The Manager Workers Pattern

An Activity Parallelism Architectural Pattern for Parallel Programming

Jorge L. Ortega-Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM
jloa@fciencias.unam.mx

Abstract

The Manager-Workers pattern is an architectural pattern for parallel programming, used when a design problem can be understood in terms of activity parallelism. This pattern proposes a solution in which the same operations are performed simultaneously and independently on different pieces of data. Operations carried out by each component are independent of operations by other components.

1. Introduction

Parallel processing is the division of a problem, presented as a data structure or a set of actions, among multiple processing components that operate simultaneously. The expected result is a more efficient completion of the solution to the problem. The main advantage of parallel processing is its ability to handle tasks of a scale that would be unrealistic or not cost-effective for other systems [CG88, Fos94, ST96, Pan96]. The power of parallelism centres on partitioning a big problem in order to deal with complexity. Partitioning is necessary to divide such a big problem into smaller sub-problems that are more easily understood, and may be worked on separately, on a more "comfortable" level. Partitioning is especially important for parallel processing, because it enables software components to be not only created separately, but also executed simultaneously.

Requirements of order of data and operations dictate the way in which a parallel computation has to be performed, and therefore, impact on the software design [OR98]. Depending on how the order of data and operations are present in the problem description, it is possible to consider that most parallel applications fall into one of three forms of parallelism: *functional parallelism*, *domain parallelism*, and *activity parallelism* [OR98].

2. The Manager-Workers Pattern

The Manager-Workers pattern is a variant of the Master-Slave pattern [POSA96] *for parallel systems, considering an activity parallelism approach where the same operations are performed on ordered data. The variation is based on the fact that components of this pattern are proactive rather than reactive* [CT92]. *Each processing component simultaneously performs the same operations, independent of the processing activity of other components. An important feature is to preserve the order of data* [OR98].

Example

Consider the Polygon Overlay problem: the objective is to obtain the overlay of two rectangular maps, *A* and *B*, each one covering the same area that is decomposed into a set of non-overlapping rectangular polygons [WL96, WRMPD95]. This kind of problem frequently arises in geographical information systems, in which the first map might represent, for example, soil type, and the second, vegetation. Their overlay shows how

combinations of soil type and vegetation are distributed. So, overlaying both maps create a new map, consisting of the non-empty polygons in the geometric intersection of *A* and *B*.

In order to simplify this problem, it is required that all polygons be non-empty rectangles, with vertices on a rectangular integer grid [0...N]×[0...M] (Figure 1). It is also required that input maps have identical extents, that each be completely covered by its rectangular decomposition, and that the data structures representing the maps be small enough to fit into physical memory. It is not required that the output map to be sorted, although all of the input maps used in this example are usually sorted by lower-left corner [WL96, WRMPD95].

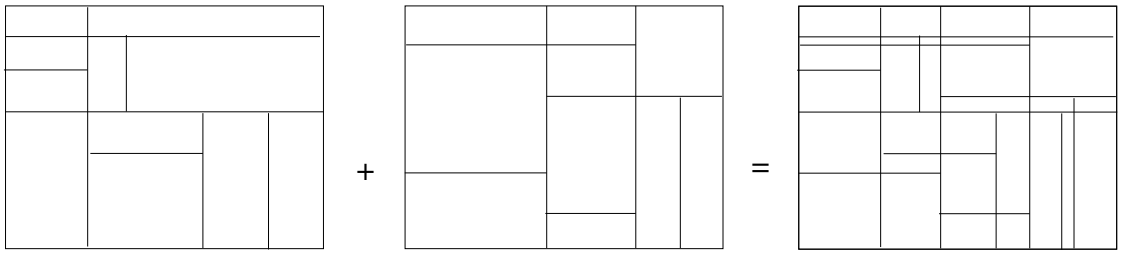

Figure 1. The Polygon Overlay problem for two maps, *A* and *B*.

Normally, the sequential solution goes through each all the polygons belonging to *A*, and for each one of them, finds all the intersections with any polygon in *B*. This is an effective solution, although it is a rather slow one. However, since the overlay of a couple of polygons can be performed independently of the overlay of other polygons, it is possible to take advantage on simultaneously obtaining intersections.

Context

Start the design of a parallel program, using a particular programming language for certain parallel hardware. Consider the following context constraints:

- The problem involves tasks of a scale that would be unrealistic or not cost-effective for other systems to handle and lends itself to be solved using parallelism.
- The parallel hardware platform or machine to be used is given.
- The main objective is to execute the tasks in the most time-efficient way.

Problem

The same operation is required to be repeatedly performed on all the elements of some ordered data. Data can be operated without a specific order. However, an important feature is to preserve the order of data. If the operation is carried out serially, it should be executed as a sequence of serial jobs, applying the same operation to each datum one after another. Generally, performance as execution time is the feature of interest, so the goal is to take advantage of the potential simultaneity in order to carry out the whole computation as efficiently as possible.

Forces

The following forces should be considered:

- Preserve the order of data. However, the specific order of operation on each piece of data is not fixed.
- The operation can to be performed independently on other different pieces of data.
- Data pieces may exhibit different sizes. This means that the independent computations on the pieces of data should adapt to the data size to be processed, in order to obtain automatic load-balancing.
- The solution has to scale over the number of processing elements. Changes in the number of processing elements should be reflected by the execution time.
- The coordination of the independent computations has to take up a limited amount of time in order not to impede performance of the processing elements.
- Mapping the processing elements to processors has to take into account the interconnection among the processors of the hardware platform.

Solution

Introduce activity parallelism by having multiple data sets processed at the same time. The most flexible representation of this is the Manager-Workers pattern approach. This structure is composed of a manager component and a group of identical worker components. The manager is responsible of preserving the order of data. On the other hand, each worker is capable of performing the same independent computation on different pieces of data. It repeatedly seeks a task to perform, performs it and repeats; when no tasks remain, the program is finished. The execution model is the same, independent of the number of workers (at least one). If tasks are distributed at run time, the structure is naturally load balanced: while a worker is busy with a heavy task, another may perform several shorter tasks. This distribution of tasks at runtime copes with the fact that data pieces may exhibit different size. To preserve data integrity, the manager program takes care of what part of the data has been operated on, and what remains to be computed by the workers [POSA96, CG88, Pan96, CT92]. Also, the manager component could optionally be an active component, in order to deal with data partitioning and gathering, so such tasks can be done concurrently while receiving data request from the workers. Hence, manager operations need capabilities for synchronisation and blocking. Moreover, the manager could be also responsible for the hardware mapping as well, in addition to starting the appropriate number of workers. Mapping requires experiments at execution time and experience, but performing the mapping (according to a pre-determined policy) can be considered as another responsibility of the manager.

Structure

The Manager-Workers pattern is represented as a manager, preserving the order of data and controlling a group of processing elements or workers. Usually, only one manager and several identical worker components simultaneously exist and process during the execution time. In this architectural pattern, the same operation is simultaneously applied in effect to different pieces of data by worker components. Conceptually, workers have access to different pieces of data. Operations in each worker component are independent of operations in other components. The structure of the solution involves a central manager that distributes data among workers by request. Therefore, the solution is presented as a centralised network, the manager being the central common component. An Object

Diagram, representing the network of elements that follows the Manager-Workers structure is shown in Figure 2.

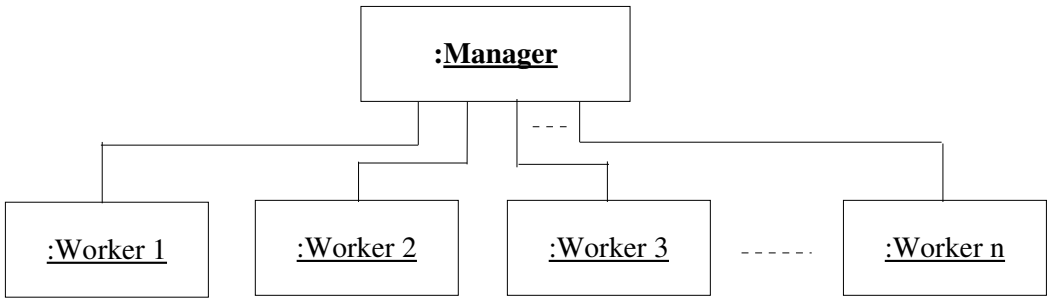

Figure 2. Object Diagram of the Manager-Workers pattern.

Participants

- **Manager.** The responsibilities of a manager are to create a number of workers, to partition work among them, to start up their execution, and to compute the overall result from the sub-results from the workers.
- **Worker.** The responsibility of a worker is to seek for a task, to implement the computation in the form of a set of operations required, and to perform the computation.

Dynamics

A typical scenario to describe the run-time behaviour of the Manager-Worker pattern is presented, where all participants are simultaneously active. Every worker performs the same operation on its available piece of data. As soon as it finishes processing, it returns a result to the manager, requiring more data. Communications are restricted between the manager and each worker. No communication between workers is allowed (figure 3).

In this scenario, the steps to perform a set of computations is as follows:

- All participants are created, and wait until a computation is required to the manager. When data is available to the manager, this divides it, sending data pieces by request to each waiting worker.
- Each worker receives the data and starts processing an operation *Op.* on it. This operation is independent of the operations on other workers. When the worker finishes processing, it returns a result to the manager, and then, requests for more data. If there is still data to be operated, the process repeats.
- The manager is usually replying to requests of data from the workers or receiving their partial results. Once all data pieces have been processed, the manager assembles a total result from the partial results and the program finishes. The non-serviced requests of data from the workers are ignored.

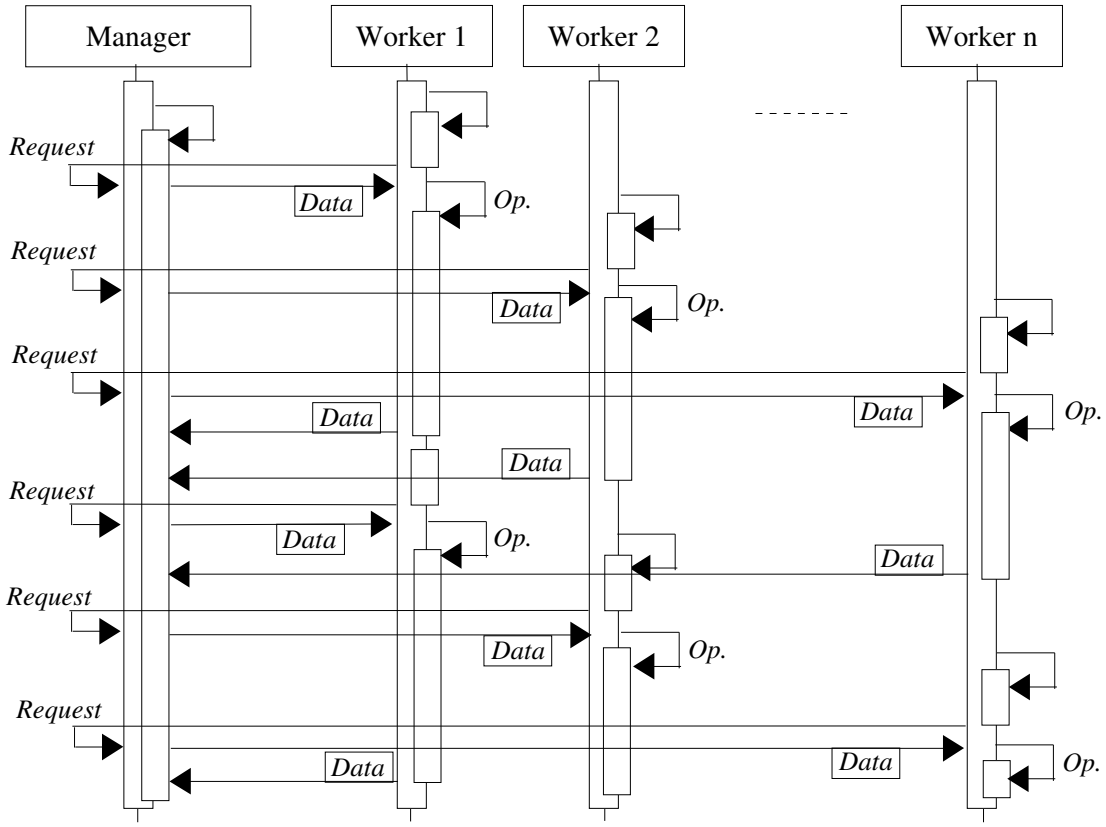

Figure 3. Interaction Diagram of the Manager-Workers pattern.

Implementation

An architectural exploratory approach to design is described below, in which hardware-independent features are considered early, and hardware-specific issues are delayed in the implementation process. This method structures the implementation process of parallel software based on four stages [Fos94, OR98]. During the first two stages, attention is focused on concurrency and scalability characteristics. In the last two stages, attention is aimed to shift locality and other performance-related issues. Nevertheless, it is preferred to present each stage as general considerations for design, instead of providing details about precise implementation. These implementation details are pointed more precisely in the form of references to design patterns for concurrent, parallel, and distributed systems of several other authors [Sch95, Sch98a, Sch98b, POSA00].

1. *Partitioning.* The ordered data to be operated on by the common computation is decomposed into a set of data pieces. This partitioning criteria of the ordered data is a clear opportunity for parallel execution, and it is used to define the partitioning and gathering activity of the manager component. On the other hand, the same computation to be performed on different data pieces is used to define the structure of each one of the worker components. Sometimes, the manager is also implemented to perform the computation on data pieces as well. Usually, the structure of the manager can be reused if it is designed to deal with different data types and sizes, delimiting its behaviour to divide, deliver and gather data pieces to the worker components. It is possible to implement either manager or workers using a single sequential component approach (for instance, a process, a task, a function, an object, etc.), or to define a set of components that perform manager or worker activities. Usually, concurrency among

these components can be used, defining different interfaces for different actions. Design patterns [GHJV95, POSA96] can help to define and implement such interfaces. Patterns that particularly can help with the design and implementation of the manager and worker components are the Active Object pattern [LS95, POSA00], (which allows to create a manager and workers able to execute concurrent operations on data) and the Component Configurator pattern [POSA00] (also known as the Service Configurator pattern [JS96], which allows the link and unlink of worker implementations at run-time in case that, in a particular application, they are permited to be created or destroyed dynamically). In the case of the worker components, other design patterns that may provide information about their implementation are the "Ubiquitous Agent" pattern [JP96] and the Object group pattern [Maf96].

2. *Communication.* The communication structure that coordinates the execution between the manager and worker should be defined. As workers are just allowed to communicate with the manager to get more work, defining an appropriate communication structure between manager and worker components is a key task. The communication structure should allow the interactions between the manager and each worker (request a data piece and, once processed, its delivery to the manager). Important parameters to consider are the size and format of data, the interface to service a request of data, and the synchronisation criteria. In general, a synchronous coordination is commonly used in Manager-Worker pattern systems. The implementation of communication structures depends on the programming language used. In general, if the language contains basic communication and synchronisation instructions, communication structures can be implemented relatively easily, following the single element approach. However, if it is possible to reuse the design in more than one application, it would be convenient to consider a more flexible approach using configurable communication sub-systems for the exchange of different types and sizes of data. Design patterns can help to support to the implementation of these structures; especially, consider the Composite Messages pattern [SC95], the Service Configurator pattern [JS96, POSA00], and the Visibility and Communication between Control Modules and Client/Server/Service patterns [AEM95, ABM96].
3. *Agglomeration.* The data division and communication structure defined previously are evaluated with respect to the performance requirements. If necessary, the size of data pieces is changed, modifying the granularity of the system. Data pieces are combined or divided into larger or smaller pieces to improve performance or to reduce communication costs. Due to inherent characteristics of this pattern, the process is automatically balanced among the worker components, but granularity is modified in order to balance the process between the manager and the workers. If the operations performed by the workers are simple enough and workers receive relatively small amount of data, workers may remain idle while the manager is busy trying to serve their requests. On the contrary, if worker operations are too complex, the manager will have to keep a buffer of pending data to be processed. It is noticeable that load-balance between manager and workers can be achieved simply by modifying the granularity of data division.
4. *Mapping*. In the best case, the hardware allows that each component is assigned to a processor with enough communication links to perform efficiently. However, generally the number of components is defined a lot bigger than the number of available processors. In this case, it is common to place a similar number of worker components on each processor. To keep the structure the most balanced possible, the manager component can be executed on a dedicated processor, or at least on a processor with a reduced number of working components. The competing forces of maximising

processor utilisation and minimising communication costs are almost totally fulfilled by this pattern. Mapping can be specified statically or determined at run-time, allowing a better load-balance. As a "rule of thumb", parallel systems based on the Manager-Workers pattern will perform reasonably well on a MIMD (multiple-instruction, multiple-data) computer, and it may be difficult to adapt to a SIMD (single-instruction, multiple-data) computer [Pan96].

Example Resolved

Considering the Polygon Overlay problem description, the Manager-Workers pattern is used to create a parallel solution. Briefly, such a parallel solution is described as follows: for the two input maps *A* and *B*, divide all the polygons belonging to *A* into sub-maps, and for each one of them, find all the intersections with a sub-map of *B*. The key for the parallel solution is to limit the part of both maps, *A* and *B*, that has actually to be looked at to find the overlaps. Using the Manager-Workers pattern, a set of workers do the actual polygon overlaying by simultaneously finding intersections for each sub-map A_{ij} with each sub-map B_{ij} (Figure 4). As defined in the pattern, a manager provides on request a sub-map to each one of the workers. Once processing is finished, the manager is sent the results by the workers. In the parallel implementation, the manager and workers are all made active objects.

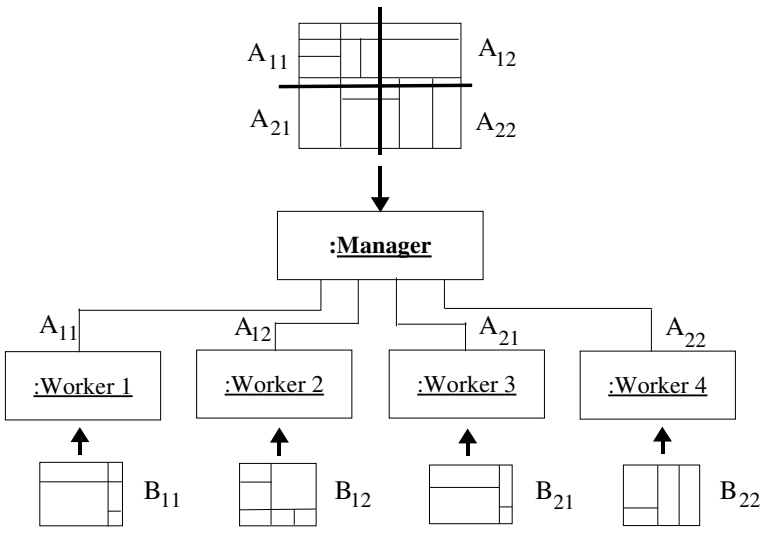

Figure 4. Object Diagram for the Polygon Overlay problem, dividing map *A* in four sub-maps and assigning them to four workers.

Partitioning

In the Manager-Worker pattern, the manager divides the data to be operated on into a set of data pieces, and gathers partial results to obtain a global result. For the present problem, the manager is considered exclusively to perform such operations, and it does not perform any other computation on the data. Figure 5 shows a partial implementation of the class `Manager`, which follows such considerations for partitioning the data structure `map` (map *A*), but does not deal with communication issues yet (these are dealt with during the communication step). Notice that, by now, it defines an attribute `workSize`, which allows to partition data in different sizes. Also, it has some other private attributes such as `lastPolySent`, which keeps record of the polygons operated, and `results`, where the

list of polygons are gathered once operated. The manager here acts as a single active object programmed in UC++ [WRMPD95] as subclass of the class `Activatable`.

```
class Manager: Activatable {
public:
    Manager ();
    ...
private:
    Map* map;
     polylist results;
     int lastPolySent;
     int workSize;
     ...
};
```

Figure 5. Class `Manager` for the partitioning of the Polygon Overlay problem.

On the other hand, as part of this step, the same computation to be performed on each sub-map is used to define the structure of the worker components. Figure 6 shows the class `Worker`, considering the essential elements to operate on a local `map` (actually, a sub-map of *A* provided by the manager). Notice that each worker is created using a constructor with `mapfile` and `m` as arguments. These refer to the file in which the second map (map *B*) is stored, and a reference to the manager object. Moreover, these arguments are asigned to the private attributes `map` and `manager`. These references are used during the communication.

```
class Worker: Activatable {
public:
    Worker ();
    Worker (char* mapfile, Manager* m);
    virtual void startWork();
    ...

private:
    Map* map;
    Manager* manager;
    ...
};
```

Figure 6. Class `Worker` for the Polygon Overlay problem.

Communication

During construction, each worker establishes connection with the manager. Immediately after construction, each worker requests data from the manager. It is here that the parallelism of the algorithm occurs, as this function is executed on each worker in parallel. During operation, when the provided data has been operated, each worker returns a partial result to the `results` data structure, repeating until the whole process has been carried out on all polygons. It is then when the function `printResults()` is called, sorting the result and writing it to a file.

Figure 7 shows the class `Manager`, which handles the communication and synchronisation with the workers. The code for the constructor is not shown: it just initialises the buffer values for temporarily store the exchanging maps with from the

workers. Notice the key functions `getWork()`, which requests the manager for a data piece to operate on, and `send()`, that allows retrieving results from workers.

```
class Manager: Activatable {
public:
    Manager ();
    virtual polylist* getWork();
    virtual void send(polylist& send_result);
private:
    Map* map;
     polylist results;
     int lastPolySent;
     int workSize;
};
```

Figure 7. Class `Manager` considering the communication of the Polygon Overlay problem.

Agglomeration and Mapping

Finally, Figure 8 shows the `main()` function for the program. This function initiates and manages the synchronisation of the manager and workers as active objects. Each worker is given a pointers to the manager, in order to request and receive data from it. Active objects are instantiated from the defined classes `Manager` and `Worker` by `activenew_Manager` and `activenew_Worker` respectively, as defined by UC++ [WRMPD95]. A non-blocking function call to `startWork()` is then made on each active object, which starts each one of them. Once all elements are active, a blocking function call to `blockWait()` is done on each of them after the loop for `startWork()` calls is finished, allowing all elements to complete their computations. The final statement requests to all elements to print their results.

```
int UCPP_main(int argc, char** argv)
{
  Manager *m = activenew_Manager();

  int nWorkers = 5;
  Worker* workers[nWorkers];

  int i;
  for (i = 0; i < nWorkers; i++){
    workers[i]  = activenew_Worker("",m);
  }
  for (i = 0; i < nWorkers; i++){
    workers[i] -> startWork();
  }

  for (i = 0; i < nWorkers; i++){
     blockWait(workers[i]);
  }

  m -> printResults();

  return 0;
}
```

Figure 8. `main()` function in UC++ for the Polygon Overlay problem.

Known uses

- Connectivity and Bridge Algorithms are an application of the Manager-Workers pattern. The problem is to determine if a connected graph has a bridge. A bridge is an edge whose removal disconnects the graph. A simple algorithm attempts to verify if an edge is a bridge by removing it and testing the connectivity of the graph. However, the required computation is very dense if the number of edges in the graph is large. A parallel version using a Manager-Worker pattern approach is described as follows: each worker, using the algorithm proposed, is responsible for verifying if an edge is a bridge. Different workers check for different edges. The manager distributes the graph information to the workers, builds the final solution, and produces results [NHST94].
- Matrix multiplication is the most classical parallel application of the Manager-Workers pattern. The matrixes are distributed among the workers by the manager. Each worker calculates products and returns the result to the manager. Finally, with all the results available, the manager can build the final result matrix [POSA96, Fos94].
- In image processing, the Manager-Worker pattern is used for transformations on an image that involve an operation on each piece of the image independently. For example, in computing discrete cosine transform (DCT), the manager divides the image in sub-images, and distributes them among the workers. Each separate worker obtains the DCT of its sub-images or pixel block and returns the result to the manager. The final image is then composed by the manager, using all the partial results provided by the workers [POSA96, Fos94].

Consequences

Benefits

- The order and integrity of data is preserved and granted due to the defined behaviour of the manager component. The manager takes care of what part of the data has been operated on, and what remains to be computed by the workers
- An important characteristic of the Manager-Workers pattern is due to the independent nature of operations that each worker performs. Each worker requests for a different piece of data during execution, which makes the structure to present a natural load balance [POSA96, CT92].
- As every worker component performs the same computation, granularity can be modified easily, due to it depends only on the size of the pieces in which the manager divides the data. Furthermore, it is possible to exchange worker components or add new ones without significant changes to the manager, if an abstract description of the worker is provided [POSA96].
- Synchronisation is simply achieved because communications are restricted to only between manager and each worker. The manager is the component in which the synchronisation is stated.
- Using the Manager-Worker pattern, the parallelising task is relatively straightforward, and it is possible to achieve a respectable performance if the application fits this pattern. If designed carefully, the Manager-Worker pattern enables the performance to be increased without significant changes [POSA96, Pan96].

Liabilities

- The Manager-Workers systems may present poor performance if the number of workers is large, the operations performed by the workers are too simple, or workers receive small amounts of data. In all cases, workers may remain idle for periods of time while the manager is busy trying to serve all their requests. Granularity should be modified in order to balance the amount of work.
- Manager-Worker architectures may also have poor performance if the manager activities - data partition, receive worker requests, send data, receive partial results, and computing the final result - take a longer time compared with the processing time of the workers. The overall performance depends mostly on the manager, so programming the manager should be done taking special consideration to the time it takes to perform its activities. A poor performance of the manager impacts heavily on the performance of the whole system [POSA96, CT92].
- Many different considerations must be carefully considered, such as strategies for work subdivision, manager and worker collaboration, and computation of final result. In general, the issue is to find the right combination of worker number, active or passive manager, and data size in order to get the optimal performance, but experience shows that this still remains a research issue. Moreover, it is necessary to provide error-handling strategies for failure of worker execution, failure of communication between the manager and workers, or failure to start-up parallel workers [POSA96].

Related patterns

The Manager-Workers pattern can be considered as a variant of the Master-Slave pattern [POSA96, POSA00] for parallel systems. Many parallel programming authors consider it as a basic organisation for parallel computation [CT92, KSS96, Hart98, LB00, Andr00]. Other related patterns with similar approaches are the Object Group pattern [Maf96], and the Client/Server/Service pattern [ABM96].

3. Summary

The goal of the present paper is to provide parallel software designers and engineers with an overview of the Manager-Workers pattern as a common structure used for activity parallel in software systems. The architectural pattern described here can be linked with other current pattern developments for concurrent, parallel and distributed systems. Work on patterns that support the design and implementation of such systems has been addressed previously by several authors [Sch95, Sch98a, Sch98b, POSA00].

4. Acknowledgements

The author wishes to express his recognition to all those who were involved in the improvement of the present paper, in particular to Egon Wuchner, my shepherd for EuroPLoP 2004, whose advise was extremely important in the development of the last version. Also, I would like to recognise the effort and help of the participants of the Writers' Workshop A, for their comments and companion during EuroPLoP 2004.

5. References

[ABM96] Amund Aarsten, David Brugali and Giuseppe Menga. *Patterns for Cooperation*. Pattern Languages of Programming Conference (PLoP'96). Allerton Park, Illinois, USA. September 1996.

[AEM95]Aarsten, A., Gabriele Elia, G., and Giuseppe Menga, G. *G++: A Pattern Language for the Object Oriented Design of Concurrent and Distributed Information Systems, with Applications to Computer Integrated Manufacturing.* Department of Automatica e Informatica, Politecnico de Torino. In J. Coplien and D. Schmidt (eds.) Pattern Languages of Program Design. Reading, MA: Addison-Wesley, 1995.

[Andr00] Gregory R. Andrews. *Foundations of Mutithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[CG88] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs. A Guide to the Perplexed.* Yale University, Department of Computer Science, New Heaven, Connecticut. May 1988.

[CT92] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, Inc., Boston, 1992.

[Fos94] Ian Foster. *Designing and Building Parallel Programs, Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Publishing Company, 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Systems.* Addison-Wesley, Reading, MA, 1994.

[Hart98] Stephen J. Hartley. *Concurrent Programming. The Java Programming Language.* Oxford University Press, 1998.

[JS96] Prashant Jain and Douglas C. Schmidt. *Service Configurator. A Pattern for DynamicConfiguration and Reconfiguration of Communication Services.* Third Annual Pattern Languages of Programming Conference, Allerton Park, Illinois. September 1996.

[LS95] R. Greg Lavender and Douglas C. Schmidt. *Active Object. An Object Behavioral Pattern for Concurrent Programming* .In *Patterns Languages of Programming 2* (PLOP'95). Addison-Wesley, 1996.

[LB00] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Java Technology.* Sun Microsystems, Inc., 2000.

[KSS96] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads.* Sun Microsystems, Inc. 1996.

[Maf96]Maffeis, S. *Object Group, An Object Behavioral Pattern for Fault-Tolerance and Group Communication in Distributed Systems.* Department of Computer Science, Cornell University. Proceedings of the USENIX Conference on Object-Oriented Technologies. Toronto, Canada, 1996.

[NHST94] Christopher H. Nevison, Daniel C. Hyde, G. Michael Schneider, Paul T. Tymann. *Laboratories for Parallel Computing.* Jones and Bartlett Publishers, 1994.

[OR98] Jorge L. Ortega-Arjona and Graham Roberts. *Architectural Patterns for Parallel Programming.* Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing, EuroPloP'98. Jens Coldewey and Paul Dyson (editors), Universitätsverlag Konstantz GmbH, 1999.

[Pan96] Cherri M. Pancake. *Is Parallelism for You?* Oregon State University. Originally published in Computational Science and Engineering, Vol. 3, No. 2. Summer, 1996.

[POSA96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal. *Pattern-Oriented Software Architecture.* John Wiley & Sons, Ltd., 1996.

[POSA00] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. *Pattern-Oriented Software Architecture, Vol. 2 - Patterns for Concurrent and Distributed Objects.* John Wiley and Sons, Ltd., 2000.

[SC95] Aamond Sane and Roy Campbell. *Composite Messages: A Structural Pattern for Communication Between Components.* OOPSLA'95, Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems. October 1995.

[Sch95] Douglas Schmidt. *Accepted Patterns Papers.* OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems. http://www.cs.wustl.edu/~schmidt/OOPSLA-95/html/papers.html. October, 1995.

[Sch98a] Douglas Schmidt. *Design Patterns for Concurrent, Parallel and Distributed Systems.* http://www.cs.wustl.edu/~schmidt/patterns-ace.html. January, 1998.

[Sch98b] Douglas Schmidt. *Other Pattern URL's. Information on Concurrent, Parallel and Distributed Patterns.* http://www.cs.wustl.edu/~schmidt/patterns-info.html. January, 1998.

[ST96] David B. Skillicorn and Domenico Talia. *Models and Languages for Parallel Computation.* Computing and Information Science, Queen's University and Universita della Calabria. October 1996.

[WL96] Wilson, G. V., and Lu, P., eds. *Parallel Programming using C++*. Scientific and Engineering Computation Series. The MIT Press. Cambridge, Massachusets. 1996.

[WRMPD95] Winder, R., Roberts, G., McEwan, A., Poole, J., Dzwig, P. *The UC++ Project.* `http://ww.dcs.acl.ac.uk/UC++/`