The Iso-Level Scheduling Heuristic for Heterogeneous Processors

Olivier Beaumont, Vincent Boudet and Yves Robert
LIP, UMR CNRS–ENS Lyon–INRIA 5668
Ecole Normale Supérieure de Lyon
69364 Lyon Cedex 07, France
[Olivier.Beaumont,Vincent.Boudet,Yves.Robert]@ens-lyon.fr

Abstract

Scheduling computational tasks on processors is a key issue for high-performance computing. Although a large number of scheduling heuristics have been presented in the literature, most of them target only homogeneous resources. We present a new scheduling heuristic for heterogeneous processors, which improves the load-balancing achieved at each decision step while keeping a low complexity. Experimental comparisons with five heuristics taken from the literature (BIL, GDL, CPOP, HEFT and PCT) and using six classical testbeds, show very favorable results.

1. Introduction

The efficient scheduling of application tasks is critical to achieving high performance in parallel and distributed systems. The objective of scheduling if to find a mapping of the tasks onto the processors, and to order the execution of the tasks so that: (i) task precedence constraints are satisfied; and (ii) a minimum schedule length is provided. Since the scheduling problem with communication delays is NP-hard [4], various heuristics have been proposed in the literature (see the tutorial [1]).

Although various different approaches are used to solve the task scheduling problem, most of them target homogeneous processors only. Heterogeneity poses new challenges to scheduling techniques. Scheduling methods that are suitable for homogeneous environments may well not be efficient for heterogeneous domains. For instance, clustering techniques (such as Gerasoulis and Yang's dominant sequence clustering [11]) are widely used in the context of homogeneous parallel machines, while they seem difficult to use in the context of heterogeneous processors. In the literature, most heuristics for heterogeneous processors are adaptations of list-scheduling techniques for homogeneous processors, and are still based upon critical paths and bottom levels.

In this paper, we introduce a new scheduling heuristic for heterogeneous processors, which we name the *Iso-Level Heterogeneous Allocation (ILHA)* heuristic. In a word, the main characteristic of the ILHA heuristic is a better load-balancing at each decision step, which is achieved by considering a chunk of several ready tasks rather than a single one; the idea is to allocate to each processor a number of the tasks in the chunk that is proportional to its computing power. We compare the ILHA heuristic with five heuristics taken from the literature: the *minimum Partial Completion Time static priority* (PCT) heuristic, the *Best Imaginary Level* (BIL) heuristic, the *Heterogeneous Earliest Finish Time* (HEFT) heuristic, the *Critical Path on a Processor* (CPOP) heuristic and the *Generalized Dynamic Level* (GDL) heuristic. For the experimental comparisons, we use six classical testbeds: LAPLACE, LU, STENCIL, FORK-JOIN, DOOLITTLE, and LDMt. All these comparisons show very favorable results. Note that the ILHA heuristic requires a simple graph traversal, which renders it very attractive to process huge size problems.

The rest of the paper is organized as follows. Section 2 is devoted to some technical preliminaries (definitions and notations). We briefly survey the five heuristics from the literature in Section 3. We present the ILHA heuristic in Section 4. In Section 5, six classical testbeds are used to compare the different heuristics. Finally, we give some concluding remarks in Section 6.

2. Preliminaries

In this section, we specify some notations for the standard *macro-dataflow* model, which is widely used in the scheduling literature [1]. For each task scheduling algorithm, the input is a directed acyclic graph

Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (EUROMICRO-PDP'02)
1066-6192/02 $17.00 © 2002 IEEE

$\mathcal{G} = (\mathcal{N}, \mathcal{A})$, that models a parallel program, where $\mathcal{N} = \{N_i : i = 1, \cdots, N\}$ is a set of N nodes and $\mathcal{A} = \{A_{i,j}\}$ is a set of edges. A node in the DAG represents a task. Each task has a *computation cost* which is defined as the amount of computation cycles needed to process it. The time needed to compute this task on a processor is then the product of this computation cost by the cycle time of the processor. An edge corresponds to a precedence constraint and has a *communication cost*. Each edge $A_{i,j}$ carries a label $D_{i,j}$ which specifies the amount of data that N_i passes to N_j. This can be used to compute the time needed to achieve the communication. We suppose that if both tasks are assigned to the same processor, there is no communication cost; otherwise, we pay a cost proportional to $D_{i,j}$, regardless the location of the processors. Moreover, we suppose that we can realize an unlimited number of communications simultaneously.

A task without any input edge is called an *entry* task while a task with no output edge is called an *exit* task. A task is said *ready* when all its predecessors have finished their execution. We denote by $Pred(N_i)$ the set of the immediate predecessors of task N_i and by $Succ(N_i)$ the set of the immediate successors of task N_i.

The target architecture consists of a set of p heterogeneous processors $\mathcal{P} = \{P_1, P_2, \ldots, P_p\}$ so that computation can be overlapped with communication and there is no limitation on the communication links: as soon as a task N_i is completed, data $D_{i,j}$ is sent to all its successors. The execution time of node N_i on processor P_j, given by $E(N_i, P_j)$ (denoted further by $e_{i,j}$) is available at compile time for each node-processor pair.

Our scheduling objective is to minimize the *scheduling length* where all interprocessor communication overheads are included. This scheduling problem is NP-complete even if there is an infinite number of processors available [4], hence the need to rely on heuristics.

3 The algorithms

3.1 Minimum partial completion time static priority (PCT) algorithm

Maheswaran and Siegel present in [7] a dynamic algorithm. Their heuristic refines a given mapping which has already been computed statically. It can be used from scratch to compute a static mapping at compile time by using any basic schedule as input (for example, assume that every task is allocated to the fastest processor). In the following, we assume that we already have a scheduling of our task graph. The first phase or the algorithm assigns ranks to each task. The second phase orders the tasks and uses a minimization criterion to solve the mapping problem.

Consider the first phase of the algorithm. We assign to each task a priority equals to an estimation of the time needed to finish the program. As we already have a scheduling, we take communications into consideration. Letting P_j be the processor to which task N_i is assigned in the given scheduling, we define the priority as follows:

$$priority(N_i) = e_{i,j} + \max_{N_k \in Succ(N_i)} (c_{i,k} + priority(N_k))$$

where $c_{i,k} = D_{i,k} \times \tau$ is the time needed to send the data from the node N_i to the node N_k with bandwidth τ. If both nodes are on the same processor, we have $c_{i,k} = 0$.

In the second phase, we allocate the ready tasks to processors in the order given by their priority. We first compute the node with the highest priority, then the following node and so on. Let P_j be a candidate processor for task N_i. We note $pct(N_i, P_j)$ the partial completion time of task N_i on processor P_j, and $dr(N_i)$ the instant where all the data needed to compute N_i is available by P_j, i.e. the time at which the last data item required by N_i to begin its execution is available by P_j. N_i may be computed after the instant $dr(N_i)$ and as soon as the processor P_j is available. Let $proc(N_k)$ be the processor which executes task N_k, we deduce the following equations for $dr(N_i)$ and $pct(N_i, P_j)$: For an entry task, $pct(N_i, P_j) = e_{i,j}$. For any other task,

$$pct(N_i, P_j) = e_{i,j} + \max(Available[j] + dr(N_i))$$

$$dr(N_i) = \max_{N_k \in Pred(N_i)} (c_{k,i} + pct(N_k, proc(N_k)))$$

where $Available[j]$ is the instant where processor P_j is free to start the execution of a new task. The task N_i is mapped onto the processor which minimizes $pct(N_i, *)$.

THE PCT ALGORITHM
Compute the priority for each task
$ReadyTask \leftarrow$ {Entry tasks}
While $ReadyTask$ is not empty
  Choose n in $ReadyTask$ with the highest priority
  Compute $pct(n, p)$ for all p in $\mathcal{P}$
  Assign n to the processor which minimize $pct(n, p)$
  Update $A[p]$ and $ReadyTask$
End while

3.2 Best imaginary level (BIL) algorithm

Oh and Ha present in [8] a list scheduler. The general idea is to assign a priority, or a static level, to each

node. Then the list scheduler schedules the runnable nodes in the decreasing order of priority, and tries to determine the optimal processor for the selected node.

They define the level of a node N_i as the *Best Imaginary Level* (BIL). The BIL is the length of the critical path beginning with N_i if this node is remapped onto a processor P_j, including the communications and assuming that all the children are perfectly scheduled. Since it is not always possible to schedule the nodes at the best times, we use the term *imaginary*:

$$BIL(N_i, P_j) = e_{i,j} + \max_{N_k \in Succ(N_i)} [min(BIL(N_k, P_j), \min_{p \neq j}(BIL(N_k, P_p) + c_{i,p}))]$$

The BIL of a node is then used to compute a priority order over the nodes.

Once the BIL is computed for each node, we start the second phase which consists of selecting a node, i.e. computing a priority order. To select a node, we adjust the level of a N_i on processor P_j to measure the *Best Imaginary Makespan* (BIM). BIM is defined as follows: $BIM(N_i, P_j) = Available[j] + BIL(N_i, P_j)$. For each node, there exist P different BIM values, one for each processor. Assuming there exist k runnable nodes at a step, we define the priority of a node N_i as the k^{th} smallest BIM value, or the largest finite BIM value if the k^{th} is undefined. The selected node is the one with the highest priority.

Once we have selected a node, we have to find a processor where to map it. If the number of ready nodes k is high, i.e. greater than the number of processors, the execution time becomes more important than the communication overhead, since the communication overhead is likely to be hidden. Therefore, we define the revised BIM as follows :

$$BIM^*(N_i, P_j) = BIM(N_i, P_j) + e_{i,j} \times max(\frac{k}{P} - 1, 0)$$

We select the processor that has the highest revised BIM value. If more than one processor have the same revised BIM value, we select the processor that makes the sum of the revised BIM values of other nodes on the processor maximum. As soon as the task is assigned to a processor, we update the runnable nodes and continue while there exists a ready task.

THE BIM ALGORITHM
Compute $BIL(n, p)$ for all n and p
$ReadyTask \leftarrow$ {Entry tasks}
While $ReadyTask$ is not empty
  Compute BIM for every task in $ReadyTask$
  Choose the node n with the highest priority
  Compute $BIM^*(n, p)$ for all p
  Assign n to the processor p that maximizes $BIM^*(n, p)$
  Update $ReadyTask$
End while

3.3 Heterogeneous earliest finish time (HEFT) and critical path on a processor (CPOP) algorithms

Topcuoglu, Hariri and Wu present in [10] two heuristics. The general idea of both heuristics is the same. In a first phase, they compute a priority on the runnable nodes and select the node with the highest priority. Then, in the second phase, using two different criteria, they determine a processor to which the selected node is mapped. Before studying the two different algorithms, we need some definitions. We define the earliest start time, EST, and the earliest finish time, EFT, of node N_i on processor P_j as follows: $EST(N_i, P_j) = max(A[j], \max_{N_k \in Pred(N_i)}(EFT(N_k, proc(N_k)) + c_{k,i}))$, and $EFT(N_i, P_j) = e_{i,j} + EST(N_i, P_j)$, where $proc(N_k)$ is the processor where N_k is assigned. EST returns the ready time, i.e. the time when all data needed by N_i is available at the host P_j, and when the host P_j itself is available.

In the algorithm, tasks are ranked upward and downward to set the scheduling priorities. The *upward rank* of a task N_i is recursively defined by

$$rank_u(N_i) = \bar{e_i} + \max_{N_k \in Succ(N_i)}(c_{i,k} + rank_u(N_k))$$

where $\bar{e_i} = \sum \frac{e_{i,j}}{p_j}$ is the average execution time of the task N_i over the processors. $rank_u$ is the length of the critical path from N_i to the exit node, including the computation cost of the node itself. Similarly, the *downward rank* of a task N_i is recursively defined by

$$rank_d(N_i) = \max_{N_k \in Pred(N_i)}(c_{k,i} + \bar{e_k} + rank_d(N_k))$$

The $rank_d$ is the longest distance from the start node to the node N_i, excluding the computation cost of the node itself.

3.3.1 HEFT

To set priority to a task N_i, the HEFT algorithm uses the upward rank value of the task. Ready tasks are sorted with respect to decreasing $rank_u$ values. If two nodes to be scheduled have the same priority, one of them is selected randomly.

The HEFT algorithm uses the EFT value to select the processor for the selected task. It is natural to

consider the EFT value to select a processor. Indeed,when all nodes in the graph have been scheduled, the schedule length is the earliest finish time of the exit node. We assign node N_i to the processor p which minimizes the value of $EFT(N_i, p)$.

THE HEFT ALGORITHM
Compute $rank_u$ for all nodes
$ReadyTask \leftarrow$ {Entry tasks}
While $ReadyTask$ is not empty
  Select the task n with highest priority
  Assign the task n to the processor p that minimizes the EFT value of n
  Update EST values and $ReadyTask$
End while

3.3.2 CPOP

The CPOP algorithm uses $rank_u(n) + rank_d(n)$ to assign the node priority. As previously, we select the node with the highest priority, i.e. we first consider the tasks that belong to the critical path. A task is on the critical path if its value of $rank_u + rank_d$ is equal to the value of $rank_u(N_s)$ where N_s is the start node. The critical-path-processor (CPP) is the one that minimizes the length of the critical path. If the current task is on the critical path, then it is assigned to the CPP, otherwise it is assigned to the processor that minimizes the EFT. The time needed to compute the tasks along the critical path is a lower bound of the execution time. Hence it appears to be a good criterion in order to (try to) minimize the length of the critical path. The CPP is often the fastest processor.

THE CPOP ALGORITHM
Compute $rank_u$ and $rank_d$ for all nodes
$ReadyTask \leftarrow$ {Entry tasks}
While $ReadyTask$ is not empty
  Select the task n with highest priority
  If n is on the critical processor
    Assign n to the CPP
  Else
    Assign the task n to the processor p that minimizes the EFT value of n
  Update EST values and $ReadyTask$
End while

3.4 Generalized dynamic level (GDL) algorithm

Sih and Lee propose in [9] a compile-time scheduling heuristic for heterogeneous networks called GDL. As in the previous algorithm, the GDL scheduler computes the critical path in a heterogeneous system. Contrarily to the $CPOP$ algorithm, GDL defines the assumed execution time of node N_i denoted $e^*(N_i)$ as the median execution time of the node over all processors.

We define the static level of a node N_i, $SL(N_i)$ as the largest sum of execution times along any directed path from N_i to an exit node of the graph. $SL(N_i)$ can be easily computed recursively. To take the difference of processors speeds into account, we introduce the quantity $\Delta(N_i, P_j) = e^*(N_i) - e_{i,j}$. Then we introduce a dynamic level $DL(N_i, P_j)$ which reflects how well node N_i and processor P_j are matched. This quantity will be re-evaluated at each step of the algorithm to take next decisions into account: $DL(N_i, P_j) = SL(N_i) - EST(N_i, P_j) + \Delta(N_i, P_j)$. The term $EST(N_i, P_j)$ is the earliest start time defined in the same way as for the $HEFT$ and the $CPOP$ algorithms. This algorithm is very simple. While there exists a ready task, we select the node and the processor which maximize the expression DL.

Descendant consideration Although $DL(N_i, P_j)$ indicates how well node N_i is matched with processor P_j, it fails to consider how well the descendants of N_i are matched with P_j. For each node N_i we note $D(N_i)$ the descendant to which N_i passes the most data and $d(N_i, D(N_i))$ the amount of data passed between them. We then define $F(N_i, D(N_i), P_j)$ to indicate how quickly $D(N_i)$ can be completed on any other processor if N_i is executed on P_j. Then $F(N_i, D(N_i), P_j) = \tau \times d(N_i, D(N_i)) + \min_{k \neq j} E(D(N_i), P_k)$. This is a lower bound of the time necessary to finish the execution of $D(N_i)$ on any processor other than P_j. We then define a descendant consideration term as $DC(N_i, P_j) = e^*(D(N_i)) - min\{E(D(N_i), P_j), F(N_i, D(N_i), P_j)\}$.

Resource scarcity We generally fail to consider how important it is for two nodes to obtain the same processor. To characterize this resource scarcity cost, we first define the preferred processor of a node, i.e the processor that maximizes its dynamic level. We then define the cost of not scheduling N_i onto its preferred processor: $C(N_i) = DL(N_i, P_j^*) - \max_{k \neq j^*} DL(N_i, P_k)$, where j^* is the index of the preferred processor of N_i. If the cost is zero, N_i will still have at least one processor with which it can obtain the same dynamic level.

Generalized dynamic level By taking into account the descendant consideration and the resource scarcity we can now define a generalized dynamic level $GDL(N_i, P_j) = DL(N_i, P_j) + DC(N_i, P_j) + C(N_i)$. The algorithm is unchanged. We select among the runnable tasks the task and the processor which

maximize the GDL.

THE GDL ALGORITHM
Compute $e^*(n)$ and $D(n)$ for all nodes n
Compute $SL(n)$ for all nodes n
$ReadyTask \leftarrow$ {Entry tasks}
While $ReadyTask$ is not empty
  Compute $GDL(n,p)$ for every node n in $ReadyTask$
    and every processor p
  Select the pair (n,p) that maximizes GDL
  Update $ReadyTask$
End while

4. The Iso-Level Heterogeneous Allocation (ILHA) algorithm

The main idea of the algorithm that we propose is to perfectly balance the load between those computations that can be performed in parallel. First we sketch a simple method to balance a set of n tasks on p heterogeneous processors, then we explain how to split the task graph into task levels which we assign to processors using the load-balancing algorithm. Finally we outline the *ILHA* algorithm.

Load balancing algorithm Given n independent chunks of computations, each requiring the same amount of work, how can we assign these chunks to p processors so that the load is best balanced ? Intuitively, the load of P_i should be inversely proportional to its cycle-time t_i, i.e. P_i should receive $c_i = \frac{\frac{1}{t_i}}{\sum_{j=1}^{p} \frac{1}{t_j}} \times n$ chunks. This strategy leads to a perfect load balance when n is a multiple of $C = lcm(t_1, t_2, \cdots, t_p) \sum_{j=1}^{p} \frac{1}{t_j}$, a quantity that may be very large. For the general case, the following algorithm provides the best solution [3]:

OPTIMAL DISTRIBUTION
$forall\, i \in \{1, \ldots, p\}$, $c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times n \right\rfloor$.
for $m = c_1 + c_2 + \ldots + c_p$ **to** n
  find $k \in \{1, \ldots, p\}$ s.t.
    $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1))\}$
  $c_k = c_k + 1$

4.1. First version

Iso-level splitting We split a task graph into *levels* made up of independent tasks, by considering the tasks that will be ready at the same time-step. In other words, two tasks belong to the same *level* if they have the same top-level, using the terminology of [11]. This is done by a traversal of the graph. Initially, the 0-level is composed of the entry tasks. The $(i+1)$-th level groups the tasks that are ready when the i-th level is achieved.

The *ILHA*-0 algorithm A first version of the *ILHA* algorithm is the following: we traverse the task graph to split it into levels made of independent tasks. We compute the number of tasks that we allocate to each processor using the load-balancing algorithm. Once this is done, we have to determine exactly which task is given to each processor. The criteria is to minimize the communication costs. So for each task of the level, we consider its predecessors. If they are all allocated to the same processor, we try to allocate the task to the same processor (i.e. if the processor may receive another task), otherwise, we allocate the task to the fastest processor that is not yet saturated (able to receive new tasks according to the load-balancing strategy). This simple strategy leads to the following algorithm:

THE ILHA-0 ALGORITHM
$ReadyTask \leftarrow$ {Entry tasks}
While $ReadyTask$ is not empty
  Compute the optimal distribution with $||ReadyTask||$ tasks
  For each task t of $ReadyTask$
    If all predecessors of t are on p and p is free
      Assign t to p
  For each task t of $ReadyTask$ not yet assigned
    Assign t to the first free processor
  Update $ReadyTask$
End while

4.2. Refined version

In the previous version of the ILHA algorithm, we process all the ready tasks at each step. In some cases, it would be better to take into account the bottom level of the ready tasks and to consider first the tasks on a critical path. To this purpose, we sort the ready tasks according to their bottom level. Then, we introduce a parameter B, the maximal number of ready tasks that will be considered at each step. We consider those B tasks with the higher bottom levels and we allocate them using the load balancing algorithm. Then, we update the set of ready tasks (indeed some new tasks may have become ready) and we re-sort them according to their bottom level. Thus, we expect that the tasks on a critical path will be processed as soon as possible. Unfortunately, we face a tradeoff for choosing an appropriate value for B. On one hand if B is large, it will

be possible to better balance the load and minimize the communications. On the other hand, a small value of B will enable us to process the tasks on the critical path sooner. Of course B must be at least equal to the number of processors, otherwise some processors will be kept idle. The choice of B will be discussed furthermore in the Section 5.

We obtain the final version of the $ILHA$ algorithm:

THE ILHA ALGORITHM
Compute the bottom level of each task
$ReadyTask \leftarrow$ {Entry tasks} sorted by decreasing value of their bottom level
While $ReadyTask$ is not empty
  Take the B first tasks of the $ReadyTask$
  Compute the optimal distribution with B tasks
  For each task t of $ReadyTask$
    If all predecessors of t are on p and p is free
      Assign t to p
  For each task t of $ReadyTask$ not yet assigned
    Assign t to the first free processor
  Update the list of $ReadyTask$ by inserting the new ready tasks in the sorted list
End while

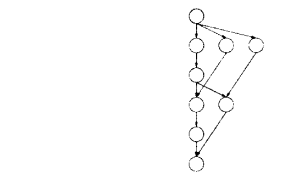
5. Experiments

5.1. Testbeds

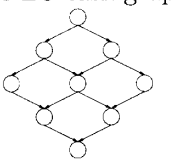
In order to compare the different algorithms, we consider six classical kernels representing various types of parallel algorithms. The selected task graphs are *LU decomposition* ("LU"), *Laplace equation solver* ("LAPLACE"), a *stencil algorithm* ("STENCIL"), a *fork-join graph* ("Fork-Join"), *Doolittle reduction*("DOOLITTLE") and *LDM^t decomposition* ("LDMt"). Miniature versions of each task graph are shown in Figure 1.

For the LAPLACE, STENCIL, and FORK-JOIN testbeds, all tasks have same weight, which we normalize to 1. For the linear algebra testbeds, i.e. LU, DOOLITTLE and LDMt, the situation is more complicated, because the amount of work to be done at each step of the algorithm is not constant (see [6, 5]). For the LU kernel, the weight of a task at level k is $N - k$, where N is the size of the graph. For the DOOLITTLE and LDMt kernels, the the weight of a task at level k is k, where k varies from 1 to N, the size of the graph.

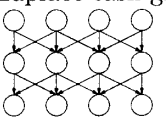
For each testbed, we let the communication costs be proportional to the task weights: indeed in each kernel, we always communicate the data that has just been updated. In other words, the communication cost from a task v to a task v' is equal to c times the weight of v,



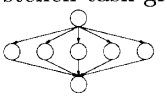
The LU task graph



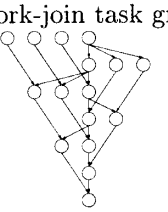
The Laplace task graph



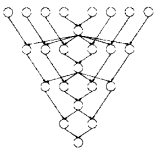
The stencil task graph



The fork-join task graph



The DOOLITTLE reduction task graph



The LDM^t decomposition task graph

Figure 1. The different task graphs

where c is a parameter that models the communication-to-computation ratio of the target platform. For each kernel, we perform a first simulation with small communication costs, i.e. $c = 1$, as well as a second simulation with large communication costs, i.e. $c = 10$.

5.2. Results

A full set of results is available in [2], where we consider two sets of experiments. In the first set, we use only 3 processors with respective cycle times 6, 10 and 15. Remember that the time to execute a task is the product of its weight by the processor cycle-time. In the second set, we use 10 processors: five processors with cycle time 6, three processors with cycle time 10, and two processors with cycle time 15. Also, for each

kernel, we perform a first simulation with small communication costs, i.e. $c = 1$, as well as a second simulation with large communication costs, i.e. $c = 10$.

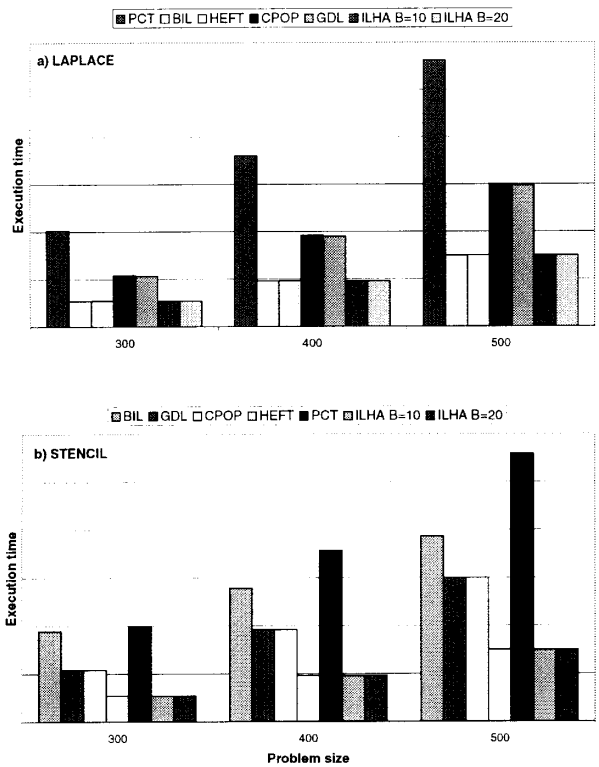


Figure 2. Comparison of the heuristics with 3 processors for LAPLACE and STENCIL.

Due to the lack of space, we only report one representative subset of the whole experiments, with 3 processors and $c = 10$. The three processor speeds are strongly heterogeneous: it turns out that many heuristics fail to be more efficient than a sequential execution. This should not be too surprising: if we were able to achieve a full utilization of the computing resources, i.e. an efficiency equal to one, we would improve the sequential time (using the fastest processor alone) by a factor $6 * (\frac{1}{6} + \frac{1}{10} + \frac{1}{15}) = 2$ (which represents the maximum number of tasks that we may compute during a cycle time).

In Figures 2 to 4, we show the expected execution time of the five heuristics (BIL, GDL, CPOP, HEFT and PCT) and of two instances of the ILHA heuristic with different values for B, namely $B = 10$ and $B = 20$. Results for LDMt are similar to those for DOOLITTLE. See [2] for a detailed analysis of these experiments; here we only give a few synthetical com-

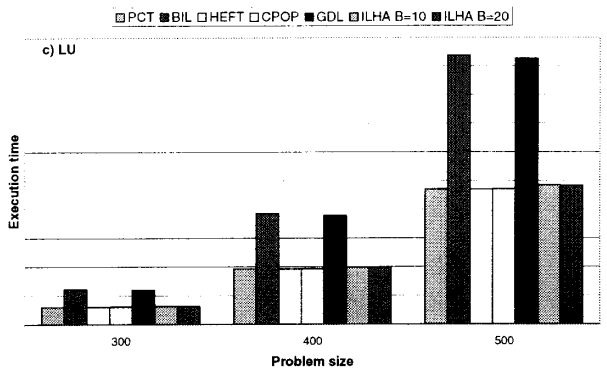


Figure 3. Comparison of the heuristics with 3 processors for LU.

ments:
(i) The CPOP heuristic is not efficient for regular problems, where each node of the task graph is on a critical path: this comment holds for LAPLACE, STENCIL, and FORK-JOIN. However, CPOP leads to an efficient scheduling for LU, DOOLITTLE and LDMt.
(ii) The BIL heuristic gives good results for LAPLACE, FORK-JOIN, DOOLITTLE and LDM^t, but relatively bad ones for STENCIL and LU.
(iii) PCT is efficient for LU, DOOLITTLE, LDMt and FORK-JOIN but it is dramatically bad for LAPLACE and STENCIL.
(iv) GDL never gives good results, but it is never the worst heuristic. (v) The value of the B parameter of ILHA is important for LU decomposition: the ILHA version with $B = 10$ is quite better than the ILHA version with $B = 20$, which can be explained as follows: the shape of the LU task graph is such that the critical path must be executed rapidly, hence the need for a smaller value of B. (v) ILHA and HEFT give very satisfactory results.

It appears that HEFT and ILHA heuristics give very close results. However, if we consider the number of communications generated by both heuristics, we point out that our algorithm involves significantly fewer communications: see Table 1. In the classical scheduling model (as outlined in Section 2), we assume that the number of communications that can be performed at the same time is not limited. While this model is widely used in the scheduling community, it is not realistic in practice. The interest of ILHA may well become more important in actual implementations on heterogeneous networks of workstations, where communication resources often turn out to be the limiting factor to achieving good performances.

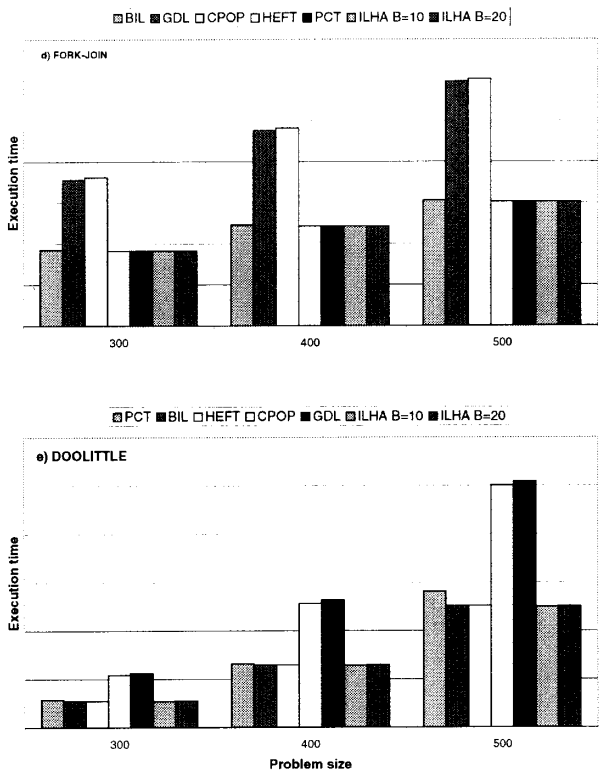


Figure 4. Comparison of the heuristics with 3 processors for FORK-JOIN and DOOLITTLE.

6. Conclusion

In this paper we have dealt with five different heuristics from the literature to solve the task scheduling problem on heterogeneous platforms: PCT, BIL, HEFT, CPOP and GDL. It appears that HEFT is the only heuristic giving good results for all the six testbeds used in our comparison.

We have introduced a new heuristic, ILHA, which seems very promising for the following main reasons: (i) ILHA is always the best heuristic with HEFT, (ii) ILHA only requires a traversal of the task graph, so its low complexity makes its suitable to process huge task graphs in a reasonable time. (iii) ILHA generates very few communications, which renders is quite attractive for more realistic models where communication contentions are taken into account. Further work must be devoted to the tuning of the B parameter in ILHA, which results from a trade-off between fast execution of the critical path (small value of B) and better load-balancing (large value of B).

Problem	HEFT	ILHA B=10	ILHA B=100
STENCIL	17129	5742	396
LU	11244	4494	2508
LAPLACE	17129	8669	392
FORKJOIN	100	100	100
DOOLITTLE	13727	10585	5018
LDMt	14387	10494	5272

Table 1. Number of communications with HEFT and ILHA

References

[1] B.A.Shirazi, A. Hurson, and K. Kavi. *Scheduling and load balancing in parallel and distributed systems.* IEEE Computer Science Press, 1995.

[2] O. Beaumont, V. Boudet, and Y. Robert. The iso-level scheduling heuristic for heterogeneous processors. Technical Report 2001-22, LIP, ENS Lyon, France, May 2001. Available at www.ens-lyon.fr/ ~yrobert.

[3] V. Boudet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). In H. R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99).* CSREA Press, 1999. Extended version available as LIP Technical Report RR-99-17.

[4] P. Chrétienne, E. C. Jr., J. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications.* John Wiley and Sons, 1995.

[5] M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystram. Parallel Gaussian elimination on a MIMD computer. *Parallel Computing*, 6:275–296, 1988.

[6] G. H. Golub and C. F. V. Loan. *Matrix computations.* Johns Hopkins, 2 edition, 1989.

[7] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Seventh Heterogeneous Computing Workshop.* IEEE Computer Society Press, 1998.

[8] H. Oh and S. Ha. A static scheduling heuristic for heterogeneous processors. In *Proceedings of Europar'96*, volume 1123 of *LNCS*, Lyon, France, Aug. 1996. Springer Verlag.

[9] G. Sih and E. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[10] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Eighth Heterogeneous Computing Workshop.* IEEE Computer Society Press, 1999.

[11] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel and Distributed Systems*, 5(9):951–967, 1994.