Coordination Patterns for Parallel Computing

Bernd Freisleben and Thilo Kielmann

Dept. of Electrical Engineering and Computer Science, University of Siegen
Hölderlinstr. 3, D–57068 Siegen, Germany
{freisleb|kielmann}@informatik.uni-siegen.de

Abstract. The aim of this paper is to promote the idea of developing
reusable *coordination patterns* for parallel computing, i.e. customizable
components from which parallel applications can be built by software
composition. To illustrate the idea, a fundamental *manager/worker* coor-
dination pattern useful for programming a variety of parallel applications
is presented.

1 Introduction

Although coordination models based on generative and anonymous communi-
cation allow to express complex process interactions in a straightforward man-
ner, their programming interface is often felt to be rather low level. Therefore,
higher–level abstractions on top of the basic communication operations would
significantly ease concurrent program development. Instead of reinventing the
wheel each time new concurrent programs have to be written, such *reusable co-
ordination patterns* should provide basic abstractions common to frequently used
settings in which concurrent processes have to interact in a coordinated manner.
Developing concurrent programs from these reusable basic building blocks would
then simply require to parameterize the behaviour of the patterns to the needs
of the given problem and to compose the concurrent application out of these
patterns. This approach follows the idea of *software composition* [4], i.e. produc-
ing new software by composing it from already existing components which can
simply be "plugged together". This paper is intended to initiate the discussion
and collection of suitable coordination patterns (in the sense of *design patterns*
as initially introduced by Gamma et al. [2]) which may be reused in various
areas of parallel programming. In order to illustrate the idea, we present a fun-
damental parallel computing paradigm useful in a variety of situations, namely
a *manager/worker* pattern.

2 The Manager and Worker Patterns

The intent of these patterns is to decouple coordination–level issues such as
task assignment strategies and worker termination from application–level issues
such as task creation, task computation, and result combination. The Manager
pattern is responsible for providing and assigning task units and for collecting
results. The Worker pattern is responsible for acquiring and executing task units
and for transmitting computed results to the manager.

2.1 Motivation

It is a very common situation in parallel programming to employ a specific manager process to divide a given problem into smaller tasks and to distribute these tasks among several worker processes. While workers repeatedly process such tasks and return the computed results to the manager process, the managerial task is much more complex. The manager not only has to operate on the application level by providing task units and later combining the received results to the overall result of the application. It also has to perform coordination–level tasks, such as assigning tasks to workers and terminating workers.

Although both levels of managerial tasks are independent of each other, they are typically intermixed in existing applications. They are hardly made explicit, but instead implicitly performed by the communication operations of manager and workers. Therefore, it is our motivation to provide clearly defined abstractions for both levels in the form of coordination patterns suitable for building reusable coordination components.

2.2 Structure and Participants

The structure of the participants in the Manager and Worker patterns are illustrated by the Booch diagram shown on the right. The figure is divided into two horizontal layers, one for coordination aspects and one for application aspects. Furthermore, the diagram is vertically divided into the manager role, the worker role, and the data exchanged between both, namely objects and object spaces.
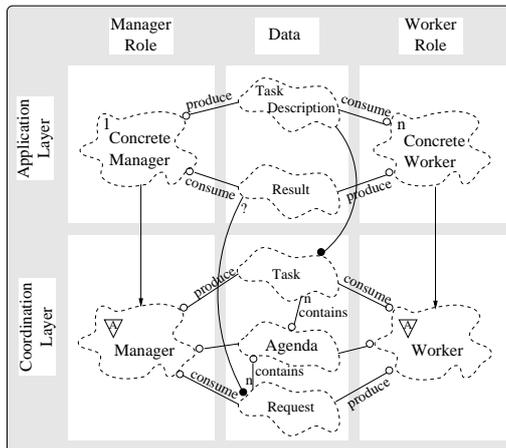
The Agenda is the central shared data structure (i.e. the object space) through which the abstract components in the coordination layer, Manager and Workers, communicate. The Task and Request objects exchanged in this layer are primarily used as containers for their application–specific counterparts, Task Description and Result, which are transparently transported via the coordination layer. Concrete Manager and Concrete Worker are instantiated by inheriting from Manager and Worker, respectively, while providing suitable implementations for their application–specific, abstract methods.

2.3 Implementation and Sample Usage

In the following, an implementation of Manager and Worker components implemented in the C++ language and using the Objective Linda coordination

model [3] will be sketched. The presentation is restricted to the evaluate routines. These define the behaviour of the components once they are activated.

```
public: void Manager::evaluate(void){
 OIL_OBJECT *task_descriptor;  Request *r;  Task *t;  int workers = 0;
 bool still_work_to_do = true; agenda = new OBJECT_SPACE;
 context->out(*new OS_LOGICAL(agenda_id(),agenda));  setup_agenda();
 while ( still_work_to_do || (workers > 0) ){
  r = agenda->in(*new Request);
  switch (r->get_tag()){
   case JOIN_GROUP: workers++;  break;
   case LEAVE_GROUP: workers--;  break;
   case RESULT: still_work_to_do = process_result(r->get_result()); break;
   case TASK_REQUEST: still_work_to_do = process_result(r->get_result());
             task_descriptor = get_next_task(); t = new Task(r->get_id());
             if ( still_work_to_do )  t->assign_task(task_descriptor);
             else  t->set_stop_task(true);
             agenda->out(*t);  break;
} delete r; } }
```

```
public: void Worker::evaluate(void){
 OIL_OBJECT *result;  bool running;  Task *t;
 agenda = context.attach(new OS_LOGICAL(agenda_id()));
 if (do_register() || use_handshake()) // register with manager
  agenda->out(*new Request(JOIN_GROUP));
 result = NULL;  running = accept_tasks();
 while (running){
  if (use_handshake()){ // request new task
   agenda->out(*new Request(TASK_REQUEST,my_worker_id(),result));
   t = agenda->in(*new Task(my_worker_id()));
   if (t->is_stop_task())  running = false;
   else {  result = do_the_work(t->get_task());
           running = accept_tasks();  delete t; } }
  else { t = new Task(my_worker_id()); t->match_valid_tasks_only(true);
         t = agenda->in(*t,0);
         if ( t == NULL )  running = false;
         else { agenda->out(*new Request(RESULT,my_worker_id(),
                 do_the_work(t->get_task()); running = accept_tasks();
                 delete t; } }
  if (do_register()||use_handshake())
     agenda->out(*new Request(LEAVE_GROUP)); } }
```

To illustrate the sample usage of the manager and worker components, a sophisticated technique taken from a raytracing application, adaptive scheduling [1] will be presented, where the concrete Adaptive_Manager performs adaptive load distribution by successive reduction of task sizes and hence by assignment of larger tasks to faster workers. The implementations of Adaptive_Manager and Adaptive_Worker show how the concrete instances parameterize the components' coordination protocol and implement the application–specific parts.

```
class Adaptive_Manager : public Manager{
private: int lines, lines_received, next_line_out;
protected: virtual OIL_OBJECT *get_next_task(void){  int size;
  range *result;  if ( next_line_out >= lines ) return NULL;
  else { size = next_size(); // compute the current task size
         result = new range(next_line_out,next_line_out+size-1);
         next_line_out += size; return result; } }
virtual bool process_result(OIL_OBJECT* imagelines) {
  // store contents of imagelines to file
  lines_received += (imagelines->end - imagelines->start + 1);
  return lines_received == lines; }
public: Adaptive_Manager (int size_of_image)
 { lines = size_of_image; lines_received = 0; next_line_out = 0; } };
```

```
class Adaptive_Worker : public Worker{  protected:
virtual bool do_register(void) { return true; }
virtual bool use_handshake(void) { return true; }
virtual char *my_worker_id(void) { return new uuid; }
virtual OIL_OBJECT *do_the_work(OIL_OBJECT* my_range)
  { return imagelines(my_range); }// compute imagelines from line range
};
```

3   Conclusions

The major benefit of employing the manager and worker patterns is the decoupling of the coordination layer from the application layer. Hence, application programs only need to provide application–specific code whereas all program code related to coordination between the manager and its workers can easily be "plugged in". This alleviates application programmers from coordination aspects, allowing them to focus on the application itself.

The decoupling of the application code from the control flow also enforces a "framework–like" structuring of the manager implementation. Typically, the manager implements the control flow of the application as a whole. But because the patterns take over this part, the application–specific methods of a concrete manager simply have to react whenever they are called. Therefore, the concrete manager's only duty is to produce one task after the other and to process incoming results when it is requested to do so.

References

1. B. Freisleben, D. Hartmann, and T. Kielmann. Parallel Raytracing: A Case Study on Partitioning and Scheduling on Workstation Clusters. In *Proc. of the 30th Ann. Hawaii Int. Conf. on System Sciences*, Vol. 1, pp. 596–605, IEEE Press, 1997.
2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object–Oriented Software*. Addison Wesley, 1994.
3. T. Kielmann. Designing a Coordination Model for Open Systems. In *Coordination Languages and Models*, LNCS 1061, pp. 267–284, Cesena, Italy, Springer, 1996.
4. O. Nierstrasz and T.D. Meijler. Research Directions in Software Composition. *ACM Computing Surveys*, 27(2):262–264, 1995.