Hochschule Reutlingen
Reutlingen University

Parallel and Distributed Computing Group
Department of Computer Science
Reutlingen University

Aspect-oriented component assembly—a case study in parallel software design

Clemens Dangelmayr and Wolfgang Blochinger

(Accepted Peer-Reviewed Manuscript Version)

BibTeX

```
@article{Dangelmayr2008,
       author = {Dangelmayr, Clemens and Blochinger, Wolfgang},
       title = {Aspect-oriented component assembly—a case study in parallel
                software design},
       journal = {Software: Practice and Experience},
       year = {2009},
       volume = {39},
       number = {9},
       pages = {807--832},
       doi = {10.1002/spe.912},
       url = {https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.912}
}
```

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Aspect-Oriented Component Assembly—A Case Study in Parallel Software Design

C. Dangelmayr and W. Blochinger*,†

University of Tübingen, Symbolic Computation Group, Sand 14, D-72076 Tübingen, Germany

SUMMARY

In this paper we deal with building parallel programs based on sequential application code and generic components providing specific functionality for parallelization like load balancing or fault tolerance. We describe an architectural approach employing aspect-oriented programming to assemble arbitrary object-oriented components. Several non-trivial crosscutting concerns arising from parallelization are addressed in the light of different applications which are representative for the most common types of parallelism. In particular, we demonstrate how aspect-oriented techniques allow us to leave all existing code untouched. We evaluate and compare our approach to its counterparts in conventional object-oriented programming.

KEY WORDS: aspect-oriented programming; parallel programming; code re-use; software metrics

1. Introduction

Parallel software development has recently gained considerable importance. With the advent of multi-core processors, traditional (sequential) software is only able to utilize a fraction of the available computational power. Since parallelism must be exploited on the application level, this new approach in microprocessor architecture requires substantial effort in the development of appropriate software. Also, the increasing popularity of high performance computing clusters built from commodity components further drives the demand for parallel applications.

The software developer typically employs parallel libraries or platform components to access the functionality of the underlying parallel hardware. Traditionally, for parallelization one has to make modifications to existing sequential code, e.g. restructuring parts of the code

*Correspondence to: University of Tübingen, Symbolic Computation Group, Sand 14, D-72076 Tübingen, Germany
†E-mail: blochinger@informatik.uni-tuebingen.de

Received
Revised

and inserting API calls at appropriate places. Most often, when switching to another parallel architecture different components have to be used. Thus, further modifications of the sequential code become necessary.

Our work especially considers the fact that most often there exist highly optimized sequential application code and conventionally implemented components of a parallel platform addressing all functional and non-functional requirements of parallelization. In general, both parts are developed by independent parties and integrated by a third party. Often, the parts accumulate considerable domain-specific knowledge and are also subject to further development.

In this setting, non-intrusive techniques, i.e. an approach leaving existing code completely untouched, are highly desirable. Subsequently, we show how techniques from aspect-oriented programming [31] can be beneficially employed to address this specific issue. Specifically, we make the following contributions:

1. We investigate the applicability of our approach in the light of representative applications which exhibit different types of parallelism. We study several non-trivial crosscutting concerns arising from parallelization and show how we can address them using generic reusable components.
2. We describe a versatile approach to component assembly utilizing aspect-oriented bridge templates to assemble arbitrary object-oriented software artifacts as lightweight components. We identify several specific assembly issues and discuss how they can be solved in an elegant way by aspect-oriented implementations of common design patterns.
3. We evaluate and compare our approach to its counterparts in pure object-oriented programming. Here, we identify and quantify suitable code quality metrics. Our analysis shows significantly improved code quality metrics and full reusability of all employed components. Additionally, we conduct performance measurements which give evidence that our techniques do not compromise parallel efficiency.

The remainder of our paper is organized as follows. In Section 2, we give a brief overview of aspect-oriented concepts. Section 3 discusses our approach in detail. In Section 4, we present code quality metrics and performance measurements. Section 5 deals with related work. Our Conclusion (Section 6) discusses the benefits and limits of our method.

2. Aspect-Orientation

Like most other imperative programming paradigms, object-oriented programming exhibits one significant drawback. It neglects the crosscutting nature of the various concerns of complex software systems. Thus, the programmer is forced to apply a decomposition (*dominant decomposition* [54]) which is constricted to one dimension. Whatever concrete approach of decomposition is chosen, there will always be functionality scattering throughout the code. Typical examples of this phenomenon are logging, persistence [41], and security [45]. The tangling of modules, as a similar effect, mixes up code fragments of different concerns. Scattering and tangling both lead to greatly reduced reusability and error-prone redundancies. Thus, dominant decomposition seriously impairs keeping different concerns separate. This

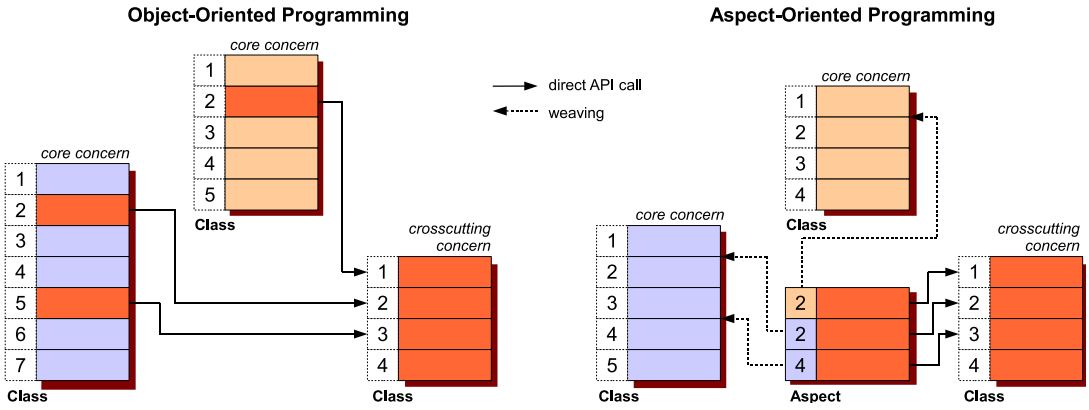

Figure 1. Object-Oriented vs. Aspect-Oriented Integration of a Crosscutting Concern.

applies to object-oriented programming as well as most imperative programming paradigms (for example *COBOL* [15]).

Technically, in common imperative programming languages these issues result from the necessity to group consecutive functionality in the same compilation unit. Instructions that are to be executed consecutively must be stated directly after another, even when they address completely different concerns. In contrast, aspect-orientation permits to explicitly state the position (or positions) where to insert code sequences. Specifically, it allows to separately define instruction sequences together with their position in the code in an abstract and flexible way. These groupings correspond to the different concerns of the system.

Aspect-orientation distinguishes between core and crosscutting concerns. Crosscutting concerns cut across other concerns—they are assumed orthogonal to each other. To realize a system that needs to address different concerns, aspect-orientation recommends *separation of concerns* (coined by Dijkstra in [14]). After identifying the required concerns (compare [35, 51]), core concerns are implemented as conventional components, while crosscutting concerns are implemented as *aspects*. (There is a whole array of understandings of the concept aspect; compare [30, 34, 42, 40, 11]. We will illustrate our own viewpoint in detail in Section 3.2.3.)

Ideally, all concerns can be addressed independently and implemented self-contained. At some point in the development process, however, crosscutting concerns have to be integrated into the concerns they cut across. This is illustrated in Figure 1. In the object-oriented case, the crosscutting concern (e.g. logging) is integrated into the core concerns by inserting corresponding method calls (e.g. logging statements) into the intercepted concern implementation. Aspect-oriented programming, on the other hand, allows us to explicitly state where to insert additional code sequences using aspect declarations. This keeps the intercepted code base untouched and corresponding concerns separate. An *aspect weaver* reads out these aspect declarations and inserts appropriate statements during compilation.

3. Component Assembly

In this section, we discuss our aspect-oriented approach to build a parallel application from sequential code and generic components of a parallel platform. The specific challenge we address is to assemble the original sequential algorithm and all components—neither modifying sequential code nor the implementation of the components.

Section 3.1 gives an overview of our example applications, methods for their parallelization, and the required system functionality. (A detailed treatment of this topic can be found in [22].) Particularly, our discussion focuses on identifying crosscutting concerns.

In Section 3.2, we demonstrate how aspect-oriented techniques can be applied to achieve our goal. To assemble all software artifacts (application code and platform components), we employ tools provided by *AspectJ* [30] which is the most common language extension to Java for aspect-oriented programming.

Our discussion in Section 3.3 shows benefits from using aspect-oriented instead of conventional object-oriented programming. (In contrast to this qualitative analysis, we compare both approaches quantitatively in Section 4.)

3.1. Concerns and Components

In order to demonstrate the wide applicability of our approach we have chosen three example applications which exhibit different types of parallelism (see Section 3.1.2).

Typically, all of our example applications show a high degree of irregularity. An irregularly structured problem is an application whose computation and communication patterns are input-dependent, unstructured, and/or evolving during the computation. Consequently, in order to minimize parallel overhead (e.g. idle times of processors and communication) several decisions have to be made at run-time (e.g. problem decomposition, load balancing, or allocation of communication channels). Such advanced functionality is typically provided by sophisticated, pre-existing components of a parallel platform (see Section 3.1.3). This setting enables us to discuss several non-trivial assembly issues.

3.1.1. Example Applications

Quicksort. Divide-and-conquer-algorithms recursively partition workload into smaller units, which can be handled in parallel. Quicksort, as our first example application, recursively divides the list to be sorted into smaller sublists separated by a pivot element. Smaller elements end up in one sublist, larger ones in the other. Then, both lists can be sorted recursively. A basic *Java* implementation of the sorting method is shown in Listing 1.

Matrix Multiplication. Our second example application is matrix multiplication. When computing the product of a matrix and a vector, every element of the result vector is obtained by multiplying the corresponding row of the matrix with the input vector—as highlighted for the second element of the result vector in the following equation.

```
void sort(Range range) {
   if (!range.partitionable()) return;
   Element pivot = getElement(range.center());
   Partition partition = partition(range, pivot);
   sort(partition.lowerHalf());
   sort(partition.upperHalf());
}
```

Listing 1. Basic Quicksort Method.

$$\begin{pmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1m} & a_{2m} & \dots & a_{nm} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_{11}b_1 + a_{21}b_2 + \cdots + a_{n1}b_n \\ a_{12}b_1 + a_{22}b_2 + \cdots + a_{n2}b_n \\ \vdots \\ a_{1m}b_1 + a_{2m}b_2 + \cdots + a_{nm}b_n \end{pmatrix}$$

Listing 2 shows (restricted to relevant methods) a basic implementation of a `Matrix` class allowing for multiplication with other matrices and/or vectors.

```
class Matrix {
   Matrix multiply(Matrix b) {
      Matrix c = new Matrix();
      for (int i = 0; i < b.getWidth(); i++)
         c.setColumn(i, multiply(b.getColumn(i)));
      return c;
   }
   Vector multiply(Vector b) {
      Vector c = new Vector();
      for (int i = 0; i < getHeight(); i++)
         c.setElement(i, getRow(i).multiply(b));
      return c;
   }
}
```

Listing 2. Basic Matrix/Vector Multiplication.

State Space Search. Discrete optimization problems are concerned with finding elements out of a given (finite and discrete) set of possible solutions maximizing or minimizing a certain function. These are often tackled by search techniques based on a *state space graph*. Paths in this graph represent possible solutions. The state space graph is constructed step-by-step heading for an optimal solution. Thus, the optimization process basically constitutes a graph traversal. A generic graph search algorithm employs an *agenda*, which contains all currently open nodes of the state space graph: the search frontier. In every step an element of the agenda is extracted, visited, and expanded if possible. Discovered graph nodes are appended to the agenda. This process is repeated until the agenda is empty. A basic traversal algorithm is illustrated in Listing 3.

```
void traverse(Agenda agenda, Node root) {
   agenda.append(root);
   while (!agenda.isEmpty()) {
      Node node = agenda.extract();
      node.visit();
      for (Node successor : node.expand())
         agenda.append(successor);
   }
}
```

Listing 3. Basic State Space Traversal Algorithm.

Often additional (heuristic) information is used for steering the graph traversal. As our third example application we use a *branch-and-bound* approach tackling instances of the *travelling salesperson problem* (TSP). Here, the state space tree is only expanded (branched) as long as the length of the constructed tour does not exceed the length of the best tour already found (bound). This can considerably reduce the amount of nodes to be visited.

3.1.2. Parallelization Techniques

In [22], parallel task decomposition is classified into recursive, data, explorative, and speculative decomposition. *Recursive decomposition* typically employs a divide-and-conquer scheme, recursively-decomposing tasks into smaller subtasks, which can be mapped to different processors. *Data decomposition* can be applied to programs operating on large data sets. These data sets are partitioned into subsets, which are processed in parallel. *Explorative decomposition* is usually applied to search procedures. The search space is divided into disjunctive subspaces, which are treated by different processors. (*Speculative decomposition* is of less relevance for our topic. Typically, it is transparently applied by modern CPUs on the instruction level.) Subsequently, we will see that each of these three types of parallelism is exhibited by one of our example applications.

An example for recursive decomposition is quicksort. Here, tasks are associated with recursive calls to the sorting method `sort` (compare Listing 1). These are associated with parts of the sorting range, which can be sorted independently. An approach to parallelization lies in mapping calls to this method to different processors, thus distributing associated workload over available computation units. Depending on actual size of the two sublists irregular computational effort can occur.

In case of matrix multiplication (as an example for data decomposition), parallelism lies in the concurrent computation of elements of the result vector/matrix. Thus, parallelization breaks down to partitioning loop ranges and distributing sub-ranges over participating processors (loop parallelization). For sparse matrices with an unstructured sparsity pattern (e.g. found in simulation applications) a high degree of irregularity results.

State space search is a typical example for explorative decomposition. Parallel execution is achieved by distributing open nodes of an agenda over participating processors. Thus, every processor maintains its own agenda and executes the algorithm presented in Listing 3. As explained above, optimization procedures like state space search often generate additional knowledge (e.g. bounds in a branch-and-bound method) during execution, which can greatly

advance the optimization process. In a distributed parallel environment this knowledge is generated at different locations, requiring its explicit exchange between processors to increase overall efficiency. State space search can exhibit a high degree of irregularity since it is in general not possible to estimate the costs for traversing a specific region of the state space graph.

3.1.3. Components

We assume the availability of suitable, pre-existing components addressing non-trivial crosscutting concerns like load balancing or fault tolerance. (In this case, effort for re-implementing corresponding functionality outweighs effort for assembly with existing components.) We assemble parallel versions of our example applications using the components described subsequently. However, our assumptions on specific functionality to be provided are general such that other libraries or custom components could be employed as well. Moreover, the assembly techniques presented in this work can be employed for arbitrary parallelization and (in general) assembly scenarios. Thus, specifics of the described interfaces or implementation details in general are only detailed to be able to concisely illustrate the different assembly issues.

Before we describe the components we employ, we first define how the term component is used in conventional component-based software engineering and how this relates to the concept component in our context. In [12], a component is described as a building block of a computer program that conforms to a component model. *Component models* define how to assemble components (*composition standard*) and provide an infrastructure offering commonly required mechanisms like persistence, security, or communication. When complying with the composition standard, components can be composed without modification.

In our setting, we presume no constraints on assembly imposed by a component model or composition standard. Thus, arbitrary (object-oriented) software artifacts as *lightweight* components can be assembled for the desired deployment scenario. (We use the term *assembly* in contrast to composition, as we explicitly presuppose no composition standard.) Initially, all components, application or platform components, are implemented fully self-contained. That means no implementation or compilation dependencies between any two of them impair concurrent development.

This is crucial, when acting under the assumptions made in Section 1. On the one hand, different components constituting a final program may have been developed by independent parties. (Even single platform components may be procured from varying sources.) On the other hand, the sequential application will determine in most cases the execution context. This holds especially for subsequent parallelization of pre-existing, already deployed programs. In this case, only lightweight components as described above will be suitable—in contrast to (heavyweight) components presupposing (e.g.) dedicated application servers.

Load Balancing. We assume very generic load balancing functionality realized by the following concepts. The balancing component monitors the instantiation and handling of dynamically created content (workload), which is to be distributed dynamically. Components requiring this functionality must call methods of the interface shown in Listing 4.

```
interface Balancer {
   void notifyInstantiation(Object w);
   void notifyHandling(Object w);
}
```

Listing 4. Interface: Balancer.

By observing relevant changes, an attribute (like the workload status) is computed. A constraint on this attribute is reviewed regularly. If it applies, the balancing component chooses a neighbor from the set of known processors. Balancing actions are then realized through exchanging content between the local processor and this neighbor. To manipulate the local status the balancing component uses dedicated methods of the callback interface `BalancerCallback` (compare Listing 5). Every processor must supply an implementation of this interface to the balancing component.

```
interface BalancerCallback {
   void inject(Object w);
   void extract(Object w);
}
```

Listing 5. Interface: Balancer Callback.

Fault Tolerance. In this context, fault tolerance is restricted to losses of remotely handled workload. The corresponding component creates local copies whenever a task is delegated to other processors. An example for a delegated task is the remote traversal of a sub-tree in state space search, whose root node has been transferred by the balancing component. When fault detection signalizes that a processor went off-line which previously received workload, the local copy of the task is re-instantiated on the processor it originally was extracted from.

Termination Detection. Another concern to be addressed in our context is detection of the termination of a parallel computation, i.e. every created task has been executed. An explicit termination detection protocol is required, since every processor in the system can dynamically create new tasks without central control. Here, the corresponding component uses a *fixed-energy-based* approach [22]. This approach to termination detection uses the notion of a fixed quantity within the system, often termed "energy". When the parallel computation starts, processor P_0 has a single task representing the whole problem and is associated with an energy of 1. When dynamic problem decomposition is carried out, processor P_0 keeps one half of the energy and gives the other half to the processor receiving the corresponding task, such that each of the two processors has an energy of 0.5. This principle is applied every time when the work of a processor is partitioned. When a processor completes its task, it returns its current energy to the processor from which it received the work where it is added to the current energy. Termination of the parallel computation can be determined when processor P_0 has no more work and its energy becomes 1.

Distribution. Crosscutting concerns addressed so far require functionality to coordinate concurrent control flows on different processors. Technically this can be realized by a remote procedure call abstraction. We assume that the parallel platform provides (asynchronous)

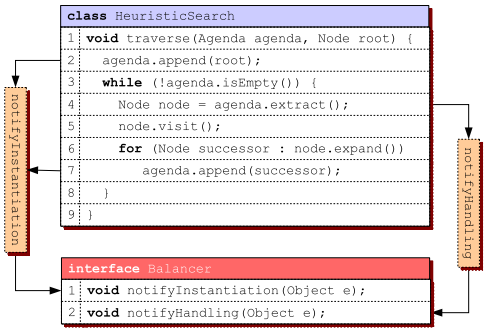

Figure 2. Monitoring Node Insertion/Extraction.

remote procedure calls in various manifestations like uni-, multi-, or broadcasts. Besides that, this component provides commonly required functionality to realize distributed data structures or shared objects.

3.2. Aspect-Oriented Component Assembly

This section describes different assembly issues and how aspect-oriented programming can be employed to address these issues appropriately. We illustrate the application of each of these techniques in the light of one of our example applications. Our discussion of the different assembly techniques adheres to the following structuring:

1. We describe specific requirements for assembly.
2. We introduce the relevant aspect-oriented concepts and techniques.
3. We show how to apply these constructs to address the requirements.

We close with a summary and discussion of the employed techniques in Section 3.2.7.

3.2.1. Monitoring of Internal State with after*-Advice*

As mentioned, parallelization of state space search can be achieved by dynamic balancing of open nodes of the agendas. Thus, the balancing component must be kept up-to-date on relevant changes, i.e. local instantiation or handling of workload. When such an event takes place, observing methods of implementations of the balancer interface must be called. For example, we assume workload as instantiated with the insertion of nodes into the agenda. Thus, we must forward corresponding events to the balancing component. This is illustrated in Figure 2 for instantiation as well as handling of workload. Similarly, the fault tolerance and termination detection components must be informed about workload being transferred between different processors.

Aspect-Oriented Concepts and Constructs. The basic construct in aspect-oriented programming are *join points* which are well-defined points in the control flow, e.g. method calls, accesses to fields, or handling of exceptions. In the light of the drawbacks of imperative programming paradigms discussed previously (see Section 2), this concept becomes of primary interest because at join points additional code can be inserted. In the example for state space search (see Listing 3), relevant join points are calls to the methods `append` and `extract` of the agenda. *Pointcuts* allow the abstract declaration of sets of join points. To state pointcuts, AspectJ offers a sophisticated template language. Pointcuts can include calls to methods complying with a specified signature. An example for a pointcut is `call(void Agenda+.append(Node))` which intercepts calls to the method `append` of all classes implementing the `Agenda` interface. To pointcuts *advices* can be applied. Advices consist of instruction sequences using reflexive information about the joined execution point to direct their control flow.

Application. Listing 6 shows advice applied to pointcuts forwarding control flow to the balancer interface.

```
after(Node n) : call(void Agenda+.append(Node)) && args(n) {
    balancer.notifyInstantiation(n);
}
after() returning(Node n) : call(Node Agenda+.extract()) {
    balancer.notifyHandling(n);
}
```

Listing 6. Pointcut/Advice: Intercepting Node Insertion and Extraction.

The balancing component is now notified about the insertion of nodes into and extraction of nodes from the monitored agenda. Fault tolerance and termination detection components use similar pointcuts and advice to observe when nodes are transferred to other processors.

3.2.2. Manipulating Internal State with Introductions

As the balancing component requires a suitable callback interface (`BalancerCallback`), an adapter (see *adapter pattern* [19]) to the local agenda must be interconnected. In general, components must be able to manipulate internal state in other components. Thus, additional assembly code is required to make necessary adaptations between involved components.

Aspect-Oriented Concepts and Constructs. Here, *introductions* [33] come into play. Introductions statically extend the signature of classes by adding fields or methods, introducing implemented interfaces, or modifying inheritance structures. As the visibility scopes of Java apply to aspects as well, private fields of modified classes are not accessible from introduced methods. This condition can be relaxed by using *privileged aspects*.

Application. We enrich the signature of classes implementing the interface `Agenda` (in our case we employ the class `Stack` to realize a depth-first-search [22]) by the methods of the

interface `BalancerCallback`. These allow modifying the set of open nodes, referenced by the `private` field `nodes` in Listing 7.

```
privileged aspect BalancingCallbackAdapter {
   declare parents: Stack implements BalancerCallback;
   void Stack.inject(Object w) { nodes.add(w); }
   void Stack.extract(Object w) { nodes.remove(w); }
}
```

Listing 7. Privileged Aspect: Balancing Callback Adapter.

3.2.3. Decoupling Components with Bridge Templates

We have shown in Listing 6 how to connect different components without having to modify them. Subsequently, we discuss an approach to decouple components and to be able to easily connect other applications or, in general, core concerns to the platform components.

Aspect-Oriented Concepts and Constructs. AspectJ allows deferring the specification of pointcuts by permitting them to be separately stated, named, or even declared *abstract* (similar to abstract methods and classes in conventional Java). They act as placeholders for concrete pointcuts to be instantiated later. The same applies to *aspects*, the main representation of a compilation unit in AspectJ. An aspect is a language construct that represents the encapsulation of a number of code elements, like pointcuts or advices.

Our definition of the concept aspect is based on [33]. Here, an aspect is described as the modular implementation of the *client-part* of a crosscutting concern. (The term client or client-part is not to be confused with its common meaning in client-server-computing.) In this context, one distinguishes between code actually crosscutting other concerns (modularized as client-part) and the conventionally implemented server-part. This allows a modular and reusable implementation of the server-part of the concern, while only the client-part has to be changed when adapting to a new environment. In our case, server-parts are pre-existing, object-oriented platform components.

Application. To connect components we interpose aspect-oriented bridges (see *bridge pattern* [19]) between them. Bridges allow further decoupling components by mediating control flow. The bridge, together with the code connecting related components, represents the client-part of another component acting as server. Assembling components then requires connecting client as well as server components (server-part) to the bridge. While an object-oriented implementation may be sufficient for connecting the server-part (in most cases this would be an adapter), connecting the client to the bridge in a non-intrusive fashion entails the full bandwidth of aspect-oriented tools.

To allow applications to use the services of the balancing component, the client-part contains abstract pointcuts (as elements of the respective bridge template, see *template pattern* [19]). Connecting the client component to the bridge then requires to instantiate these pointcuts. Both, together with their respective advice, are shown in Listing 8, representing the bridge template.

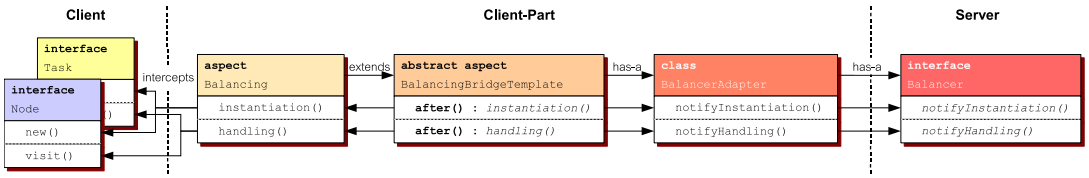

Figure 3. Component Bridge connecting Client- and Server-Part.

```
abstract aspect BalancingBridgeTemplate {
   abstract pointcut instantiation(Object w);
   abstract pointcut handling(Object w);
   after(Object w) : instantiation(w) {
      balancer.notifyInstantiation(w);
   }
   after(Object w) : handling(w) {
      balancer.notifyHandling(w);
   }
}
```

Listing 8. Aspect Template: Balancing Bridge.

To integrate the balancing component with a distributed application, one must instantiate both pointcuts. This allows the balancing component to observe changes to the local workload status and to take necessary balancing actions. Figure 3 illustrates how all discussed techniques work together. In general, a number of equivalent pointcuts can be used to inject the same crosscutting functionality. For workload instantiation/handling we intercept instantiations of classes implementing the interface `Node` and calls to the method `visit`—as alternatives to intercepting `Agenda+.append`/`Agenda+.extract`.

3.2.4. Integrating Low-Level Functionality with Annotation Introduction

As outlined in Section 3.1.2 advanced state space search algorithms often generate additional heuristic knowledge which must be exchanged between processors to increase efficiency. Thus, tools are required which allow to tag the specific execution points where this heuristic content is generated such that new heuristic information can be broadcast upon encounter. Tagging such execution points is a special case of a common requirement in the creation of complex systems: integrating low-level concerns like logging, persistence, or security. This can be accomplished using annotations. Section 3.2.7 further discusses the benefits of annotation-based approaches.

Aspect-Oriented Concepts and Constructs. In Section 3.2.2, we discussed the use of introductions. Besides allowing to insert new fields and methods, they enable us to attach annotations. Calls to methods tagged this way can then be intercepted by pointcuts incorporating these annotations.

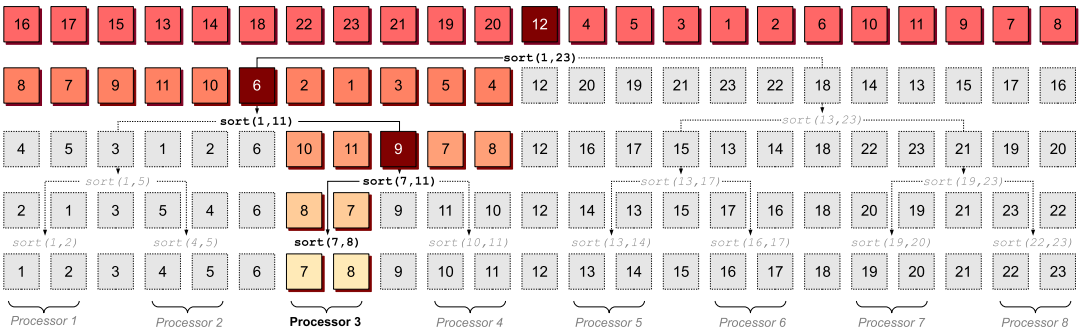

Figure 4. Parallel Quicksort. Local Method Calls are Highlighted.

Application. We illustrate the use of annotating join points on the example of propagating heuristic information—as an element of the client-part of the communication component. To declare method calls to be propagated, we attach annotations to dedicated methods. For our traveling salesperson example these are the methods setting newly found bounds (as specified in Listing 9).

```
declare @method : void Bound.setValue(double) : @Distribute;
```

Listing 9. Attached Annotation: Tagging Heuristic Knowledge.

Propagating these method calls is then realized with after-advice (as illustrated in Listing 10).

```
abstract aspect HeuristicKnowledgeDistribution {
   abstract void propagate(RPC rpc);
   after() : call(@Distribute * *(..)) {
      propagate(new RPC(thisJoinPoint));
   }
}
```

Listing 10. Aspect Template: Heuristic Knowledge Distribution.

3.2.5. Non-intrusive Adaptation of Component Functionality with around*-Advice*

Section 3.1.2 outlines how quicksort can be parallelized. Every processor executes the algorithm in Listing 1. However, instead of executing all calls to `sort`, different processors execute different subsets of all initiated calls. This is illustrated in Figure 4. In this version of parallel quicksort, different processors are responsible for different sub-ranges of the total sorting range (determined by subsequent choices of pivot elements). Thus, a non-intrusive technique which allows intercepting and possibly circumventing the execution of selected method calls is needed.

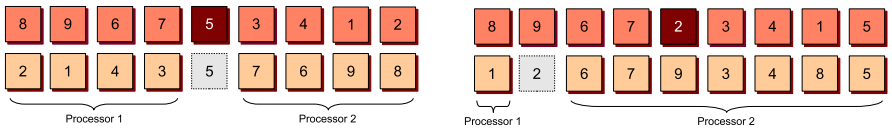

Figure 5. Favorable vs. Unfavorable Problem Configuration.

Aspect-Oriented Concepts and Constructs. Aspect-orientation allows us to non-intrusively redirect the inner control flow of arbitrary components. Here, we use the *around*-advice construct, which is able to intercept and circumvent the execution of selected statements. (In contrast to after-advice, around-advice is executed instead of the intercepted join point and not after it.)

Application. Listing 11 shows an aspect template intercepting and circumventing calls to dedicated methods. Methods tagged with `@Task` are intercepted. The corresponding join point is then mapped to a processor. Only if this processor equals the local processor, the method call is executed with `proceed()`.

```
abstract aspect TaskDispositionTemplate {
   abstract Processor mapToProcessor(JoinPoint joinPoint);
   void around() : call(@Task * *(..)) {
      if (mapToProcessor(thisJoinPoint).equals(localProcessor))
         proceed();
   }
}
```

Listing 11. Aspect Template: Task Disposition.

To instantiate this template, we tag `sort(Range)` and let `mapToProcessor` implement the mapping function illustrated in Figure 4. Now, every processor executes the same algorithm but only a subset of all initiated calls to `sort`.

With the problem configuration illustrated in Figure 4, workload is distributed equally over participating processors. In general, workload balance between processors depends upon the subsequent selection of pivot elements. This is illustrated in Figure 5 for different problem configurations. Thus, in general, dynamic load balancing becomes additionally necessary. Prematurely idle processors request unsorted sub-ranges, corresponding to calls to `sort`, from non-idling processors. Here, `Range` objects represent exchangeable content (compare Section 3.2.3). By implementing `BalancerCallback`, we incorporate the delegation of sub-ranges into processor mapping, superposing the application-specific part illustrated in Figure 4.

The application range of the around-advice construct is extremely wide, as it can be applied on every level (e.g. to intercept a high-level method) and is able to redirect the control flow of arbitrary software elements. Even a non-intrusive adaptation of the parallel platform, e.g. to cope with unreliability is possible as we discuss subsequently.

Tackling Unreliability with Non-intrusive Behavior Adaptation. In some cases the functionality of one or more of the available components may be inadequate for a specific application or deployment scenario. Thus, an adaptation of the inner behavior of components becomes necessary. A typical example is dealing with high unreliability, e.g. prevailing in large, loosely coupled distributed systems where participating processors frequently join and leave. Often, unreliability cannot be tackled efficiently in a fully generic way, such that application specific measures become mandatory.

In addition to generic fault tolerance measures (see Section 3.1.3), *eager scheduling* [36] can be an efficient way to tackle high probability of losses of tasks. The basic principle is to dynamically assign a task which is already scheduled (but yet uncompleted) to additional processors when its result is not delivered within a given time frame. To introduce this sub-aspect into the parallel platform, we intercept the lookup method for candidates for remote handling of the balancer component and add already distributed tasks to the list of potential candidates for remote execution (as illustrated in Listing 12).

```
abstract aspect EagerSchedulingTemplate {
   abstract pointcut workload();
   abstract Collection getDelegatedTasks();
   Collection around() : workload() {
      Collection candidates = new LinkedList(proceed());
      candidates.addAll(getDelegatedTasks());
      return candidates;
   }
}
```

Listing 12. Aspect Template: Eager Scheduling.

While the method `getDelegatedTasks` retrieves tasks which are already scheduled, the abstract pointcut `workload` is used to intercept access to objects qualifying as candidates for workload exchange. In Listing 13, we instantiate the originally abstract pointcut to intercept read access to the field `load` of the `LoadBalancer` object.

```
pointcut workload() : get(Collection LoadBalancer.load)
   && withincode(void LoadBalancer.selectContent());
```

Listing 13. Instantiated Pointcut: Workload Access.

In highly volatile settings, processors going off-line should also try to minimize lost workload by sending back partially handled workload. In our state space search example, partially handled workload is constituted by unvisited nodes of a sub-tree, either still in the agenda or present as reference on another node. Here, the introduced recovery aspect requires to instantiate an abstract pointcut, whose advice collects these findings and sends them to their appropriate receivers. The employed parallel platform must contain join points qualifying for an instantiation of this abstract pointcut which are e.g. join points indicating an upcoming shutdown of the processor.

In addition, the technique can be used to realize specific routing functionality to compensate the lack of direct communication channels. Here, we intercept the sending of messages in analogous manner. When the target processor of a message is not visible, the message is

encapsulated in a special routing message, which is sent to a currently visible, randomly chosen processor. When this routing message arrives at the remote processor, it passes the original message to the communication subsystem. Re-routing is repeated until the message is finally delivered or a termination criterion applies.

3.2.6. Restructuring Control Flow with loop-*Pointcuts*

Data decomposition requires partitioning data sets into disjunctive subsets, which can be handled in parallel. In most cases, this means partitioning control flow structured by `for`-loops. In our matrix multiplication example as implemented in Listing 2 different iterations of the loop compute different elements of the result matrix/vector. Thus, we subdivide the loop range into partitions processed by different processors.

Aspect-Oriented Concepts and Constructs. Aspect-oriented programming with AspectJ unfortunately still lacks the possibility to intercept single loop iterations on a structural level. In [26], the authors deal with this shortcoming. A *loop*-Pointcut is presented addressing this issue. However, in order to ensure broad applicability we restrict our approach to the common standards Java and AspectJ. Thus, we inject parallelism on a lower level and use around-Advice on workload-intensive parts of the loop body (similarly to constraining method execution as described in Section 3.2.5).

Application. An aspect template similar to the one shown in Listing 11 is used to intercept parts of the loop body. For parallel matrix multiplication, we attach `@Task` to methods multiplying matrices with vectors (`Matrix.multiply(Vector)` in Listing 2), thus computing single rows of the result matrix. For method calls which are not executed the around-advice returns a proxy object. A generic implementation is employed for processor mapping (illustrated in Listing 14): every processor executes only every n-th method call (assuming n processors) starting with an offset represented by its index in an ordering over all processors.

```
int index = 0;
Processor mapToProcessor(JoinPoint joinPoint) {
   return processors.get(index++ % processors.size());
}
```

Listing 14. Instantiated Method: Generic Processor Mapping.

As we focus on irregularly distributed workload, we additionally use dynamic load balancing to distribute remaining method calls over idle processors. Here, matrix-vector-multiplications are handled as first-class-objects, implementing the *command pattern* [19].

3.2.7. Summary and Discussion

In the previous sections, we showed how aspect-oriented programming allows us to assemble all components without having to modify them. As our techniques require introspection of components, it is basically a glass-box approach [21]. In this section, we briefly recapitulate

our use of the different aspect-oriented techniques, classifying them into dynamic and static crosscutting [31].

Dynamic Crosscutting. We use *after*-advice to non-intrusively inject internal state monitoring between components (compare Section 3.2.1). This forwarding of control comes always into play, when changes to an object must be passed on to structures dependent on this object. As opposed to after-advice, *around*-advice can be used to intercept and circumvent arbitrary method calls. Section 3.2.5 shows how to employ this construct to non-intrusively redirect the inner control flow of arbitrary components.

To further decouple components, advice is incorporated into bridge templates (see Section 3.2.3). These are implemented using *abstract aspects* and *abstract pointcuts*. This approach decouples components and facilitates connecting varying implementations. By replacing the server-adapter, other platform components can be integrated. Just as new instantiations of the bridge template can integrate crosscutting functionality into different applications.

Static Crosscutting. In Section 3.2.2, we showed how to allow components to manipulate internal state of other components. Here, we used *introductions* to provide for an adapter implementing a callback interface with corresponding get/set methods. As this adapter accesses `private` object state, we use a *privileged aspect*.

With introductions, we are not only able to introduce additional methods or fields, but also to attach *annotations* to methods. In Section 3.2.4, we showed how to tag low-level crosscutting functionality this way. The main advantage of this approach is conciseness, since assembly code is here restricted to annotation introductions. These can be arranged arbitrarily, allowing for more freedom in decomposition and also enhance readability. Especially, this approach turns out to be effective for low-level functionality, where no parameter binding is required and many join points are intercepted.

3.3. Aspect-Oriented vs. Object-Oriented Component Assembly

In this section we compare aspect-oriented and object-oriented approaches to component assembly in a qualitative manner. (A quantitative analysis is presented in Section 4.) In contrast to object-oriented programming, we do not have to impose artificial structures on our design, but can use appropriate and especially designed techniques and tools. We will see that with conventional object-oriented programming, one could at best fall back to the usual design patterns [19], to which these aspect-oriented techniques correspond to, while still having to manipulate and adapt the original code. Prime example for this is the conventional implementation of the observer pattern in comparison to the use of *after*-advice. Before discussing the different assembly issues, we describe how object-oriented programming prevents component immutability in component-based systems.

Component Immutability. Inter-component communication in object-oriented component systems is traditionally restricted to interfaces (or *ports*), that are established when the components were implemented. Unfortunately, this seriously impairs subsequent integration of crosscutting concerns. If the constraint on the immutability of components holds (compare

Section 3.1.3), the number of crosscutting concerns, that can be integrated, is considerably reduced. Basically these are limited to crosscutting concerns anticipated with appropriate *hooks* during the implementation of the component. Thus, to subsequently integrate non-anticipated crosscutting concerns we have to employ a white-box-approach [21] with component mutability.

Monitoring of Internal State. To implement internal state monitoring, one would use the *observer pattern* [19]. This implies the modification of all classes where corresponding events occur, as these must now publish relevant status changes to observers. Besides having to change the original implementation, this can hardly be considered a native approach. In addition, it introduces redundancies, which are known to be error-prone. Furthermore, it reduces code quality as intrusive assembly impairs separation of concerns.

Manipulating Internal State. To allow for the manipulation of internal component state, referenced by fields of constituting classes, one would use the *adapter pattern* [19]. However, if the corresponding fields are declared `private` and no suitable set/get methods are available, there is no possibility to access them without changing the original code. With aspect-oriented programming, however, we can separately specify appropriate adapters in aspects, preserving the original implementation and full reusability of accessed components.

Although, when using a `privileged` aspect, the aspect-oriented approach softens the *information hiding* principle of object-oriented programming, this is crucial, as Java's implementation of this principle is partially incompatible with multi-dimensional decomposition (see discussion on dominant decomposition in Section 2). For a non-intrusive adapter in object-oriented programming one would have to employ *reflection* (e.g. provided by the `java.lang.reflect` package [28]). However, reflection is a low-level technique outside the language scope and impacts performance considerably.

Decoupling Components. To decouple components, we interconnect bridges (based on bridge templates) between components. Applying aspect-orientation enables us to keep the code of all assembled components untouched, even when actual crosscutting is required. All additional functionality can be coherently encapsulated in aspects, defined in their respective compilation units. Although the server-adapter may be used in a pure object-oriented approach as well, installing a bridge between both components would still require intrusive modification of the crosscut code.

Integrating Low-Level Functionality. Pure Java allows also tagging low-level functionality with annotations. However, it does not permit to explicitly state where to insert them. Furthermore, to process this information we would have to use a form of reflection (see above).

Adaptation of Functionality. As mentioned, the concrete deployment scenario sometimes requires adaptation of inner control flow of components. The advantage of employing aspect-oriented programming is that instead of having to change the original implementation of components and probably impairing their usability or efficiency for other deployment scenarios, we can simply plug-in lightweight modifications if needed, while still being able to maintain and advance the original, generic component.

Especially in this case an object-oriented approach would seriously impair the code quality of involved components, probably up to the point where a complete re-design would be favorable over the adaptation of existing components. A common approach in object-orientation is to extract and encapsulate insufficient functionality in strategy-objects (see *strategy pattern* [19]), allowing for their substitution with better suiting ones.

4. Quantitative Analysis

In this section, we compare the discussed aspect-oriented and the object-oriented approach to component assembly quantitatively. We assemble parallel programs consisting of the different sequential applications described in Section 3.1.1 and the platform components described in Section 3.1.3. (As all exemplary applications feature a high degree of irregularity, dynamic load balancing is required in all cases, as well as fault tolerance and termination detection). We measure code quality metrics like cohesion or coupling where appropriate. Additionally, we conduct performance measurements for both approaches.

We ensure comparability by using a direct and bijective mapping of aspect-oriented to object-oriented code, resulting in appropriate assembly issue/design pattern implementations for both cases. Specifically, we convert AspectJ aspects to Java classes by mapping advice to methods, which in turn are called from the join points the associated pointcut intercepts. For example, the object-oriented pendant of *after*-Advice applied to the execution of a certain method (`after() : execution(..) {..}`) consists of a call to the respective method (of the converted aspect) at the bottom of the intercepted method. The same applies for introduced methods.

4.1. Code Quality

In [43], the authors present an assessment framework for comparing aspect- to object-oriented implementations (also compare [23, 32, 20]). Our selection of code quality metrics is based on this framework. In this paper, however, we focus on assembly assuming a pre-existing code base shared by aspect- as well as object-oriented implementations. Thus, in most cases derivatives of the different metrics are more meaningful than the original metrics. For example, we use the difference operator Δ to highlight the influence of the assembly approach on component code quality. In addition, for some of the computed metrics we constrain our analysis on classes constituting the original components. In these cases, the corresponding pure object-oriented code metrics are used. Furthermore, as we study code quality in a component-based approach (assembling application and platform components), we additionally show corresponding metrics with components as the primary unit of decomposition. Here, interdependencies between components are analyzed and not between single compilation units.

Separation of Concerns. Such metrics measure how well code fragments implementing a certain concern are kept together. *Concern Diffusion over Components* (CDC, Table I) counts the number of components contributing to the implementation of a certain concern. A component contributes to the implementation of a concern if its primary purpose is the

implementation of the concern (primary component) or one of its constituting classes accesses a class of a primary component within attribute declarations or method calls.

Besides diffusion over components, we look at *Concern Diffusion over Object Classes* (CDO). Before assembly, CDO for a certain concern equals the number of classes constituting the corresponding component. After assembly, classes from other components may additionally contribute (as defined above) to the concern, thus increasing concern diffusion. Table I shows corresponding difference values ΔCDO. Both metrics (CDC as well as ΔCDO) show how object-oriented programming inevitably intermingles different concerns (tangling) and splits up code sequences essentially belonging together (scattering). With aspect-oriented programming, we can prevent concern diffusion and preserve separation of concerns.

In addition, we look at how concern fragments disperse over all components as an indication for code locality. Table II shows *Concern Dispersion over Components* (measured in lines of code) as an average of all exemplary applications. Again, this demonstrates the unfavorable dispersion of concern code resulting from intrusive assembly.

Coupling. Coupling reflects how classes (or components) depend on each other, especially it shows whether changes in one have effect on the implementation of others. *Coupling Between Object Classes* (CBO) [9] for example counts the number of classes a certain class is coupled to. A class is coupled to another class, if it calls a method of this class or instantiates the class. Table I shows corresponding difference values. These are computed over average coupling values—denoted with $\langle$CBO$\rangle$—for the classes constituting the original component. Averages are computed by weighing classes with lines of code.

Coupling Between Components (CBC, Table I) applies the same principle to components. A component is coupled to another component if one of its constituting classes is coupled to a class from this other component. Figures for CBC as well as for $\Delta\langle$CBO$\rangle$ show how intrusive assembly induces unnecessary coupling between involved components and compilation dependencies specific for the actual deployment scenario.

Cohesion. Cohesion looks at how different elements of a class relate to each other. If cohesion is low, the respective class should probably be refactored into multiple ones. Metrics like *Lack of Cohesion of Methods* (LCOM) [9] or its derivate LCOM* (for details see [27]) count methods operating on disjunctive data sets. Table I shows $\Delta\langle$LCOM$^*\rangle$ for the original component implementation—again applying the difference operator on weighed averages. This demonstrates how in most cases intrusive assembly reduces class cohesion, impairing code quality of the corresponding components.

Conciseness. To study the effects of assembly on the amount of generated code, we compute a size metric as an indication for conciseness—denoted as ΔLOC/LOC in Table I. This metric measures how much the code base increases with assembly. Primarily, we are interested in whether one approach requires significantly more lines of code to assemble an equivalent program than the other. Figures show that even though aspect-oriented programming sometimes requires additional code to state pointcuts or to declare aspects it does not necessarily reduce conciseness.

Table I. Code Quality Metrics.

		Core Concern		Load Balancing		Fault Tolerance		Termination Detection	
Metric	**Appl.**	OOP	AOP	OOP	AOP	OOP	AOP	OOP	AOP
CDC	MM	1.00	1.00	3.00	1.00	2.00	1.00	3.00	1.00
	QS	1.00	1.00	3.00	1.00	2.00	1.00	3.00	1.00
	TSP	1.00	1.00	3.00	1.00	2.00	1.00	4.00	1.00
ΔCDO	MM	0.00	0.00	2.00	0.00	2.00	0.00	3.00	0.00
	QS	0.00	0.00	2.00	0.00	2.00	0.00	3.00	0.00
	TSP	0.00	0.00	4.00	0.00	1.00	0.00	3.00	0.00
CBC	MM	1.00	0.00	2.00	0.00	2.00	0.00	0.00	0.00
	QS	1.00	0.00	2.00	0.00	2.00	0.00	0.00	0.00
	TSP	2.00	0.00	2.00	0.00	2.00	0.00	0.00	0.00
Δ⟨CBO⟩	MM	0.83	0.00	1.43	0.00	1.69	0.00	0.00	0.00
	QS	0.20	0.00	1.43	0.00	1.69	0.00	0.00	0.00
	TSP	0.88	0.00	0.97	0.00	1.69	0.00	0.00	0.00
Δ⟨LCOM*⟩	MM	0.09	0.00	0.10	0.00	-0.02	0.00	0.00	0.00
	QS	0.04	0.00	0.10	0.00	-0.02	0.00	0.00	0.00
	TSP	0.07	0.00	0.05	0.00	-0.02	0.00	0.00	0.00
ΔLOC/LOC	MM	9.13	0.00	29.30	41.39	44.66	42.72	44.04	40.37
[%]	QS	2.19	0.00	31.87	39.93	44.66	42.72	44.04	40.37
	TSP	14.91	0.00	13.19	21.25	44.66	42.72	41.28	38.53
RCO	MM	86.12	100.00	49.45	100.00	43.69	100.00	100.00	100.00
[%]	QS	65.71	100.00	49.45	100.00	43.69	100.00	100.00	100.00
	TSP	71.43	100.00	58.61	100.00	43.69	100.00	100.00	100.00

MM = Matrix Multiplication, QS = Quicksort, TSP = Traveling Salesperson Problem

Table II. Concern Dispersion over Components.

	Application Component		Load Balancing		Fault Tolerance		Termination Detection	
Concern	OOP	AOP	OOP	AOP	OOP	AOP	OOP	AOP
Core concern [%]	100.00	100.00	0.00	0.00	0.00	0.00	0.00	0.00
Load Balancing [%]	7.03	0.00	90.16	100.00	2.81	0.00	0.00	0.00
Fault Tolerance [%]	0.00	0.00	6.36	0.00	93.64	100.00	0.00	0.00
Termination Detection [%]	2.53	0.00	10.99	0.00	3.06	0.00	83.42	100.00

.

Table III. Parallel Performance.

	t_s [s]	t_p, $n = 16$ [s]		**Speedup**		**Efficiency** [%]	
Application		OOP	AOP	OOP	AOP	OOP	AOP
Matrix Multiplication	4524	291	290	15.57	15.59	97.33	97.41
Quicksort	4450	320	318	13.90	13.98	86.86	87.41
State Space Search	4065	302	312	13.47	13.02	84.18	81.40

Reusability. As an index for reusability, we look at the percentage of reusable compilation units (weighted with lines of code). Classes constituting components are only regarded reusable, if they have not been modified during assembly. Table I shows figures for *Reusability of Component Object Classes* (RCO). Especially this metric shows how crucial the assembly paradigm is—as with object-oriented programming only a fraction of the original component code base remains usable outside of the current assembly scenario.

4.2. Performance

Previously, we showed how code quality benefits from our aspect-oriented approach. However, in parallel computing one must also take parallel performance and efficiency into account. Thus, we additionally compare both approaches with regard to their parallel performance.

For our matrix multiplication example, we multiply randomly created sparse matrices. As we apply a non-uniform distribution of non-zero elements, single result elements feature irregular computational effort. Thus, dynamic load balancing becomes favorable. Our quicksort procedure sorts large arrays of randomly created individuals in an evolutionary algorithm [2]. These represent strategies in an iterated prisoner's dilemma [4]. Two individuals are compared by playing the game repeatedly against each other. This application of quicksort features irregular distributed computational effort comparable to the other applications—without having to use an external sort (in contrast to simply comparing real numbers). We are thus able to concentrate on recursive parallelism. For our state space search example, we apply branch-and-bound on randomly created traveling salesperson problem instances. Bounds are communicated using the distribution component.

We conduct runtime measurements on a cluster computer with 16 Intel Xeon Dual 2,6 GHz 2 GB RAM, connected by a 1000 Mbps Ethernet network. Table III displays sequential and parallel run-times as wells as the resulting speedup and efficiency values for our three example applications. The results show on the one hand that the employed parallelization methods are able to achieve good speedups regardless which assembly approach is employed. On the other hand, we see that differences in parallel performance can be neglected — in contrast to code quality, which significantly improves with aspect-oriented programming.

5. Related Work

Several research efforts employ aspect-oriented programming to deal with parallelization issues. [6, 48, 5, 7]. (In [7] aspect-orientation is used alongside with invasive software composition [3]). In all approaches, crosscutting concerns (e.g. load balancing [6] or problem decomposition [48]) are implemented as straight-forward AspectJ-aspects from scratch. Aspect-oriented techniques have also been applied to address parallelization or concurrency on a lower level. In [13] for example, concurrency patterns and low-level concerns like synchronization issues are discussed.

Primarily, all these approaches aim at designing (reusable) aspect libraries. Although this is a worthwhile and proven approach in aspect-oriented programming, our work especially considers the fact that most often there already exist highly optimized sequential application code and conventionally implemented components of a parallel platform addressing all functional and non-functional requirements of parallelization.

In [25], the authors mine scientific code (*Java Grande Forum* benchmark suite [46, 47]) for join points qualifying to inject parallelization. Where not available, they show how to refactor the code to provide suitable join points. This mostly means introducing object-oriented design into legacy imperative programs ported to Java (typical for this application suite), thus increasing the applicability of AspectJ. In contrast to their research, we apply aspect-oriented programming not only to connect application and platform components, but in general between components in a component-based approach. In later research [26], a pointcut-model for loops is presented. Especially this could prove a highly useful addition to our approach as `for`-loops promise most intuitive parallelism (compare Section 3.2.6).

In Section 3.2, we enumerate recurring patterns in component assembly. For didactic purposes and to be able to convert the approach between aspect-oriented programming languages, these are mapped to corresponding Gang-of-Four (GoF) [19] design patterns. In [24], the authors compare object- and aspect-oriented implementations of all GoF design patterns. This qualitative analysis is complemented with a quantitative study in [20]. Additionally, we show how to use these constructs to connect components in a component-based approach. Thus, we connect software elements on a larger scale: from an object-oriented to a component-based point of view, but with similar premises.

Component-Based Software Engineering. In application server research, application integration and assembly is of primary interest. Here, mostly business-oriented services and related concerns are addressed. Important trends are the incorporation of aspect-orientation for low-level concerns like persistence or security (like in Spring [50] or JBoss AOP [29]) or the use of annotations to add metadata, which is processed during deployment, like in the EJB3 specification [17]. Here, parallelization is of less importance, because load is generated by large numbers of independent clients and its balancing is mainly a question of replicating infrastructure.

Besides primarily business-related approaches, component-based software engineering in general starts to reckon aspect-oriented programming (for example [37, 39, 52, 16, 10, 44]). Most research derives from the question how to align the black box property desired in component-based software engineering with the inherent introspective nature of aspect-oriented programming [39]. Models range from very restrictive approaches like JAsCo [52],

where only public methods and events of Java Beans can be intercepted, to more open approaches like JBoss AOP.

In general, sophisticated component models are presented incorporating aspect-oriented programming to address typical crosscutting concerns. We study the component-based approach outside of the scope of traditional component-based software engineering, neither imposing component model nor composition standard on assembly. Instead, we simply assume non-modifiable, reusable software artifacts whose implementation lies outside the scope of the current assembly process. Primary quality of assembled artifacts is full reusability independent of component model. Our focus lies on how this approach benefits separation of concerns in parallelization and allows circumventing the negative effects of intrusive assembly on code quality.

Meta-Programming. In cases where the range of aspect-oriented programming is too restricted to handle a given problem in a consistent manner more generic approaches like reflective program-transformation techniques (e.g. [8]) or *meta-programming* turned out to be effective. In [38], template-based meta-programming is used to configure middleware components based on annotations. During compilation, template processors read out these annotations and modify the source code accordingly. Although, this work focuses on the design of application servers, the approach taken could in principle also be used to inject parallelization on the application level.

However, such aggressive transformation techniques also greatly reduce type safety and implementation can thus become very error-prone. Moreover, the shift from classical object-orientation to meta-programming is even more pronounced than to the proposed combination of object-orientation and aspect-orientation, affecting issues like tool support and developer training. Besides that, the approach of meta-programming seems to be highly interesting. Especially for the task of behavior modification (see Section 3.2.5) this technique could be a useful addition to the approach discussed in our paper.

Generative Programming. Generative programming tackles problems induced by dominant decomposition (see Section 2) by introducing a higher level of abstraction into the development process. On this level, recurring code fragments are defined and distributed over generated programs. In parallelization, examples are generative parallel design patterns [53, 1] or automatically generated parallel program skeletons [18, 49]. In [49], program skeletons are generated for different parallel deployment scenarios, e.g. multi-core-systems or computational grids. In [1], parallel structural code is generated from parameterizable pattern templates to tackle the *Cowichan* problems [55]. In both cases, generated program templates are instantiated by filling *hooks* (like abstract methods) with domain specific code.

If we assume a preexisting sequential code base and code quality is considered, one disadvantage of the approach is, that the respective code has to be disassembled and fitted into hook methods in-between *protected regions*. This violates component immutability and reduces code quality (for example separation of concerns or cohesion). This deficiency can be overcome when restricting generative programming to the parameterizable generation of object-oriented components (as a legitimate alternative to the non-intrusive strategy pattern described in Section 3.2.5).

6. Conclusion

Although aspect-orientation offers only a restricted set of constructs (compared with the possibilities of generative or meta-programming), for the problem at hand, they proved to be sufficient. They allow us to integrate existing application code with different crosscutting concerns arising from parallelization, while leaving original sequential application code as well as platform components untouched.

By combining elements from component-based software engineering and aspect-oriented programming, we are able to address crosscutting concerns in a fully non-intrusive manner. Components require neither infrastructure nor hooks for anticipated crosscutting functionality. Our approach to component assembly ensures full reusability of all assembled software artifacts. Code quality properties like separation of concerns, cohesion, and coupling significantly improve with our approach. Additionally, these improvements do not come at the cost of reduced performance.

The techniques discussed can, in principle, be employed for almost all assembly and integration purposes. They allow to access and to replace arbitrary parts of existing code non-intrusively in order to make necessary adaptations for assembly with existing components. However, in the extreme case significant parts of the new program logic become part of assembly code, complicating software re-use. Thus, in the case of parallelization, a necessary precondition for sensibly applying our approach is that the existing sequential code is amenable for parallelization and that the employed platform components basically provide the required functionality.

7. Acknowledgement

Wolfgang Blochinger has been supported in part by the Ohio Supercomputer Center.

REFERENCES

1. Anvik J, Schaeffer J, Szafron D, Tan K. Asserting the utility of CO2P3S using the Cowichan Problem Set. *Journal of Parallel and Distributed Computing* 2005; **65**(12):1542–1557.
2. Ashlock D. *Evolutionary Computation for Modeling and Optimization.* Springer, 2006.
3. Aßmann U. *Invasive Software Composition.* Springer, 2003.
4. Axelrod R. The Evolution of Strategies in the Iterated Prisoner's Dilemma. In *Genetic Algorithms and Simulated Annealing.* Morgan Kaufman: Los Altos, CA, 1987.
5. Bangalore PV. Generating Parallel Applications for Distributed Memory Systems using Aspects, Components, and Patterns. In *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software.* ACM: New York, 2007.
6. Blochinger W, Dangelmayr C, Schulz S. Aspect-Oriented Parallel Discrete Optimization on the Cohesion Desktop Grid Platform. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid.* IEEE: Washington, DC, 2006.
7. Chalabine M, Kessler C. Crosscutting Concerns in Parallelization by Invasive Software Composition and Aspect Weaving. In *Proceedings of the 39th Hawaii International Conference on System Sciences.* IEEE: Washington, DC, 2006.
8. Chiba S. Load-Time Structural Reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming.* Springer: London, 2000.

9. Chidamber SR, Kemerer CF. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 1994; **20**(6):476–493.
10. Clemente PJ, Hernandez J, Sanchez F. Driving Component Composition from Early Stages Using Aspect-Oriented Techniques. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences.* IEEE: Washington, DC, 2007.
11. Constantinides CA, Bader A, Elrad TH, Fayad ME, Netinant P. Designing an Aspect-Oriented Framework in an Object-Oriented Environment. *ACM Computing Surveys* 2000; **32**(1):41.
12. Councill WT, Heineman GT. *Component-Based Software Engineering.* Addison-Wesley, 2001.
13. Cunha C, Sobral J, Monteiro M. Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development.* ACM: New York, 2006.
14. Dijkstra EW. *A Discipline of Programming.* Prentice-Hall, 1976.
15. Lämmel R, De Schutter K. What does Aspect-Oriented Programming mean to Cobol? In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development.* ACM: New York, 2005.
16. Lagaisse B, Joosen W. Component-Based Open Middleware Supporting Aspect-Oriented Software Composition. In *Component-Based Software Engineering.* Springer: Berlin, 2005.
17. JSR-000220 Enterprise JavaBeans 3.0. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html [last accessed Aug 8, 2007]
18. Ferreira JF, Sobral JL, Proenca AJ. JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid.* IEEE: Washington, DC, 2006.
19. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
20. Garcia A, Sant'Anna C, Figueiredo E, Kulesza U, Lucena C, von Staa A. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development.* ACM: New York, 2005.
21. Gill NS. Reusability Issues in Component-based Development. *ACM SIGSOFT Software Engineering Notes* 2003; **28**(6):30.
22. Grama A, Gupta A, Karypis G, Kumar V. *Introduction to Parallel Computing* (2nd edn). Addison-Wesley, 2003.
23. Greenwood P, Garcia A, Rashid A, Figueiredo E, Sant'Anna C, Cacho N, Sampaio A, Soares S, Borba P, Dosea M, Ramos R, Kulesza U, Bartolomei T, Pinto M, Fuentes L, Gamez N, Moreira A, Araujo J, Batista T, Medeiros A, Dantas F, Fernandes L, Wloka J, Chavez C, France R, Brito I. On the Contributions of an End-to-End AOSD Testbed. In *EARLYASPECTS '07: Proceedings of the Early Aspects at ICSE.* IEEE: Washington, DC, 2007.
24. Hannemann J, Kiczales G. Design Pattern Implementation in Java and AspectJ. *ACM SIGPLAN Notices* 2002; **37**(11):161–173.
25. Harbulot B, Gurd JR. Using AspectJ to Separate Concerns in Parallel Scientific Java Code. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development.* ACM: New York, 2004.
26. Harbulot B, Gurd JR. A Join Point for Loops in AspectJ. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development.* ACM: New York, 2006.
27. Henderson-Sellers B. *Object-Oriented Metrics: Measures of Complexity.* Prentice-Hall, 1996
28. Java Reflection API: java.lang.reflect (Java Platform SE 6). http://java.sun.com/javase/6/docs/api/java/lang/reflect/package-summary.html [last accessed Sep 23, 2007]
29. JBoss AOP. http://labs.jboss.com/jbossaop [last accessed Aug 8, 2007]
30. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold W. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming.* Springer: London, UK, 2001.
31. Kiczales G, Lamping J, Mendhekar A, Maeda C, Videira Lopes C, Loingtier JM, Irwin J. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming.* Springer: Berlin, 1997.
32. Kulesza U, Sant'Anna C, Garcia A, Coelho R, von Staa A, Lucena C. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance.* IEEE: Washington, DC, 2006.
33. Laddad R. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning, 2003.

34. Lieberherr K, Orleans D, Ovlinger J. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM* 2001; **44**(10):39–41.
35. McMenamin SM, Palmer JF. *Essential Systems Analysis.* Yourdon, 1984.
36. Neary M, Cappello P. Advanced Eager Scheduling for Java-based Adaptively Parallel Computing. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande.* ACM: New York, 2002.
37. Pawlak R, Seinturier L, Duchien L, Florin G, Legond-Aubry F, Martelli L. JAC: An Aspect-Based Distributed Dynamic Framework. *Software—Practice and Experience* 2004; **34**(12):1119–1148.
38. Pawlak R. Spoon: Compile-time Annotation Processing for Middleware. In *IEEE Distributed Systems Online* 2006; **7**(11).
39. Pessemier N, Seinturier L, Coupaye T, Duchien L. A Safe Aspect-Oriented Programming Support for Component-Oriented Programming. In *Proceedings of the 11th International ECOOP Workshop on Component-Oriented Programming.* Karlsruhe University, 2006.
40. Rashid A, Sawyer P, Moreira A, Araujo J. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering.* IEEE: Washington, DC, 2002.
41. Rashid A, Chitchyan R. Persistence as an Aspect. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development.* ACM: New York, 2003.
42. Sakurai K, Masuhara H, Ubayashi N, Matsuura S, Komiya S. Association Aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development.* ACM: New York, 2004.
43. Sant'Anna C, Garcia A, Chavez C, Lucena C, von Staa A. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *Proceedings of the 17th Brazilian Symposium on Software Engineering.* Manaus, Brazil, 2003.
44. Seinturier L, Pessemier N, Duchien L, Coupaye T. A Component Model Engineered with Components and Aspects. In *Component-Based Software Engineering.* Springer: Berlin, 2006.
45. Shah V. Using Aspect-Oriented Programming for Addressing Security Concerns. In *Proceedings of the 13th International Symposium on Software Reliability Engineering.* IEEE: Washington, DC, 2002.
46. Smith LA, Bull JM. A Multithreaded Java Grande Benchmark Suite. In *Proceedings of the 3rd Workshop on Java for High Performance Computing.* ACM: New York, 2001.
47. Smith LA, Bull JM, Obdrzalek J. A Parallel Java Grande Benchmark Suite. In *Proceedings of ACM/IEEE 2001 Conference on Supercomputing.* ACM: New York, 2001.
48. Sobral JL. Incrementally Developing Parallel Applications with AspectJ. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium.* IEEE: Washington, DC, 2006.
49. Sobral JL, Proena AJ. Enabling JaSkel Skeletons for Clusters and Computational Grids. In *IEEE Cluster (Cluster 2007).* IEEE: Washington, DC, 2007.
50. Spring AOP: Aspect Oriented Programming with Spring. http://www.springframework.org/docs/reference/aop.html [last accessed Aug 8, 2007]
51. Sommerville I, Sawyer P. *Requirements Engineering: A Good Practice Guide.* Wiley, 1997.
52. Suvée D, Vanderperren W, Jonckers V. JAsCo: An Aspect-Oriented Approach tailored for Component Based Software Development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development.* ACM: New York, 2003.
53. Tan K, Szafron D, Schaeffer J, Anvik J, MacDonald S. Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM: New York, 2003.
54. Tarr PL, Ossher H, Harrison WH, Sutton Jr SM. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering.* IEEE: Los Alamitos, CA, 1999.
55. Wilson GV. Assessing the Usability of Parallel Programming Systems: The Cowichan Problems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems.* Birkhäuser: Basel, 1994.