1998

Development and Utilization of Parallel Generic Algorithms for Scientific Computations

Atanas Radenski
Chapman University, radenski@chapman.edu

Andrew Vann

Boyana Norris

Recommended Citation

Radenski, A., A. Vann, B. Norris. Development and Utilization of Parallel Generic Algorithms for Scientific Computations (preliminary report). In S. Demeyer, J. Bosch (Eds), *Object-Oriented Technology: ECOOP'98 Workshop Reader*, Springer, 1998, 464-465. doi: 10.1007/3-540-49255-0_151
Later published as:
Radenski, A., A. Vann, B. Norris. Development and Utilization of Generic Algorithms for Scientific Cluster Computations. In Henderson M., C. Anderson, and S. Lyons (Eds), *Object Oriented Methods for Interoperable Scientific and Engineering Computing*, SIAM, 1999, 97-105.

Development and Utilization of Parallel Generic Algorithms for Scientific Computations[1]

Atanas Radenski[2] Andrew Vann[3] Boyana Norris[4]

Abstract

We develop generic parallel algorithms as extensible modules that encapsulate related classes and parallel methods. Extensible modules define common parallel structures, such as meshes, pipelines, or master-server networks in problem-independent manner. Such modules can be extended with sequential domain-specific code in order to derive particular parallel applications. In this paper, we first outline the essence of extensible modules. Then, we focus on a case study of the cellular automaton, a message-parallel generic algorithm from which we derive diverse parallel scientific applications.

1 Introduction

In this paper, we introduce an object-oriented enhancement to modular languages called *module extension* that facilitates the development and utilization of generic parallel algorithms. Module extension is a form of inheritance that applies to modules and that utilizes class overriding in addition to method overriding.

We have implemented module extension in a object-parallel language called *Paradigm/SP*. We use Paradigm/SP to develop generic parallel algorithms as modules that encapsulate related classes and parallel methods. Particular parallel applications are derived by extending such generic modules with sequential code. We use the Paradigm/SP implementation to test the validity of the derived parallel algorithms before finally converting them into efficient C code that runs in a cluster-computing environment, such as PVM.

Our technical goal is to develop an object-oriented programming methodology for generic scientific computations on clusters of workstations. In the paper, we focus on a case study of a *generic cellular automaton*, a message-parallel algorithm that can be applied to a variety of scientific problems. In section 2, we present the general properties of extensible modules, then define the cellular automaton and discuss how the definition can be shaped as an extensible module.

We use the generic cellular automaton to derive a hierarchy of diverse parallel algorithms with decreasing generality (Fig. 1). The derivation is achieved through module extension. In the first part of section 3, we derive *successive over-relaxation*, an iterative method that can be used to find numeric

[1] This work is partially supported by NSF grant CCR-9509223 and NASA grant NAG3-2011
[2] Computer Science Department, Winston-Salem State University, Winston-Salem, NC
[3] Computer Science Department, Clemson University, Clemson, SC
[4] Computer Science Department, University of Illinois at Urbana-Champaign, IL

solutions of partial differential equations. From this generic algorithm, we derive a parallel algorithm for a specific problem - *Laplace equation* for stationary heat flow. In the second part of Section 3, we begin again with the generic cellular automaton and derive a *parallel genetic algorithm*. The parallel genetic algorithm is then extended to produce a parallel algorithm for approximate *job scheduling*.

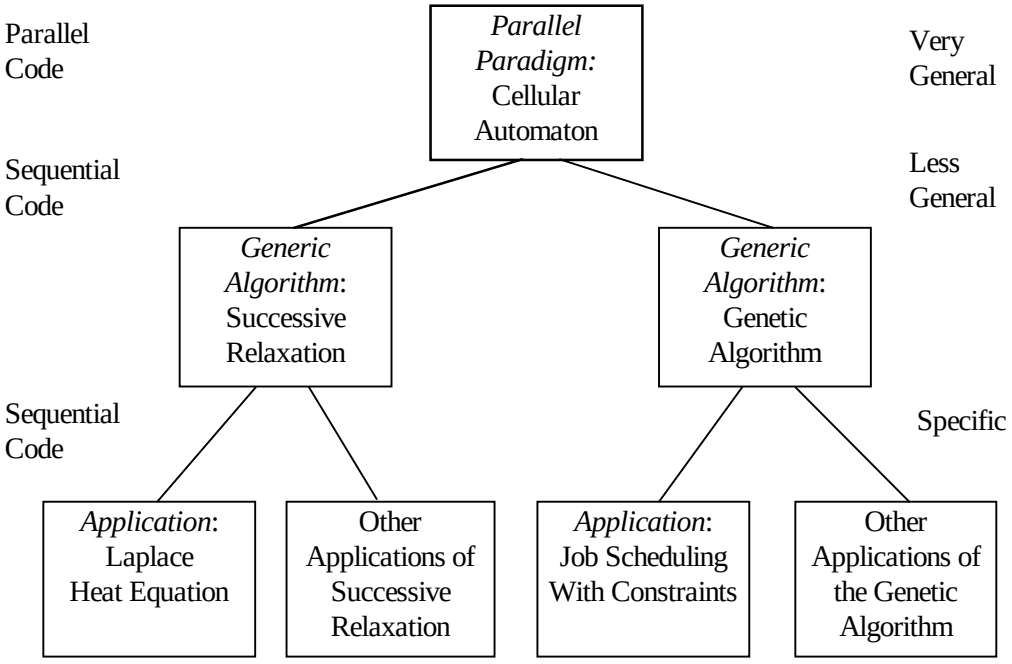

Figure 1. *A hierarchy of diverse parallel algorithms can be derived from the generic cellular automaton.*

Finally, we convert derived parallel algorithms into efficient C code that runs in a cluster computing environment, such as PVM. Section 3 presents performance results obtained on an Ethernet cluster on Sun Ultra-1 workstations. Section 4 is devoted to related work and conclusions.

2 Specification Of A Generic Parallel Algorithm As An Extensible Module

2.1 The Essence of Extensible Modules

An *extensible module* encapsulates (1) *classes* and other types, (2) procedure and function *methods*, (3) global *variables*, and (4) *statements* for module initialization. Some of the declared entities are *exported* by the module and can be used by client modules (technically, identifiers of exported entities are marked with a '*' sign). Exported entities are *public*, while non-exported entities remain *private* in the module. An example of extensible module *M0* is shown in Fig. 2.

Module extension is a code reuse mechanism that enables building of new modules from existing ones. Module extension consists of *module embedding*, *class overriding*, and *method overriding*.

2.1.1 Module Embedding. An existing module *M0* can be *embedded* in a newly declared module *M1* (Fig. 3 contains an example). The embedding module *M1 inherits* all components of its embedded module *M0*. Only components that are exported by *M0* are visible in the embedding module *M1*; such

components are *re-exported* by *M1*. The embedding module may declare new components in addition to those inherited from its embedded modules.

```
module M0;
  type C* = class
    a*: integer; p: integer;
  end;
  var object*: C; private: C;
  procedure method*(var obj: C);
  begin { first version of method }
end;
begin { M0 } object.a := 0; end.
```

Figure 2. *Extensible module M0.*

```
module M1(M0);
  { C and object from M0
    are visible in M1}
  type C = class b*: integer; end;
  { C is extended with b: integer; }
  { C and object are now (a, p, b) }
  procedure method(var obj: C);
  begin { overriding method } end;
begin { M1 } object.b := 0; end.
```

Figure 3. *Extended module M1.*

2.1.2 Class Overriding. A class that is exported by an embedded module *M0* can be re-declared in its embedding module *M1*. A class definition in *M1* extends the definition inherited from *M0*. The extended class definition includes all components originally specified in *M0* and, in addition, new components specified in *M1*. The extended class definition *overrides* the class definition inherited from *M0*. Consider, for example, a class *C* declared in *M0* and extended in *M1*, and an *object* of class *C* that is exported by *M0*. (Fig. 3). Although the *object* is originally declared in *M0*, when inherited by *M1* it contains all components that belong to the extended class C, (i.e. *a, p,* and *b* in the example from Fig. 3).

2.1.3 Method Overriding. A method that is exported by an embedded module *M0* can be re-declared in its embedding module *M1*, provided that the procedure heading in *M1* is the same as in *M0*. The newly declared method implementation *overrides* the method implementation inherited from the embedded module. For example, the procedure *method* declared in *M1* overrides the procedure *method* declared in *M0* (Fig. 3). Any reference to this *method*, including references from within the embedded module *M0*, will invoke the method implementation declared in *M1*.

2.2 The Cellular Automaton Extensible Module

We outline module extension and demonstrate its applicability by focusing on a highly applicable generic algorithm for parallel scientific applications, the cellular automaton. The state of the cellular automaton is a square *grid* of *n x n cells* (Fig. 4). As a single *step*, the automaton updates the states of all individual cells. The state transition of a cell depends only on its current state and on the states of the adjacent cells. The automaton iterates over these *steps* in order to find a numeric solution for a given problem (such as Laplace equation).

The cellular automaton implements an individual step by means of a *q x q* mesh of parallel processes called *nodes* (Fig. 5). Each process node is assigned a sub-grid of *m x m* elements, where *n = m * q*. Each node updates its own sub-grid sequentially but all nodes do this simultaneously. During each step, nodes exchange boundary cells with their immediate neighbors through two-way communication *channels*. At the end, all nodes send their final sub-grids to a special *master* process which composes a final solution to the original problem.

The principal generic parameters of the parallel cellular automaton include the *cell* type and two sequential methods:

- a method to *generate* an initial state of an individual cell;
- a method to determine the *next state* of a cell as a function of its current state and the states of the adjacent cells.

The automaton provides its clients with a method to *compute* a solution of a specific problem. The compute method incorporates the master and the node processes, but those are transparent to the clients of the automaton.

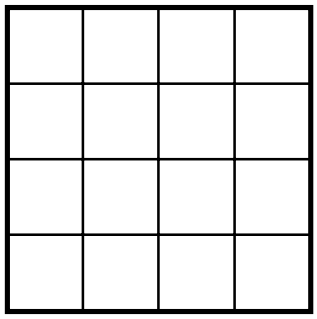

Figure 4. *The state of a cellular automation is a square grid of cells.*

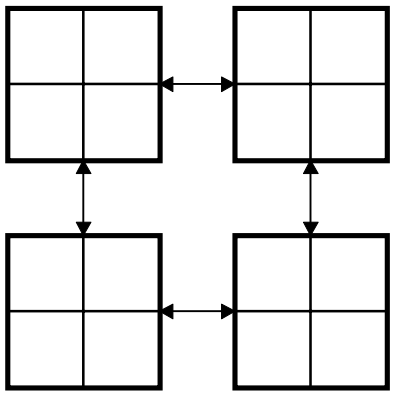

Figure 5. *Subgrids are allocated to parallel nodes.*

```
module PCA;

  type cell* = ..;

  { definitions of grid and subgrid types as arrays of cells; definition of channel type;
  definition of a communication network as matrix of individual channels }

  procedure nextState*(var u: subgrid;  i, j: integer); begin { virtual } end;
  procedure generate*(i, j: integer; var c: cell); begin { virtual } end;

  procedure compute*(steps: integer; var u: grid);
  begin { complete implementation that opens channels, spawns processes } end;

  procedure node(qi, qj, steps: integer; up, down, left, right, mast: channel);
  begin { complete implementation of a private node process } end;

  procedure master( c: network; var u: grid);
  begin { complete implementation of a private master process } end;

begin {… PCA…} end.
```

Figure 6. *An outline of extensible module Parallel Cellular Automaton, PCA.*

All constituents of the parallel cellular automaton (*PCA*) are encapsulated in an extensible module (Fig. 6). The parallel processes that constitute the cellular automaton do not utilize any problem-dependent information. Thus, the *cell* type is defined as an empty class. This empty class is extended with specific components when particular algorithms are derived from the generic algorithm, as demonstrated in the next section. Methods *generate* and *nextState*, which are supposed to initialize a cell and perform a state transition correspondingly, contain no statements at all and serve, in fact, as virtual

methods. Concrete versions of such virtual methods are provided for the derivation of particular algorithms.

The generic parameters *cell*, *generate*, and *nextState*, are exported by module *PCA*, together with a complete implementation of method *compute*. Clients of module *PCA* can (1) extend the *cell* class with problem-specific components, (2) provide domain-specific versions of methods *generate* and *nextState*, and (3) use method *compute* to solve particular problems.

The complete implementation of module PCA utilizes well established parallel programming language features, such as channel types and variables, messages, parallel and forall statements; we have adopted these features from [1]. For example, a channel capable of transmitting *cell* and *integer* values between two processes can be declared and utilized as shown on the left side of Fig. 7. The master and the node processes are implemented as private sequential methods that communicate by means of send and receive statements. All processes are generated by methods *compute* by means of familiar parallel and forall statements, as illustrated on the right side of Fig. 7.

```
type channel = *(cell, integer);
var
  chan: channel; c: cell; i: integer;
…
  open(chan);
  send(chan, i);   receive(chan, c);
```

```
parallel
  master(…master parameters…) |
  forall i := 1 to q do
    forall j := 1 to q do
      node(…node parameters…);
end;
```

Figure 7. *Sample parallel programming language features.*

The actual complete implementation of the generic cellular automaton, *PCA*, uses the problem *domain* as an additional generic parameter. This parameter is used similarly to the *cell* class and is not discussed in this paper for the sake of simplicity and shorter examples.

We use the generic cellular automaton to derive a hierarchy of more specific parallel algorithms, such as (1) successive over-relaxation, (2) Laplace heat equation, (3) genetic, and (4) job scheduling algorithms (see Fig.1). We achieve this by defining the *cell* class for each specific problem and by defining concrete sequential versions of methods *generate* and *nextState*. The derived algorithms are implemented as extensions of the parallel cellular module, *PCA*, as described in the next section.

3 Derivation And Implementation Of A Hierarchy Of Algorithms

3.1 Successive Over-Relaxation And Laplace Equation Algorithms

From the generic parallel cellular automaton we derive a *successive over-relaxation algorithm*, an iterative method that can be used to find numeric solutions of partial differential equations (Fig. 1). We restrict ourselves to equations for two-dimensional square regions. The square grid of cells is a discrete representation of such a region, with each cell representing a single point of the region. Technically, the algorithm is represented as a module *SOR* (an abbreviation for successive over-relaxation) which is an extension of module *PCA* (Fig. 8). The extended module redefines *cell* as a class with one real component, *t*, that represents a function value in the center of the cell, (2) defines *fopt*, a relaxation factor utilized by method *nextState*, (3) introduces a new generic parameter, a method called *residual* that is equation dependent, and (4) defines method *nextState* to perform successive over-relaxation in terms of method *residual*. We further extend module *SOR* into module *Laplace* by defining concrete versions of methods *residual* and *generate* that are specific for the heat equation (Fig. 8). The *compute* method, inherited unchanged from module *PCA*, uses these specific functions to represent a parallel algorithm for Laplace equation for stationary heat flow.

```
module SOR(PCA);
…
type cell* = class t*: real; end;
var fopt*: real;

function residual*(
  u: subgrid; i, j: integer): real;
begin { virtual } end;

procedure nextState*(
  var u: subgrid; i, j: integer);
  { 1 <= i <= m, 1 <= j <= m }
  var res: real;
begin res := residual(u, i, j) - u[i,j].t;
  u[i,j].t := u[i,j].t + fopt * res;
end; {nextState}

{ supportive methods, such as display
and initDomain }

begin { SOR } fopt := 2 - 2*pi / n;
end.
```

```
module Laplace(SOR);
…
function residual*(
  u: subgrid; i, j: integer): real;
begin
  residual := (u[i-1,j].t + u[i+1,j].t +
    u[i,j+1].t + u[i,j-1].t) / 4.0;
end;

procedure generate*
  (i, j: integer; var c: cell);
begin
  { initialize boundary or internal cell }
end;
…
begin { Laplace }
  …
  compute(steps, u);
  …
end.
```

Figure 8. *Derivation of successive over-relaxation and Laplace equation algorithms.*

3.2 Genetic And Job Scheduling Algorithms

Starting again with the generic parallel cellular automaton, we derive a parallel *genetic algorithm*, a method that can be used to find approximate solutions of intractable problems (Fig. 1). Our algorithm is based on the *fine-grain* (or *neighborhood*) model [9]. The algorithm is represented as a module *GA* (an abbreviation for genetic algorithm) which is an extension of the generic parallel cellular automaton module, *PCA* (Fig. 9). The extended module *GA* defines several new generic parameters: an empty class *gene*, virtual methods *mutate*, *crossover*, and others. Furthermore, *GA* defines a type *chromosome* as an array of *genes* and redefines *cell* as a class that incorporates one *chromosome* together with that chromosome's *fitness*. We then derive a distributed algorithm for *job scheduling* with penalties [8] by extending module *GA* into module *Jobs*. The extended module redefines *gene* as a concrete class with one *boolean* component and supplies concrete definitions of the *crossover* and *mutate* genetic operations that are specific for job scheduling with penalties (Fig. 9). Clients of module *Jobs* execute the algorithm by invoking method *compute* that is inherited form module *PCA*.

3.3 Performance Evaluation Of A Cluster Implementation

We use the object-parallel language *Paradigm/SP* to specify the generic *cellular automaton* and to derive more specific algorithms through module extension. We use the *Paradigm/SP* compiler to test these specifications as concurrent programs on a single-processor platform. After having validated a concrete algorithm, we convert it into efficient *C* code that runs in the PVM cluster computing

environment [10]. We have obtained performance results for the heat equation and for the job scheduling problem on a 10Mbps Ethernet cluster of Sun Ultra-1 workstations.

```
module GA(PCA);
…
type gene* = ..;
  chromosome* =
    array[1..chromoSize] of gene;
  cell* =
    class
      gene*: chromosome;
      fitness*: integer;
    end;
procedure mutate*(var c: cell);
begin { virtual } end;
procedure crossover*
  (p1, p2: cell; var c1, c2: cell);
begin { virtual } end;
{… some supplementary methods…}
procedure nextState*
  (var u : subgrid; i, j: integer);
begin { complete implementation
  using crossover and mutate }
end;
…
begin { GA } end.
```

```
module Jobs(GA);
…
type gene* =
  class
    g*: boolean;
  end;
…
procedure crossover*
  (p1, p2: cell; var c1, c2: cell);
begin { complete implementation of
the crossover genetic operator that
works well for job scheduling } end;

procedure mutate*(var c: cell);
begin { complete implementation of
the mutate genetic operator that is
relevant for job scheduling } end;

{ … some supporting methods … }

begin { Jobs }
… compute(steps, u); …
end.
```

Figure 9. *Derivation of genetic and job scheduling algorithms*

We have experimented with Laplace equation for temperature equilibrium in a square region with fixed temperatures at the boundaries. For multi-processor heat equation experiments, a single processor implements a subgrid of *500 x 500* cells. For varying number of processors, the subgrid dimension of $m = 500$ remains unchanged, i.e., the CPU load on each processor remains approximately the same. Furthermore, depending on the dimension of the $q \times q$ mesh of processors, the dimension of the global grid of $n \times n$ cells is $n = qm$. Table 1 shows the execution time $T(n, p)$ in seconds on a $q \times q$ mesh of p processors, $p = q * q$. The table also contains the processor efficiency $E(n, p) = T(n, 1)/(p*T(n, p))$ and the speedup $S(n, p) = p * E(n, p)$. The processor efficiency and speedup are higher for smaller number of processors.

Note that when the processor efficiency for *16* processors and a large grid of *2000 x 2000* cells is calculated using the equation $E(n, p) = T(n, 1)/(p*T(n, p))$, then $E(2000, 16) = 1.03$, i.e., it is greater than *1*; the corresponding speedup of 16.6 is greater than the number of processors. This result occurred because a single workstation was not capable of holding a whole *2000 x 2000* grid in memory (64MB in this case). The workstation had to swap pages between memory and disk during computations which resulted in poor single processor wall-clock time. We circumvented this difficulty by calculating the so called *scaled efficiency* [1] using the equation $E(n, p) = sqrt(p) * T(m, 1) / T(n, p)$, where $m = 500$ is the fixed subgrid dimension. The scaled efficiency $E(2000, 16) = 0.52$ is also shown in Table 1, together with the corresponding *scaled speedup* of *8.3*. Note that for smaller processor meshes ($p = 1, 4, 9$) and, correspondingly, smaller grids, both equations for $E(n, p)$ give the same values.

Table 1. *Laplace equation.*

p	n	T(n,1)	T(n,p)	E(n,p)	S(n,p)
1	500	85	85	1.00	1.0
4	1000	676	236	0.72	2.9
9	1500	2338	411	0.62	5.6
16	2000	10795	654	1.03; 0.52	16.6; 8.3

A state transition for the job scheduling problem is more computationally demanding than a state transition for the heat equation. In addition, job scheduling is based on a distributed genetic algorithm which requires a coarse grid, in contrast to Laplace equation solver which needs a finer grid. Thus, we implement a smaller subgrid of *24 x 24* cells on each processor when running multi-processor job scheduling experiments. As with the heat equation, the dimension of the global grid of *n x n* cells is *n = qm*, where *q* is the dimension of the mesh of processors. Table 2 shows the execution time *T(n, p)* in seconds on a *q x q* mesh of p processors, *p = q * q*, the processor efficiency *E(n, p) = T(n, 1)/(p*T(n, p))* and the speedup *S(n, p) = p * E(n, p)*. The processor efficiency is higher than 0.92.

Table 2. *Job scheduling.*

p	n	T(n,1)	T(n,p)	E(n,p)	S(n,p)
1	24	1950	1950	1.00	1.0
4	48	7802	1961	0.99	3.98
9	72	17559	2048	0.95	8.6
16	96	31218	2113	0.92	14.8

Figures 10 and 11 graphically represent the processor efficiency and speedup of the algorithms respectively. The job scheduling algorithm scales well on the grid dimension, *n*. The heat equation algorithm has more limited scalability due to its unfavorable proportion between communication and computation requirements.

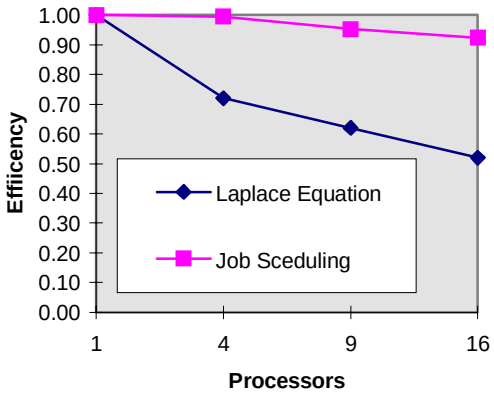

Figure 10. *Processor Efficiency*

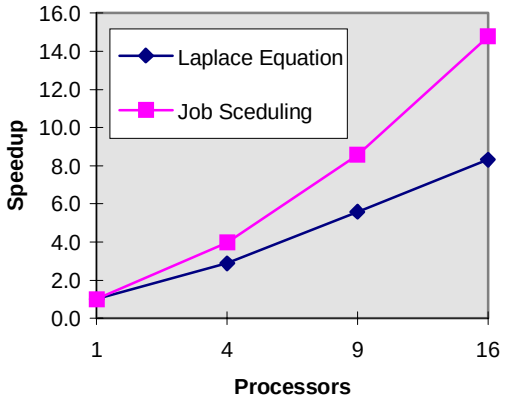

Figure 11. *Speedup.*

4 Conclusions

A parallel cellular automaton for multi-computers was first developed by Hansen [1] and applied to forest fire simulation and to the heat equation. We have adapted the cellular automaton to a cluster computing environment, in which message passing is generally slower than in a multi-computer. We have also derived a *hierarchy* of algorithms rather than isolated applications. Our derivation process is

supported by special linguistic constructs, module extension, and by a compiler. Module extension can be valuable when modules and classes are used in conjunction, and, therefore, should be extended together. Furthermore, module extension can be applied to define and implement extensible typeless entities, such as abstract data structures or libraries of functions.

We have adopted most of the algorithms for the generic parallel cellular automaton from [1], except for the global output algorithm which is suitable for a multi-computer but not for a cluster of workstation. Our node processes send their final states to the master through designated direct channels rather than through channels that connect adjacent nodes [1]. Furthermore, our relaxation algorithm is an adaptation of the algorithm proposed in [1], while the parallel genetic algorithms and the job scheduling algorithm are our contribution.

We have demonstrated that extending a generic parallel algorithm with *sequential* domain-specific code results in relatively good parallel performance in a cluster-computing environment. The introduction of a hierarchy of algorithms leads to cleaner implementation of a variety of scientific problems. In addition to providing good processor performance, our framework improves programming efficiency, allowing the application developer to focus on the sequential implementation of domain-specific details, rather than on the more difficult parallel code development.

In traditional modular object-oriented languages, such as Oberon-2, Ada-95, and Modula-3, modules are not extensible, while classes are represented by means of extensible record types [3]. The difference in our approach to classes is that class extension overrides an existing class and does not introduce a new type. The Cecil language [4] supports a form of module import called module extension. In Cecil extended modules are shared and are, in fact, similar to imported modules. Cecil modules allow standard subtyping while our extensible modules allow type redefinition.

It has been recognized [2] that both modules and classes support necessary abstractions, which should be used as complementary techniques. Overview of object-parallel language features can be found in [5, 6].

5 References

[1] P.B Hansen, *Studies in Computational Science: Parallel Programming Paradigms*, Prentice Hall, 1995.

[2] C. Szyperski, *Why we need both: Modules and classes*, in Proceedings of OOPSLA, ACM Press, 1992.

[3] N. Wirth, *Type extensions*, ACM Transactions on Programming Languages and Systems, 10 (1987), 204-214.

[4] C. Chambers and G. Leavens, *Type checking and Modules for Multimethods*, ACM Transactions on Programming Languages and Systems, 17 (1995), 805-843.

[5] G. Wilson and P. Lu, editors, *Parallel Programming using C++*, MIT Press, 1996.

[6] J.-P. Briot, J.-M.Geib, and A. Yonezawa, *Object-based parallel and distributed computation*, in Lecture Notes in Computer Science 1107, Springer, 1996.

[7] A. Radenski, A. Vann, and B. Norris, *Parallel Probabilistic Computations on a Cluster of Workstations*, in Parallel Computing: Fundamentals, Applications and New Directions, Elsevier, 1998.

[8] S. Baase, *Computer Algorithms: Introduction to Design and Analysis*, second edition, Addison-Wesley, 1988.

[9] E. CantuPaz, A Summary of Research on Parallel Genetic Algorithms, IlliGAL Report No.95007, July 1995.

[10] V. Sunderam, PVM: A Framework for Parallel Distributed Computing, *Concurrency: Practice and Experience*, 2 (1990), 315-339.