important side benefit of the annotations used for parallelization is that they can also speed up code due to elimination of runtime checks. For the eight benchmarks for which parallel Java versions were available, the parallel Java version was an average of 1.3× faster than the parallel X10 version (Figure 2). However, for two of the eight benchmarks, the parallel Java version used a different algorithm from the sequential Java version, and resulted in super-linear speedups. When the sequential and parallel X10 versions for the two benchmarks were modified to be consistent with the new algorithms, the parallel Java and X10 versions delivered the same performance on average (Figure 3).

The rest of the paper is organized as follows. Section 2 describes the language extensions (annotations and declarations) proposed in this paper. Section 3 summarizes the results of the case study including experimental results, and Section 4 contains our conclusions.

2 Language Extensions

While modern object-oriented languages such as Java have improved programming productivity and code reuse through extensive use of object encapsulation and exceptions, these same features have made it more challenging for automatically parallelizing compilers relative to Fortran programs where data structures and control flow are more statically predictable. In this section, we propose a set of declarations and annotations that enable compilers to perform automatic parallelization more effectively for these languages. Unlike annotations that explicitly manage parallelism as in OpenMP [6], our approach is geared toward enforceable declarations and annotations that can be expressed and understood in the context of sequential programs, and that should be useful from a software engineering viewpoint because of their ability to reduce common programming errors. Another difference from OpenMP is that the correctness of all our pro-

posed annotations and declarations is enforced by the language system *i.e.,* they are all checked statically or dynamically, as outlined below.

2.1 Multidimensional Arrays, Regions, Points

Multidimensional arrays in Java are defined and implemented as nested unidimensional arrays. While this provides many conveniences for guaranteeing safety in a virtual machine environment (e.g., subarrays can be passed as parameters without exposing any unsafe pointer arithmetic), it also creates several obstacles to compiler optimization and parallelization. For example, a compiler cannot automatically conclude that `A[i][j]` and `A[i+1][j]` refer to distinct locations since the nested array model allows for the possibility that `A[i]` and `A[i+1]` point to the same subarray. Instead, we propose the use of object-oriented multidimensional arrays as in X10 [3], in which a compiler is guaranteed that `A[i,j]` and `A[i+1,j]` refer to distinct locations. *Array Views* (Section 2.2) make it possible to safely work with subarrays of multidimensional arrays without introducing unsafe pointer arithmetic.

A related issue is that induction variable analysis can be challenging in cases when an iterator is used or an integer variable is incremented by a step value that is not a compile-time constant as illustrated in the following common idiom from a DAXPY-like computation:

```
iy = 0; if (incy < 0) iy = (-n+1)*incy;
for (i = 0;i < n; i++) {
  dy[iy +dy_off] += . . .;  iy += incy;
}
```

In the above example, it is not easy for compilers to establish that `incy` $\neq 0$ and that there are no loop-carried dependences on the `dy` array.

To simplify analysis in such cases, we recommend the use of *regions* and *points* as proposed in ZPL [23] and X10, with extensions to support two kinds of region constructors based on triple notation, `[<start-expr> : <end-expr> : <step-expr>]` and `[<start-expr> ; <count-expr> ; <step-expr>]`, both of which are defined to throw a ZeroStepException if invoked with a zero-valued step expression. The use of high level regions and points distinguishes our approach from past work on annotations of arrays for safe parallelization [16].

A key property of regions and points is that they can be used to define both loops and arrays in a program. The above DAXPY-like example can then be rewritten as follows:

```
iy = 0; if (incy < 0) iy = (-n+1)*incy;
// Example of [<start-expr>;<count-expr>;<step-expr>] region
for (point p : [iy ; n ; incy] ) {
  dy[p] += . . .;
}
```

In this case, the compiler will know that `incy` $\neq 0$ when the loop is executed, and that all `dy[p]` accesses are distinct.

2.2 Array Views

As indicated in the previous section, it is easier for a compiler to parallelize code written with multidimensional arrays rather than nested arrays. However, this raises the need for the programmer to work with subarrays of multidimensional arrays without resorting to unsafe pointer arithmetic. Our solution is the use of *array views*. An array view can be created by invoking a standard library method, `view( <start-point-expr>, <region-expr>)`, on any array expression (which itself may be a view). Consider the following code fragment with array views:

```
// Allocate a two-dimensional M*N array
double[.] A = new double[[1:M,1:N]];
. . .
A[i,j] = 99;
. . .
// Allocate a one-dimensional view on A for row i
double[.] R = A.view([i,1], [1:N]);
. . .
temp = R[j]; // R[j] = 99, the value stored in A[i,j]
```

In the above example, `R` can be used like any one-dimensional array but accesses to `R` are aliased with accesses to `A` as specified by the region in the call to `A.view()`. A `ViewOutOfBoundsException` is thrown if a view cannot be created with the specified point and region. All accesses to `R` can only be performed with points (subscripts) that belong to the region specified when creating the view.

Views can also be created with an optional *intent* parameter that must have a value from a standard *enum*, {`In, Out, InOut`}. The default value is `InOut` which indicates that the view can be used to read and write array elements. `In` and `Out` intents are used to specify read-only and write-only constraints on the array views. Read-only views can be very helpful in simplifying compiler parallelization and optimization by identifying heap locations that are guaranteed to be immutable for some subset of the program's lifetime [17]. The runtime system guarantees that each array element has the same intent in all views containing the element. (We also propose a `System.gcArrayViews()` call that enables the application to force collection of all array views at desired program points.) If an attempt is made to create a view that conflicts with the intent specified by a previous view, then a `ViewIntentException` is thrown.

2.3 Annotations on method parameters

We propose the use of a `disjoint` annotation to assert that all mutable (non-value) reference parameters in a method must be disjoint. (The *this* pointer is also treated as a parameter in the definition of the `disjoint` annotation.) If a disjoint method is called with two actual parameters that overlap, a `ParameterOverlapException` is thrown at runtime. Declaring a method as disjoint can help optimization and parallelization of code within the method by assisting the compiler's alias analysis. This benefit comes at the cost of runtime tests that the

compiler must insert on method entry, though the cost will be less in a strongly typed language like Java or X10 compared to a weakly typed language like C since runtime tests are not needed for parameters with non-compatible types in X10 but would be necessary in C due to its pointer addressing and cast operators. This is also why we expect it to be more effective for X10 than the `noalias` and `restricted` proposals that have been made in the past for `C`.

In addition to the `disjoint` annotation, we also propose the use of `in`, `out`, and `inout` intent annotations on method parameters as in Fortran. For object/array references, these annotations apply only to the object/array that is the immediate target of the reference.

2.4 Array and Object Privatization

It is well known that *privatization analysis* is a key enabling technique for compiler parallelization. For modern object-oriented languages with dynamically allocated objects and arrays, the effectiveness of privatization analysis is often bounded by the effectiveness of *escape analysis* [4]. We propose a `retained` type modifier[1] for declarations of local variables and parameters with reference types which asserts that the scope in which the local/parameter is declared will not cause any reference in a retained variable to escape. We also permit the `retained` modifier on declarations of methods with a non-value reference return type, in which case it ensures that the `this` pointer does not escape the method invocation.

The following loop from the MonteCarlo benchmark illustrates the use of the retained modifier to declare that each `ps` object is private to a single loop iteration.

```
results = new Vector(nRunsMC);
for( int iRun=0; iRun < nRunsMC; iRun++ ) {
    // ps object is local to a single loop iteration
    retained PriceStock ps = new PriceStock();
    // All methods invoked on ps must be declared as "retained"
    ps.setInitAllTasks((ToInitAllTasks) initAllTasks);
    ps.setTask((x10.lang.Object) tasks.elementAt(iRun));
    ps.run();
    results.addElement(ps.getResult());
} // for
```

To enable automatic parallelization, the compiler will also need information that indicates that results.addElement() is a reduction-style operator (associative and commutative). We discuss later in Section 2.7 how this information can be communicated using a `finally` clause for *gather* variables.

[1] The `retained` name chosen because other candidates like “private” and “local” are overloaded with other meanings in Java.

2.5 Pure annotation for side-effect-free methods

The return value (or exception value) and all parameters of a method annotated as `pure` must have *value types i.e.,* they must be immutable after initialization. Pure methods can call other pure methods and only allocate/read/write mutable heap locations whose lifetimes are contained within the method's lifetime (as defined with the `retained` type modifier). Therefore, if two calls are made to the same pure method with the same value parameters, they are guaranteed to result in the same return value (or exception value). The only situation in which the two calls may not have the same outcome is if one of the calls triggers a nonfunctional error such as `OutOfMemoryError`. This definition of method purity is similar to the definition of "moderately pure" methods in [26]. The correctness of all `pure` annotations is enforced statically in our proposed approach, analogous to the static enforcement of immutability of value types in the X10 language [3].

2.6 Annotations related to Exceptions

We propose the following set of declarations and annotations that can be used to establish the absence of runtime exceptions. All type declarations are assumed to be checked statically, but dynamic cast operations can be used to support type conversion with runtime checks. Some of the type declarations are based on the theory of *dependent types* (*e.g.,* see [11]) as embodied in version 1.01 of the X10 language [20].

- **Null Pointer exceptions:** A simple way to guarantee the absence of a NullPointerException for a specific operation is to declare the type of the underlying object/array reference to be *non-null*. As an example, the Java language permits null-valued references by default, with a proposal in JSR 305 [15] to introduce an @NonNull annotation to declare selected references as non-null. In contrast, the X10 language requires that all references be non-null by default and provides a special *nullable* type constructor that can be applied to any reference type. Though the results in our paper can be used with either default convention, we will use the X10 approach in all examples in this paper.
- **Array Index Out of Bounds exceptions:** A simple way to guarantee the absence of an `IndexOutOfBoundsExecption` for an array access is to ensure that the array access is performed in a loop that is defined to iterate over the array's region *e.g.,*

  ```
  for (point p : A.region) A[p] = ... ; //Iterate over A.region
  ```

  This idea can be extended by iterating over a region that is guaranteed to be a subset of the array's region, as in the following example (assuming `&&` represents region intersection):

  ```
  // Iterate over a subset of A.region
  for (point p : A.region && region2) A[p] = ... ;
  ```

When working with multiple arrays, dependent types can be use to establish that multiple arrays have the same underlying region *e.g.*,

```
final region R1 = ...;
// A and B can only point to arrays with region = R1
final double[:region=R1] A = ...;
final double[:region=R1] B = ...;
for (point p : R1 ) A[p] = F(B[p]) ; // F is a pure method
```

In the above example, the compiler knows from the dependent type declarations (and from the fact that the loop iterates over region `R1`) that array accesses `A[p]` and `B[p]` cannot throw an exception.
Dependent types can also be used on point declarations to ensure the absence of `IndexOutOfBoundsException`'s as in the access to `A[p]` in the following example:

```
final region R1 = ...;
final double[:region=R1] A = ...;
// p can only take values in region R1
point(:region=R1) p = ...;
double d = A[p];
```

– **Zero Divide/Step exceptions:** A simple way to guarantee the absence of a DivideByZeroException or a ZeroStepException for a specific operation is to declare the type of the underlying integer expression to be *nonzero* using dependent types as follows:

```
int(:nonzero) n = ...; // n's value must be nonzero
int q = m / n; // No DivideByZeroException
region R3 = [low : high : n]; // No ZeroStepException
```

– **ExceptionFree annotation:** A code region annotated as `ExceptionFree` is guaranteed to not throw any user-defined or runtime exception. As with `pure` methods, it is possible that a region of code annotated as `ExceptionFree` may encounter a nonfunctional error such as an `OutOfMemoryError`. The compiler checks all operations in the annotated code region to ensure that they are statically guaranteed to not throw an exception (by using the declarations and annotations outlined above).

2.7 Gather Computations and Reductions in Loops

A common requirement in parallel programs is the ability to either *gather* or *reduce* values generated in each loop iteration into (respectively) a collection or aggregate value. There has been a large body of past work on compiler analyses for automatically detecting gather and reduction idioms in loops and arrays *e.g.*, [14, 8], but the presence of potentially aliased objects and large numbers of virtual calls render these techniques ineffective for object-oriented programs. Instead, we propose an extension to counted pointwise `for` loops that enables the

programmer to specify the *gather* and *reduce* operators explicitly in a sequential program in a way that simplifies the compiler's task of automatic parallelization. Specifically, we extend the `for` loop with an optional `finally` clause as follows:

```
for ( ... ) <body-stmt> finally <gather-stmt>
```

A unique capability of the gather statement is that it is permitted to read private (`retained`) variables in the loop body that have primitive or value types, including the index/point variables that control the execution of the counted `for` loop. The design of `finally` clause is similar to the `inlet` feature in Cilk [24], which represents a post-processing of each parallel thread. Informally, the semantics of a `for` loop with a `finally` clause can be summarized as follows:

1. Identify `retained` variables in `<body-stmt>` with primitive or value types that are also accessed in `<gather-stmt>`. We refer to these as *gather* variables.
2. Execute all iterations of the `for` loop sequentially as usual, but store the values of *all* gather variables at the end of each iteration.
3. Execute `<gather-stmt>` once for each iteration of the `for` loop in a nondeterministic order (analogous to the nondeterminism inherent in iterating over an unordered collection in Java).
4. During execution of an instance of `<gather-stmt>` in Step 3, resolve read accesses to gather variables by returning the corresponding instances stored in Step 2.

We use the loop from the MonteCarlo benchmark discussed earlier to illustrate the use of the finally clause to specify the gather statement:

```
results = new Vector(nRunsMC);
for( point p[iRun] : [0 : nRunsMC-1] ; iRun++ ) {
    // ps object is local to a single loop iteration
    retained PriceStock ps = new PriceStock();
    // All methods invoked on ps must be declared as "retained"
    ps.setInitAllTasks((ToInitAllTasks) initAllTasks);
    ps.setTask((x10.lang.Object) tasks.elementAt(iRun));
    ps.run();
    retained ToResult R = ps.getResult(); // must be a value type
}
finally {
    // Invoked once for each iteration of the for loop
    results.addElement(R));
}
```

`retained` and `finally` would support automatic privatization and reduction by compilers [7] in complicated languages like Java. Other annotations, such as `nonzero` and `pure` can also help symbolic analysis and data dependence analysis.

3 Case Study: Java Grande Forum Benchmarks

In this section, we present the results of a case study that we undertook to validate the utility of the language annotations and extensions introduced in the previous section for compiler parallelization. The case study was undertaken on the Java Grande Forum (JGF) benchmark suite [22] because this suite includes both sequential and parallel (multithreaded) Java versions of the same benchmarks. We converted the sequential Java versions into sequential X10 versions, and then studied which annotations were necessary to enable automatic parallelization. A summary of our results can be found in Table 1.

	Series	Sparse*	SOR	Crypt	LUFact	FFT	Euler	MolDyn	Ray*	Monte*
Multi-dim arrays	×		×		×		×			
Regions, Points	×	×	×		×		×	×		
Array views		×			×					
In/Out/InOut					×					
Disjoint		×		×			×			
Retained							×		×	×
Pure method	×					×				
NonNull	×	×	×	×	×	×	×	×	×	×
Region Dep-type		×		×	×	×			×	
Nonzero						×				
Exception free	×				×	×		×	×	×
Reduction		×						×	×	×

* Sparse: SparseMatmult, Ray: RayTracer, Monte: MonteCarlo

Table 1. Annotations required to enable parallelization of Java Grande Forum benchmarks

We then compared the performance of four versions of the Java Grande Forum (JGF) benchmarks:

1. **Sequential Java:** This set consists of six Section 2 benchmarks (Crypt, FFT, LUFact, Series, SOR, SparseMatmult) and four Section 3 benchmarks (Euler, MolDyn, MonteCarlo, RayTracer) taken from version v2.0 of the JGF benchmark release [13][2].
2. **Sequential X10:** Since the sequential subset of X10 overlaps significantly with the sequential subset of Java, this version is quite close to the Sequential Java version in most cases. As in [2] we use a "lightweight" X10 version with regular Java arrays to avoid the large overheads incurred on X10 arrays in the current X10 implementation. However, all the other characteristics of X10 (*e.g.,* non-null used as the default type declaration, forbidden use of

[2] Section 1 was excluded because it only contains microbenchmarks for low-level operations.

non-final static fields, etc.) are preserved faithfully in the Sequential X10 versions.

3. **Hand Parallelized X10:** This version emulates by hand the parallel versions that can be obtained by a compiler, assuming that annotations are added to the sequential X10 versions as outlined in Table 1.
4. **Parallel Java:** This is the threadv1.0 version of the JGF benchmarks [13], which contains multithreaded versions of five of the six Section 2 benchmarks and three of the four Section 3 benchmarks. The unimplemented benchmarks are FFT and Euler. Further, the threaded versions of two of the Section 2 benchmarks, SOR and SparseMatmult, were implemented using a different underlying algorithm from the sequential versions in v2.0.

All performance results were obtained using the following system settings:

- The target system is a p570 16-way Power6 4.7GHz SMP server with 186GB main memory running AIX5.3 J. In addition, each dual-core chip can access 32MB L3 cache per chip and 4MB L2 cache per core. The size of the L1 instruction cache is 64KB and data cache is 64KB.
- For all runs, SMT was turned off and a large page size of 16GB was used. The sequential Java and X10 versions used only 1 processor, where as the parallel Java and X10 versions used all 16 processors.
- The execution environment used for all Java runs is IBM's J9 VM (build 2.4, J2RE 1.6.0) with the following options, `-Xjit:count=0,optLevel=veryHot, ignoreIEEE -Xms1000M -Xmx1000M`.
- The execution environment used for all X10 runs was version 1.0.0. of the X10 compiler and runtime, combined with the same JVM as above, IBM's J9 VM (build 2.4, J2RE 1.6.0), but with additional options to skip null pointer and array bounds checks in X10 programs in accordance with the annotation in the X10 source program. The `INIT_THREADS_PER_PLACE` parameter was set to 1 and 16 for the sequential and parallel X10 runs respectively. (`MAX_NUMBER_OF_PLACES` was set to 1 in both cases.)
- The X10 runtime was also augmented with a special *one-way* synchronization mechanism to enable fine-grained producer-consumer implementations of X10's `finish` and `next` operations.
- For all runs, the main program was extended with a three-iteration loop within the same Java process, and the best of the three times was reported in each case. This configuration was deliberately chosen to reduce/eliminate the impact of JIT compilation time in the performance comparisons.

3.1 Sequential and Parallel versions of X10

Figure 1 shows the speedup ratio of the serial and parallel X10 versions relative to the sequential Java version (JGF v2.0) for all ten benchmarks. An interesting observation is that the sequential X10 version often runs faster than the sequential Java version. This is due to the annotations in the X10 program which enabled null checks and array bounds checks to be skipped. On average, the

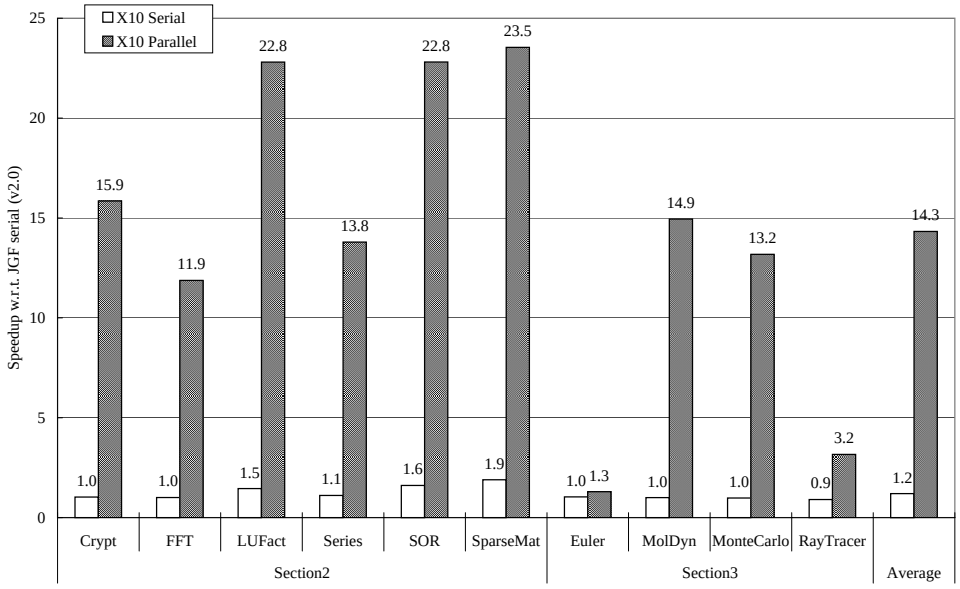

Fig. 1. Performance of Sequential and Parallel versions of X10 relative to Sequential Java

sequential X10 version was 1.2× faster than the sequential Java version, and the parallel X10 version was 11.9× faster than the sequential X10 version.

The sources of large speedups for SOR and LUFact were as follows. SOR's pipeline parallelism (faithful to the sequential version) was implemented using tightly-coupled one-way synchronizations which were added to the X10 runtime. The annotations for LUFact enabled a SPMD parallelization by following classical SPMDization techniques such as the approach outlined in [5].

The speedup was lowest for two benchmarks, *Euler* and *Raytracer*. The challenge in *Euler* is that it consists of a large number of small parallel loops which could probably benefit from more aggressive loop fusion and SPMD parallelization transformations than what was considered in our hand-parallelized experiments. The challenge in *Raytracer* is the classic trade-off between load balance (which prefers cyclic-style execution of the parallel loop) and locality (which prefers a block-style execution of the parallel loop).

3.2 Comparison with Parallel Java versions

In this section, we extend results from the previous section by including results for Parallel Java executions as well. As mentioned earlier, Parallel Java versions (threadv1.0) are available for 8 of the 10 benchmarks. Results for these 8 benchmarks are shown in Figure 2. The two benchmarks for which the Parallel Java versions significantly out-performed the Parallel X10 versions were SOR

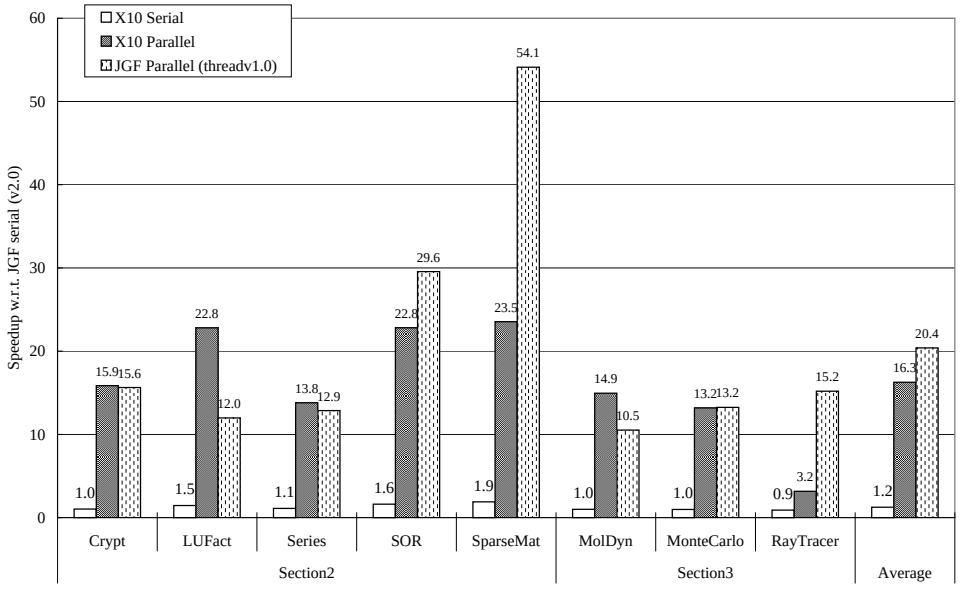

Fig. 2. Performance of Sequential and Parallel versions of X10 and Parallel Java relative to Sequential Java

and SparseMatmult. On closer inspection, we discovered that the underlying sequential algorithm was modified in both parallel versions (relative to the v2.0 sequential Java versions).

For SOR, the threadv1.0 parallel Java version uses a "red-black" scheduling of loop iteration to expose doall parallelism, even though this transformation results in different outputs compared to the sequential Java version. In contrast, the parallel X10 version contains pipeline parallelism that we expect can be automatically extracted from the sequential X10 version, and in fact returns the same output as the sequential X10 version.

For SparseMatmult, the thread v1.0 parallel Java version inserts an algorithmic step to sort non zero elements by their row value, so that the kernel computation can be executed as simple doall loop. Unfortunately, this additional step isn't included in the execution time measurement for the Parallel Java case.

To take into account the algorithmic changes in the Parallel Java versions, Figure 3 show an alternate version of Figure 2 in which the algorithms used for the sequential and parallel X10 versions are modified to match the algorithm used in the parallel Java versions. With the algorithmic changes, we see that the performance of the parallel Java and X10 versions are now evenly matched in the average case.

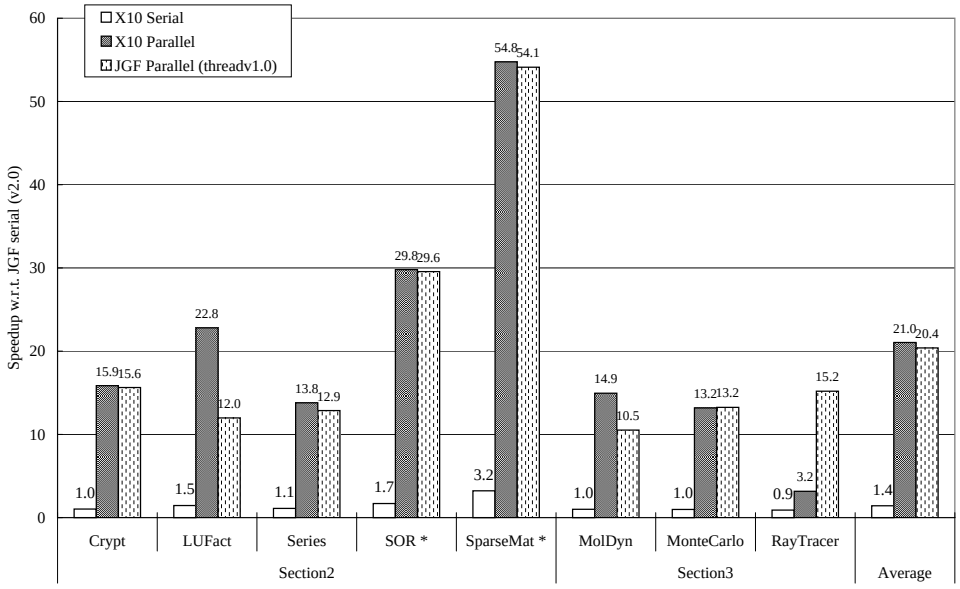

Fig. 3. Performance of Sequential and Parallel versions of X10 and Parallel Java relative to Sequential Java, with alternate Parallel X10 versions for SOR and SparseMatmult

4 Conclusions and Future Work

In this paper, we proposed a set of language extensions (enforced annotations and declarations) designed with a view to making modern object oriented languages more amenable to compiler parallelization. Many of the proposed extensions are motivated by best practices in software engineering for sequential programs. This is in contrast to the OpenMP approach where the annotations are geared towards explicit parallel programming and the correctness of user pragmas is not enforced by the language system.

We also performed a detailed case study of the Java Grande Forum benchmarks to confirm that the extensions proposed in this paper are effective in enabling compiler parallelization. Experimental results obtained on a 16-way Power6 SMP showed that the use of these language extensions can improve sequential execution time by 20% on average, and that a hand-simulation of automatically parallelized X10 programs can deliver speedup by matching the performance of the multithreaded Java versions of the JGF benchmarks.

The main topic for future work is to build an automatically parallelizing compiler which exploits the language extensions proposed in this paper. Another topic is to extend the definition of the language extensions to apply to explicitly parallel code *e.g.,* defining array views in the presence of distributions, and defining the semantics of in/out/inout intents for array views in the presence of concurrent array operations.

Acknowledgments

We are grateful to all X10 team members for their contributions to the X10 software used in this paper. We would like to especially acknowledge Vijay Saraswat's work on the design and implementation of dependent types in the current X10 implementation, and Chris Donawa and Allan Kielstra's implementation of experimental options for X10 in IBM's J9 virtual machine. While at IBM, Vivek Sarkar's work on X10 was supported in part by the Defense Advanced Research Projects Agency (DARPA) under its Agreement No. HR0011-07-9-0002. Finally we would like to thank Doug Lea, John Mellor-Crummey and Igor Peshansky for their feedback on this paper.

References

1. Randy Allen and Ken Kennedy. *Optimizaing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001.
2. Rajkishore Barik, Vincent Cave, Christopher Donawa, Allan Kielstra, Igor Peshansky, and Vivek Sarkar. Experiences with an smp implementation for x10 based on the java concurrency utilities. In *Workshop on Programming Models for Ubiquitous Parallelism (PMUP), held in conjunction with PACT 2006*, September 2006.
3. Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM Press.
4. Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.
5. Ron Cytron, Jim Lipkis, and Edith Schonberg. A compiler-assisted approach to spmd execution. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 398–406, Washington, DC, USA, 1990. IEEE Computer Society.
6. L. Dagum and R. Menon. OpenMP: An industry standard API for shared memory programming. *IEEE Computational Science & Engineering*, 1998.
7. R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. on parallel and distributed systems*, 9(1), Jan. 1998.
8. Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, 1995.
9. M. R. Haghighat and C. D. Polychronopoulos. *Symbolic analysis for parallelizing compilers*. Kluwer Academic Publishers, 1995.
10. M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 1996.
11. Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 341–354, New York, NY, USA, 1990. ACM Press.

12. S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the fortran d compiler. *Proc. of Supercomputing '93*, 1993.
13. The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande.
14. Pierre Jouvelot and Babak Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 186–194, New York, NY, USA, 1989. ACM Press.
15. Jsr 305: Annotations for software defect detection. http://jcp.org/en/jsr/detail?id=305.
16. Jose E. Moreira, Samuel P. Midkiff, and Manish Gupta. Supporting multidimensional arrays in java. *Concurrency and Computation Practice & Experience (CCPE)*, 15(3:5):317–340, 2003.
17. Igor Pechtchanski and Vivek Sarkar. Immutability Specification and its Applications. *Concurrency and Computation Practice & Experience (CCPE)*, 17(5:6), April 2005.
18. W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proc. of Super Computing '91*, 1991.
19. L. Rauchwerger, N. M. Amato, and D. A. Padua. Run-time methods for parallelizing partially parallel loops. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, pages 137–146, Jul. 1995.
20. Vijay Saraswat. Report on the experimental language x10 version 1.01. http://x10.sourceforge.net/docs/x10-101.pdf.
21. Vivek Sarkar. The PTRAN Parallel Programming System. In B. Szymanski, editor, *Parallel Functional Programming Languages and Compilers*, ACM Press Frontier Series, pages 309–391. ACM Press, New York, 1991.
22. L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 8–8, New York, NY, USA, 2001. ACM Press.
23. Lawrence Snyder. The design and development of zpl. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 8–1–8–37, New York, NY, USA, 2007. ACM Press.
24. MIT laboratory for computer science Supercomputing technologies group. Cilk 5.3.2 reference manual. http://supertech.csail.mit.edu/cilk/manual-5.3.2.pdf.
25. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.
26. Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for java programs. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 75–82, New York, NY, USA, 2007. ACM Press.