2

Pact: an environment for parallel heuristic programming

Marc Aguilar, Béat Hirsbrunner

ABSTRACT. This paper introduces Pact, a new model for parallel heuristic programming on MIMD machines. Designed and implemented as an integrated tool environment, Pact allows the user to develop and test (1) new distributed artificial intelligence applications, (2) new parallel heuristic search strategies and (3) new dynamic allocation algorithms. The modular design and layer abstraction of Pact offers to the user features such as a powerful programming expressiveness, software modularity, portability and scalability. Three different prototypes of Pact have been implemented and tested in programming environments such as C-iPSC/2, Strand-Sun and Strand-C-Sun.

RÉSUMÉ. Dans ce papier, nous présentons Pact, un nouveau modèle pour la programmation heuristique parallèle sur des machines MIMD. L'objectif principal de Pact est de fournir des outils et un environnement de programmation permettant à l'utilisateur de développer et de tester (1) de nouvelles applications dans le domaine de l'intelligence artificielle distribuée, (2) de nouvelles stratégies de recherche heuristique et (3) de nouveaux algorithmes de placement dynamique. L'approche modulaire de Pact offre à l'utilisateur une grande expressivité au niveau de la programmation. Elle autorise en outre la portabilité et l'échelonnement des applications. Trois prototypes ont été implémentés dans différents environnements, notamment sous C-iPSC/2, Strand-Sun et Strand-C-Sun.

KEYWORDS: Distributed and massively parallel programming environments, parallel heuristic programming, dynamic migration, fuzzy logic controlled allocation, postal mail delivery, modularity, portability, scalability.

MOTS-CLÉS: Environnement de programmation distribué et massivement parallèle, programmation heuristique parallèle, migration dynamique, allocation contrôlée par la logique floue, distribution postale du courrier, modularité, portabilité, échelonnement.

---

1. Introduction

The next generation of supercomputers will be massively parallel, with virtual shared memory MIMD architectures and tens of thousands of processing elements. To make these machines available as suitable platforms for parallel computing, user acceptance demands that these systems should be programmable in an appropriate, user-friendly, application-oriented way. Hence, parallel programming environments should be portable, scalable and free of machine dependent communication technique and scheduling problems. In this paper, we present Pact, a model for parallel heuristic programming that fulfills these requirements. To achieve the aims mentioned above, we designed and implemented Pact as an integrated tool environment for the development and testing of new :

- distributed artificial intelligence applications, [Courant and Ludwig, 1994];
- parallel heuristic search strategies, [Gengler and Coray, 1993];
- dynamic allocation algorithms, [Stoffel et al., 1993].

Portability and scalability of the Pact environment are guaranteed by the:

- multilevel control structure of the general model (section 2.1);
- modular software and hardware approach of Pact (see section 3.2 and 3.3).

The experiences and results gained by the Pact project are currently exploited for the design of a new coordination language called CoLa, [Aguilar et al., 1994], [Hirsbrunner, 1993], [Hirsbrunner et al., 1994].

2. Model

The main objective of Pact is to offer a programming environment which allows the user to test all kinds of heuristic search strategies (A*, Alpha-Beta, Scout, SSS*) in a massive parallel environment but also to provide a platform for the development of new unknown heuristics and increase the domain of applications. In the following sections, we present a general model for heuristic programming and illustrate how the different modules of Pact can be programmed.

2.1. A multilevel control structure

In our scheme a problem P is solved by the classical reduction technique, i.e. P generates an AND/OR tree of subproblems. In order to control the combinatorial explosion of this tree and to guarantee an optimal distribution of the subprob-

lems among the available processors, [Schiper et al., 1984] proposes a multilevel control structure (see figure 1). On level 1 we are only concerned with the resolution algorithm of a problem P. If P is easy enough it is solved by a direct method, otherwise it is splitted up into subproblems. On level 2 we express the search strategy by associating to each problem P a controller C. The job of C is to calculate periodically the priority list of P's subproblems by taking into account the heuristics available from his parent (inheritance), his childrens (synthesis) and his associated problem P. Level 1 and level 2 are machine independent and allow to separate the algorithms specific to the applications (level 1) and the search strategies (level 2). Finally on level 3, which is machine dependent, we express the physical scheduling of the subproblems. Note that this model is not limited to tree structures and can be easily generalized to any kind of interconnection topology, e. g. graphs.

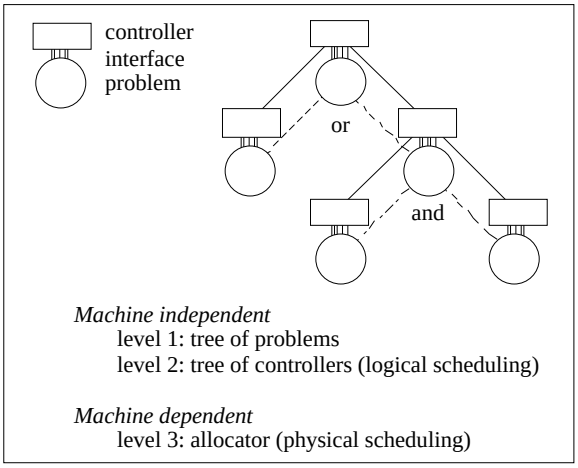

Figure 1. *Model for the parallel heuristic programming*

2.2. Pact space

Massively parallel computing requests to break up the multilevel control structure of figure 1 in its most common atomic elements which can progress in a parallel environment. Inspired by the Gamma [Banâtre and Métayer, 1993] and Linda [Gelernter and Carriero, 1990] parallel programming models, the entities are grouped in a multiset called Pact-space. The solution is then obtained by cooperation and competition of the different elements. For our parallel heuristic programming model, these elements represent the subproblems and the controllers. In the case of the multi-agent paradigm in distributed artificial intelligence, the autonomous entities are the agents. For the recognition of a picture defined by a matrix of pixels, the atomic entities could be the pixels. Of course,

the granularity of the entities depend on the application and if it is to small, atomic elements should be grouped into molecular elements until the granularity meets the requests of the application as well as the soft and hardware constraints. But what are the most suitable mechanisms to manage the distributed Pact-space? How should we represent the distributed interconnection of the elements? How can we express communication and synchronization between atomic and molecular entities? How should the resources be allocated to the entities? In the following section, we present one possible programming environment of Pact on an MIMD architecture.

2.3. Pact on a local memory MIMD machine

Figure 2 outlines one possible architecture of the Pact model on an MIMD computer with local memory. This environment runs on each node of the target machine. As presented in the general model, the resolution of the subproblems (level 1) is guaranteed by the workers and the logical scheduling (level 2) by the controller. The implementation of the Pact model on a distributed MIMD architecture requests the introduction of two machine dependent components, e.g. an allocator and a post. The allocator (level 3) physically schedules the subproblems onto the resources. The post guarantees the exchange of intra- and intersite mail. In the following sections, we give a more detailed presentation of the different Pact components[1].

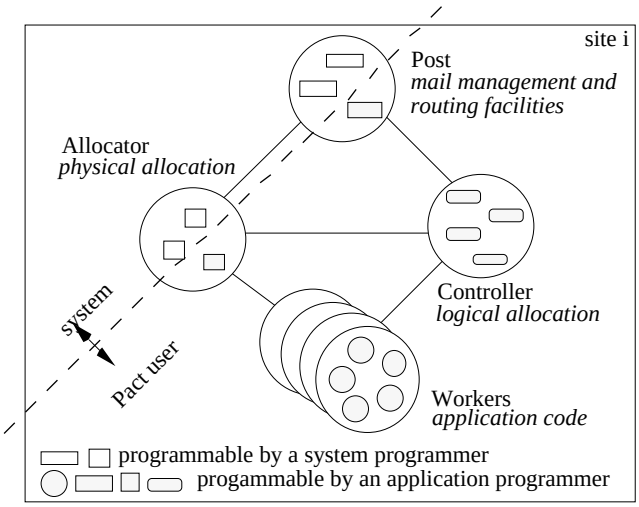

Figure 2. *Pact model for parallel heuristic programming on a MIMD machine*

[1] Pact is a french acronym for "Poste, Allocateur, Contrôleur et Travailleurs (workers)".

2.4. Workers

Workers hold the application code and are programmed by the user in an imperative, functional or declarative programming language. The best choice is application dependent. The code skeleton however is always the same, according to the heuristic programming paradigm: for a problem P, if P is easy it is solved by a direct method, otherwise it is splitted up into subproblems. An example written in an imperative programming language is given in the appendix.

2.5. Controllers

Heuristic search strategies are computed in the controllers using a declarative-like language. Periodically, workers transmit heuristic information to the controllers which is used to define the logical scheduling priorities of subproblems. Using these priorities, controllers manage the combinatorial growth of the tree and define a logical order of activation for the most urgent subproblems to be solved. Local computed priority lists are transmitted to the allocator which computes the physical scheduling of subproblems onto the resources. We refer to the appendix for the declarative code of different controllers (AND/OR, Alpha-Beta, SSS* and B*).

2.6. Allocator

Efficient programming of parallel systems requests an optimal use of the available resources. The non deterministic nature of heuristic computation implies that an efficient allocator has to be scalable and configurable. In Pact, we offer to the user beside a large range of standard mapping algorithms, a fuzzy logic controlled dynamic allocation system, [Stoffel et al., 1993]. On the user level, the allocation system can be tuned and guided by fuzzy logic (see figure 3). It has been shown, that this technique is best suitable for dynamic distribution of independent processes when the allocation strategy has to be define in terms of machine dependent information, application characteristics and load balancing criteria.

```
(1)  IF cpuLoad = high AND memoryLoad = high THEN migrationCoeff = high;
(2)  IF cpuLoad = high AND memoryLoad = middle THEN migrationCoeff = middle;
(3)  IF cpuLoad = low THEN migrationCoeff = low;
(4)  IF communicationLoad = high THEN migrationCoeff = low;
```

Figure 3. *Fuzzy logic rules sample allowing a user to tune the allocator*

2.7. Post

The post of Pact implements a new postal mail delivery service which provides portability, scalability, flexibility and transparency. Mail management tools for dynamic load balancing strategies are defined by concepts such as a distributed finite state machine, a distributed address book and a new high level identification concept, defining communicating processes as correspondents, see [Aguilar and Hirsbrunner, 1994]. This high level identification abstraction facilitates the implementation of complex communication structures by decoupling logical and physical communication structures. The post offers different communication and dynamic migration protocols, each one optimized for criteria such as memory optimization, pending mail cut down, shortest mail delivery path and runtime. Figure 4 gives an example how the user of Pact can tune the postal mail services using fuzzy logic rules.

```
(1)  IF memoryLoad = high THEN use protocol B
(2)  IF migrationLoad = high AND memoryLoad = low  THEN use protocol B*
(3)  IF migrationLoad = low  AND memoryLoad = low  THEN use protocol A
```

Figure 4. *Fuzzy logic rules sample allowing a user to tune the post*

3. Results

Three different prototypes of Pact have been implemented and tested. This section gives a short description of the characteristics, benefits and drawbacks of the prototypes and presents the results gained from our experiences.

3.1. Prototypes

To emphasize the programming innovation of the Pact model and put into practice the theoretical results, we implemented three different prototypes in a C-iPSC/2, Strand-Sun[1] and Strand-C-Sun environment, see [Goy, 1993]. For all prototypes, the Pact components use a message passing communication mechanism. The C-iPSC/2 prototype implements the Pact model in a heavy weighted Unix-process environment. The Strand-Sun prototype is a light weighted process version of the Pact model and was used to test and develop the controllers of Pact. The third prototype implements Pact in a hybrid heavy-light weighted process environment[2] and gave some interesting results on application granularity, hybrid interprocess communication and scheduling of the Pact components.

[1] Strand is a parallel logic programming language.
[2] The workers are written in C and the other components of Pact in Strand.

All prototypes have been tested with the *Awélé* two players game which is simple to program, well formalized and allows an easy dynamic variation of the granularity of the subproblems. As this paper emphasizes on the new programming methodologies in massive parallel systems rather than on performances results, the latter are intentionally omitted here.

The following two sections present the advantages of the modular Pact approach. On the software level, we will discuss power of expression, code-reuse and application granularity. On the hardware level, we will present portability and scalability aspects.

3.2. Software modularity

EXPRESSIVENESS: The modular design of Pact offers to the user a high level, powerful programming expressiveness for massive parallel systems. The four abstraction levels in Pact allow the user to program each component independently by using the most suitable formalism for its functionality (e. g. imperative, declarative or fuzzy logic formalism). The hybrid Strand-C-Sun version confirms that a system can be reasonably realized in a programming environment with heavy and light weighted processes and with different programming formalisms.

CODE-REUSE: Another important aspect of software modularity is code-reuse. For all prototypes we used the same code written for the *Awélé* application and the associated controllers. Furthermore, the postal mail delivery concept makes message passing communication transparent to the application. For the C-iPSC/2 prototype, the mail management facilities of Pact could be developed, upgraded and extended with no side effects on the application code.

GRANULARITY: Application granularity is a very important parameter for tuning parallel systems. We tested the effects of fine to medium process granularity in the Strand-Sun prototype and medium to large granularity in the C-iPSC/2 and Strand-C-Sun versions. Test cases in the hybrid version showed that an optimum is only reached for dynamic tuning of the granularity. Therefore, future programming environment such as Pact have to offer dynamic tuning facilities for application granularity, logical and physical scheduling as well as for postal mail delivery services.

3.3. Hardware modularity

The modular layer abstraction of Pact seems to be an important paradigm for massive parallel programming environments. Our modular approach shows that hardware characteristics of the target machine can be hidden from the user or be

controlled on a high abstraction level, if the programming environment implements a modular decomposition and a clear separation of its functionality. Under these conditions, portability, scalability and transparency become reasonable.

PORTABILITY: Portability becomes realistic, because the Pact developer only needs to change machine dependent modules. Interfaces to other modules remain unchanged, only the functionality has to be adapted to the underlying characteristics of the target machine. We are currently underway to port the Post of Pact onto the ISIS platform without changing the offered facilities of the Post.

SCALABILITY: We also tested scalability of the machines which hosted Pact. Scalability by a factor of 10 was achieved without any changes. Tuning of the application granularity and the fuzzy logic controlled tuning of the Post and the Allocator allow us to go beyond this limit. Our thesis is that this new tuning technique makes next generation parallel systems even more scalable.

4. Discussion

Our different prototypes clearly show that in a massively parallel system:

- future programming environments have to be realized with heavy and light weighted processes;
- the classical available communication and synchronization tools are neither adequate for a clever specification nor for an efficient implementation of a parallel programming environment.

For the C-iPSC/2 prototype, the lack of high level communication mechanisms, dynamic distributed memory control, local priority process scheduling and tuning represent an important barrier to the development of Pact. For the Strand-Sun prototype, we miss an efficient Strand-memory garbage collector, which would allow us to make performance tests for real world applications.

But one of the most important problem of programming massive parallel systems consists in the lack of a general computing model such as the Von Neuman machine represents for sequential programming. This new general model necessarily has to include dynamic and open systems. If the definition of an optimal solution was sufficient for sequential programming, parallel computing needs to elaborate and define in addition an optimal distributed resolver. In such a system, global state reasoning makes no sense any more and optimization cannot entirely be defined by convergence results. Criteria, such as scalability, adaptability, evolution, self-organization and coordination have to be considered and included in the models, see [Aguilar et al., 1994].

The theoretical and practical results gained by the Pact project are currently used for the design and implementation of a new coordination language, called CoLa, see [Hirsbrunner et al., 1994]. In this project, we define some new seman-

tics for communication mechanisms in parallel systems. Especially, logical global communication topologies are locally defined and updated without any global state control.

5. Conclusion

Pact is an environment for parallel heuristic programming on MIMD architectures with local memory. Different prototypes of Pact have been realized and tested in programming environment such as C-iPSC/2, Strand-Sun and Strand-C-Sun. It has been shown that Pact makes massively parallel computers much easier to program. The high level expressiveness in Pact introduces a new programming style for massively parallel environments. The modular approach of the Pact model allowed us to test portability, scalability and code-reuse.

The specification and implementation of Pact clearly showed that the fundamental models and tools needed for the programming of massively parallel systems are still missing. We will concentrate our efforts on one of this, namely in the development of a new coordination language called CoLa, see [Aguilar et al., 1995]. In this project, we define new semantics for communication and synchronization mechanisms in a massively parallel environment.

It remains, that the most severe limit to innovative programming methods for parallel systems is the absence of a general computing model such as the Von Neuman machine has been for sequential programming.

References

[Aguilar and Hirsbrunner, 1994] Marc Aguilar, Béat Hirsbrunner: "Post: A new postal mail delivery model for parallel heuristic programming". In "Conference on Programming Environments for Massively Parallel Distributed Systems"; IFIP WG 10.3, Ascona, Switzerland, April 25-30, 1994.

[Aguilar et al., 1994] Marc Aguilar, Michèle Courant, Béat Hirsbrunner: "Le défi des architectures parallèles à l'I.A.: La dimension collective de l'intelligence". Revue d'Intelligence Artificielle, vol. 8, 1994. A paraître.

[Aguilar et al., 1995] Marc Aguilar, Béat Hirsbrunner, Oliver Krone: "The CoLa Approach: A Coordination Language for Massively Parallel systems". HICSS-28 (28th Hawaii International Conference on Systems Sciences); Maui, Hawaii, January 3-6, 1995. Abstract accepted, full paper submitted.

[Banâtre and Métayer, 1993] J.-P. Banâtre, D. Le Métayer: "Programming by Multiset Transformation". Communications of the ACM, January 1993, pp. 98-111.

[Courant and Ludwig, 1994] M. Courant, M. Ludwig: "Un modèle d'interaction basé sur les forces". Paru dans E. Bonabeau, Th. Fuhs (éds): "Autonomie et interactions fonctionnelles", Actes des journées de Rochebrune, 17-21 janvier 1994, 9 pages.

[Gelernter and Carriero, 1990] D. Gelernter, N. Carriero: "How to Write Parallel Programs: A First Course". MIT Press, 1990.

[Gengler and Coray, 1993] M. Gengler, G. Coray: "A Parallel Best-First Branch & Bound Algorithm and Its Axiomatization". Proceedings of the 26th Hawaï International Conference on System Sciences, vol. 2, January 1993, pp. 265-274.

[Goy, 1993] G. Goy: "Pact: un environnement pour le développement d'applications heuristiques en programmation logique parallèle". Thèse de doctorat, IIUF-Fribourg, juin 1993, 148 pages.

[Hirsbrunner, 1993] Béat Hirsbrunner: "CoLa: un langage de coordination pour la programmation heuristique parallèle". Grant n° 5003-034409 of the Swiss National Science Foundation, 1993-1995.

[Hirsbrunner et al., 1994] B. Hirsbrunner, M. Aguilar, O. Krone: "CoLa: a coordination language for massively parallel systems". ACM Symposium on Principles of Distributed Computing; Los Angeles, USA, August 14-17, 1994.

[Schiper et al., 1984] A. Schiper, G. Coray, B. Hirsbrunner: "A two-level control structure for parallel heuristic programming". Technology and Science of Informatics, vol. 3, n° 1, 1984, pp. 27-36.

[Stoffel et al., 1993] K. Stoffel, I. Law, B. Hirsbrunner: "Fuzzy Logic Controlled Dynamic Allocation System". In H. Kitano, V. Kumar, C. B. Suttner (editors): "Parallel Processing For Artificial Intelligence", vol. 2, Elsevier Science Publishers B.V., North Holland, 1993.

Appendix

In this appendix we present a skeleton code for the Workers and their associated controllers for two typical search tree problems: the AND/OR and MIN/MAX (with their variants Alpha-Beta, SSS* and B*).

The Worker skeleton procedure for an AND/OR search tree is given in figure A2. At line (24), the Worker transmits heuristic information to its associated interface, which will be used by the controllers to compute the search strategy and to control the combinatorial growth of the tree. At line (30), all subproblems sp[i] are created. The controller of type AND or OR, responsible to calculate periodically the priorities of the subproblems sp[i], is created at line (29).

The associated AND/OR controllers are defined in a declarative-like language, see figure A1. Line (06), for example, ensures that at all time the merit is equal to the minimum of the associated children's merits.

The codes for an AND/OR and MIN/MAX search tree are very similar. Figures A4-A6 give the changes needed to transform the AND/OR code of figures A1-A3 to an Alpha-Beta code (a pure MIN/MAX code is obtained by simply removing the two lines (07) and (13) of figure A4). The well known deep cut of Alpha-Beta is expressed here by the explicit inheritance of lines (02)-(03) and (08)-(09) of figure A4.

```
(01)  interface AndOr is
(02)    merit: integer := parent.merit;
(03)  end AndOr;

(04)  controller And is
(05)    -- interface update
(06)    merit := min(c.merit: c in children);
(07)    -- priority update
(08)    for all c1, c2 in children: c1.priority = c2.priority;
(09)  end And;

(10)  controller Or is
(11)    -- interface update
(12)    merit := max(c.merit: c in children);
(13)    -- priority update
(14)    for all c1, c2 in children:
(15)      c1.merit < c2.merit => c1.priority < c2.priority;
(16)      c1.merit = c2.merit => c1.priority = c2.priority;
(17)  end Or;
```

Figure A1. *AND/OR controllers and their associated interface*

```
(18)  with interface AndOr;
(19)  with controller And, Or;

(20)  procedure Worker(p: in problem; s: out solution) is
(21)  var sp:array of problem;
          ss:array of solution;
          ctr:controller;
(22)  ...
(23)  begin
(24)    merit := p.merit;  -- AndOr interface update
(25)    if ProblemIsEasy(p) then
(26)      SolveProblem(p,s);
(27)    else
(28)      SplitProblemIntoSubproblems(p,sp,ctr);
(29)      NewControllerProcess(ctr);
(30)      for all sp[i] do
            NewWorkerProcess(Worker(sp[i],ss[i]));
(31)      case ctr of
(32)        when And => wait for all ss[i];
(33)        when Or  => wait for one ss[i];
(34)      ConstructSolution(ss,s);
(35)  end Worker;
```

Figure A2. *Worker skeleton procedure for a typical AND/OR search tree*

```
(36)  procedure SolveProblem (p: in problem; s: out solution) is
(37)  begin
(38)     loop  -- typical code for combinatorial problems
(39)       Update(merit);  -- periodical AndOr interface update
(40)       ...
(41)  end SolveProblem;
```

Figure A3. *SolveProblem skeleton procedure for the Worker of figure A2*

A lot of variants of Alpha-Beta have been discussed in the literature. All these variants can be expressed with very little changes of figures A4-A6. For example, line (02) of figure A7 expresses the fact that the SSS* search strategy controls the combinatorial explosion of the tree by always working at most on a solution tree (all MAX children, one MIN child). Line (01) of figure A8 expresses the fact that the B* search strategy is informed, that means the development of a subtree T is guided by heuristic information before an exhaustive solution tree of T has been constructed.

```
-- For the AlphaBeta interface,replace line (2) of figure A1 with:
(01)    alpha, beta: integer := (-∞, +∞);

-- For the Alpha controller,replace line (12) of figure A1 with:
(02)    alpha := max(parent.alpha, max(c.alpha: c in children));
(03)    beta  := min(parent.beta,  max(c.beta : c in children));
(04)    -- merit update
(05)    merit := alpha;
(06)    -- cut update
(07)    for all c in children: (alpha ≥ c.beta) => cut(c);

-- For the Beta controller, replace line (12) of figure A1 with:
(08)    alpha := max(parent.alpha, min(c.alpha: c in children));
(09)    beta  := min(parent.beta,  min(c.beta : c in children));
(10)    -- merit update
(11)    merit := -beta;
(12)    -- cut update
(13)    for all c in children: (beta ≤ c.alpha) => cut(c);
```

Figure A4. *Alpha-Beta controllers and their associated interface*

```
-- Remove line (24) and remove lines (32)-(34) of figure A2 with:
(14)        when Alpha => wait for all ss[i];
(15)                      for all ss[i] do s := max(s,ss[i]);
(16)        when Beta  => wait for all ss[i];
(17)                      for all ss[i] do s := min(s,ss[i]);
```

Figure A5. *Worker skeleton procedure for a typical Alpha-Beta search tree*

```
-- Replace lines (38)-(40) of figure A3 with:
(18)   -- typical code for game problems, like chess
(19)   CalculateEvaluation(eval);
(20)   (s, alpha, beta) := (eval, eval, eval);
```

Figure A6. *SolveProblem skeleton procedure for the Worker of figure A5*

```
-- Same as alpha-beta, by just adding after line (13) of fig. A4:
(01)    -- freeze update
(02)    for all but one s in children: freeze(s);
```

Figure A7. *SSS* controllers, their associated interface and the Worker skeleton procedure*

```
-- Same as alpha-beta, by just reintroducing
-- a line corresponding to line (24) of figure A2:
(01)    (alpha, beta) := (p.alpha, p.beta); -- B* interface update
```

Figure A8. *B* controllers, their associated interface and the Worker skeleton procedure*

This work has been presented at the MPP'94 Conference (First International Conference on Massively Parallel Processing: Applications and Development), Delft, The Netherlands, June 21 - 23, 1994.

HyperSpark: A Data-Intensive Programming Environment for Parallel Metaheuristics. Publisher: IEEE. 4.Â In this paper, we address this gap by proposing HyperSpark, an optimization framework for the scalable execution of user-defined, computationally-intensive heuristics. We designed HyperSpark as a flexible tool meant to harness the benefits (e.g., scalability by design) and features (e.g., a simple programming model or ad-hoc infrastructure tuning) of state-of-the-art big data technology for the benefit of optimization methods. We elaborate on HyperSpark and assess its validity and generality on a library implementing several metaheuristics for the Permutation Flow-Shop Problem (PFSP). This paper introduces Pact, a new model for parallel heuristic programming on MIMD machines. Designed and implemented as an integrated tool environment, Pact allows the user to develop and test (1) new distributed artificial intelligence applications, (2) new,parallel heuristic search strategies and (3) new,dynamic,allocation algorithms. The modular,design and layer abstraction of Pact offers to the user features such as a powerful programming expressiveness, software modularity, portability and scalability. Three different prototypes of Pact have been,implemented,and tested in programming env... SPC3 PM, Serial Parallel and Concurrent Core to Core programming Model provides an environment to decompose the application into tasks using its task decomposition rules and then execute these tasks in serial, parallel and concurrent fashion. As LKH-2 software is written in function style so we have to only restructure the some part of the code to make it suitable for SPC3 PM.Â Documents Similar To A Parallel and Concurrent Implementation of Lin-Kernighan Heuristic (LKH-2) for Solving Traveling Salesman Problem for Multi-Core Processors Using SPC3 Programming Model.