Heterogeneous task scheduling : a survey

Vincent Boudet

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

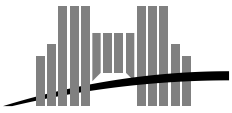
SPI

Heterogenous task scheduling : a survey

Vincent Boudet

February 2001

Research Report N° 2001-07



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

Heterogenous task scheduling : a survey

Vincent Boudet

February 2001

Abstract

Scheduling computation 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 survey here five low-complexity heuristics for heterogeneous platforms, the Best Imaginary Level (BIL), the Generalized Dynamic Level (GDL), the Critical-Path-on-a-Processor (CPOP), the Heterogeneous Earliest Finish Time (HEFT) and the Partial Completion Time (PCT) algorithms. These five heuristics aim at scheduling directed acyclic weighted task graphs on a bounded number of heterogeneous processors. We compare the performances of the heuristics using four classical testbeds.

Keywords: scheduling, task graphs, high-performance, heterogeneous platforms, different-speed processors, mapping.

Résumé

Ordonnancer des tâches de calculs sur des processeurs est une nécessité pour du calcul haute performance. Bien qu'un grand nombre d'heuristiques d'ordonnancement existe dans la littérature, la plupart d'entre elles ne visent que des ressources homogènes. Nous présentons ici cinq heuristiques pour des architectures hérogènes : le "Best Imaginary Level" (BIL), le "Generalized Dynamic Level" (GDL), le "Critical-Path-on-a-Processor" (CPOP), le "Heterogeneous Earliest Finish Time" (HEFT) et le "Partial Completion Time" (PCT). Ces cinq algorithmes ont pour but d'ordonnancer des graphes de tâches acycliques et pondérés sur un nombre limité de processeurs hérogènes. Nous comparons les performances de ces heuristiques sur quatre problèmes classiques.

Mots-clés: ordonnancement, graphes de tâches, calcul haute performance, plateforme hétérogène, processeurs de vitesses différentes, distribution.

1 Introduction

An efficient scheduling for 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 task precedence requirements are satisfied and a minimum schedule length is provided. Since the scheduling problem is NP-hard in the strong sense, many research efforts have proposed various heuristics for this problem.

Although a wide variety of different approaches are used to solve the task scheduling problem, most of them target only homogeneous processors. Scheduling methods that are suitable for homogeneous environments may well not be efficient for heterogeneous domains.

In this paper, we will focus on heuristics. We will survey five of them for the task scheduling problem : the *minimum Partial Completion Time static priority* algorithm (PCT), the *Best Imaginary Level* (BIL) algorithm, the *heterogeneous earliest finish time* (HEFT), the *critical path on a processor* (CPOP) and the *generalized dynamic level* (GDL) algorithm. In section 4, we present the four classical testbeds that we use to compare the different heuristics. We outline and comment the results in section 5. Finally we give concluding remarks in section 6.

2 Preliminaries

For each task scheduling algorithm, the input is a directed acyclic graph $\mathcal{G} = (\mathcal{N}, \mathcal{A})$, 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 cycle needed to achieve 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 task dependencies (communication messages or precedence constaints) 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 two tasks are assigned to the same processor, there is no communication penalty. Moreover, we suppose that we can realize many communications at the same time. A task without any input edge is called an *entry* task while a task with no output task is called an *exist* 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 the task N_i and $Succ(N_i)$ the set of the immediate successors of the task N_i.

The target achitecture contains a set of heterogeneous processors $\mathcal{P} = \{P_k : k = 1, \cdots, p\}$ 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 succesors. 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 in the strong sense even if there are an infinite number of processors available (see [1]). Hence we will rely on heuristics.

3 The algorithms

3.1 Minimum partial completion time static priority algorithm (PCT)

M. Maheswaran and H. J. Siegel present in [2] a dynamic algorithm. Their heuristic refines a given mapping computed statically but it can be used from scratch to compute a static mapping at compile time by using a basic schedule (for example, every task is allocated to the fastest processor) as input. So we assume that we already have a scheduling of our graph.

The first phase or the algorithm assigns ranks to each task. The second phase orders the tasks and uses a minimization criteria 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 may take the communications into consideration. Let P_j the processor to which the 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 the bandwith τ. If the two 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 the task N_i on the processor P_j and $dr(N_i)$ the instant where all the data needed to compute N_i is available on P_j, i.e. the time at which the last data item required by N_i to begin its execution arrives at P_j. N_i may be computed after the instant $dr(N_i)$ and after the moment where the processor P_j is available. So we can deduce the following equations for $dr(N_i)$ and $pct(N_i, P_j)$. We note $proc(N_k)$ the processor which is assigned the task N_k.

For an entry task, we have :

$$pct(N_i, P_j) = e_{i,j}$$

and for any other task, we have :

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

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

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

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

3.2 Best imaginary level (BIL)

Hyunok Oh and Soonhoi Ha present in [3] a list scheduler. The global 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. Then they try to determine the optimal processor for the selected node.

They define the level of a node N as the *Best Imaginary Level*, BIL. The BIL is the length of the critical path beginning with N if this node is remapped onto P including the communications 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 of *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 on the nodes.

Once the BIL is computed for each node, we start the second phase which consists on selecting a node, i.e. computing a priority order. To select a node, we adjust the level of a node 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 exists 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 a node selected, we have to find a processor where to map it. If the number of ready nodes k is high, i.e. greatest than the number of processors, the execution time becomes the more important factor 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.

```
1: The BIM algorithm
2: Compute BIL(n, p) for all n and p
3: ReadyTask ← {Entry tasks}
4: While ReadyTask is not empty
5:     Compute BIM for every task in ReadyTask
6:     Choose the node n with the highest priority
7:     Compute BIM*(n, p) for all p
8:     Assign n to the processor p that maximizes BIM*(n, p)
9:     Update ReadyTask
10:End while
```

3.3 Heterogeneous earliest finish time and critical path on a processor

H. Topcuoglu, S. Hariri and M.-Y. Wu present in [5] two heuristics. The global idea of the two heuristics is the same. In a first phase, we compute a priority on the runnable nodes and we select the node with the highest priority. Then, in the second phase, using two different criteria, we select a processor to map the selected node.

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 :

$$EST(N_i, P_j) = max(A[j], \max_{N_k \in Pred(N_i)}(EFT(N_k, proc(N_k)) + c_{k,i}))$$

$$EFT(N_i, P_j) = e_{i,j} + EST(N_i, P_j)$$

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 has arrived at the host P_j and when the host P_j 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 heterogeneous earliest finish time algorithm uses the upward rank value of the task. We sort the ready tasks with respect to the decreasing order of the $rank_u$ 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, after all nodes in the graph are scheduled, the schedule length will be the earliest finish time of the exit node. We assign the node N_i to the processor p which minimize the value of $EFT(N_i, p)$.

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

3.3.2 CPOP

The critical-path-on-a-processor algorithm uses $rank_u(n) + rank_d(n)$ to assign the node priority. As previously, we select the node with the highest priority. That is to say that 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, it is assigned to the CPP, otherwise is 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. So it appears to be a good criteria to try to minimize the length of the critical path. The CPP is often the fastest processor.

```
THE CPOP ALGORITHM
1: Compute rank_u and rank_d for all nodes
2: ReadyTask ← {Entry tasks}
3: While ReadyTask is not empty
4:     Select the task n with highest priority
5:     If n is on the critical processor
6:         Assign n to the CPP
7:     Else
8:         Assign the task n to the processor p that
           minimizes the EFT value of n
9:     Update EST values and ReadyTask
10:End while
```

3.4 Generalized dynamic level

Gilbert C. Sih and Edward A. Lee propose in [4] a compile time scheduling heuristic for heterogeneous network called GDL. As in the previous algorithm, the GDL scheduler computes the critical path in a heterogeneous system. Contrary 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 then 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 account of the difference of processors speed, we introduce the quantity :

$$\Delta(N_i, P_j) = e^*(N_i) - e_{i,j}$$

We introduce then 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 reevaluated at each step of the algorithm to take into account of the next decisions.

$$DL(N_i, P_j) = SL(N_i) - EST(N_i, P_j) + \Delta(N_i, P_j)$$

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.

The algorithm is very simple. While there exist a ready task, we select the node and the processor which maximize the expression DL.

3.4.1 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.

$$F(N_i, D(N_i), P_j) = \tau \times d(N_i, D(N_i)) + \min_{k \neq j} E(D(N_i), P_k)$$

This is a lower bound on 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

$$DC(N_i, P_j) = e^*(D(N_i)) - min\{E(D(N_i), P_j), F(N_i, D(N_i), P_j)\}$$

3.4.2 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 in not scheduling N_i on 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.

3.4.3 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 inchanged. We selected among the runnable tasks the task and the processor which maximize the GDL.

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

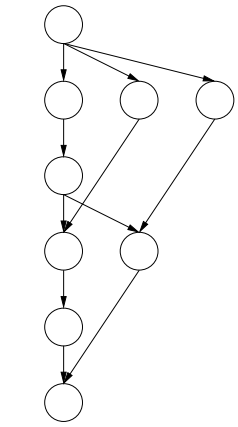
4 Experiments

To compare the different algorithms, we consider four classical kernels representing various types of parallel algorithms. The selected task grpahs are *LU decomposition* ("LU"), *Laplace equation solver* ("LAPLACE"), a *stencil algorithm* ("STENCIL") and a *fork-join graph* ("Fork-Join"). Miniature versions of each task graph are shown in Figure 1.

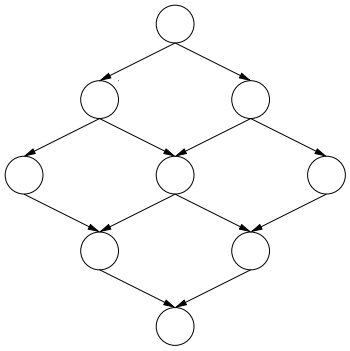
A communication between two different processors has a cost of 1. In the problem of *LU* decomposition, a node of level k has a cost egal to $N - k$ where N is the size of the graph. In the other problems the cost of a task is 1.

For each of these problems, we varied the size of the graph, the communication-computation ratio by modifying the speed of the processors, the relative speed of the processor and the number of processors.

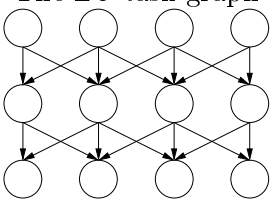
The speed of the k^{th} processor is given by the formulae $\bar{s} + k \times \sigma$ where σ is the difference of speed between two consecutive processors. To study the influence of the ratio communication-computation we varied the value of $\bar{s}$. To measure the impact of the difference of speed between processors, we modify the value of σ.



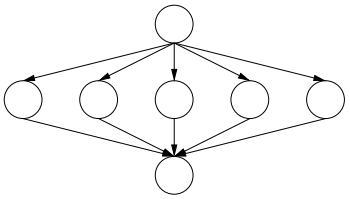
The LU task graph



The Laplace task graph



The stencil task graph



The fork-join task graph

Figure 1: The different task graphs

5 Results

5.1 *LAPLACE* problem

Consider the results shown in Figures 2 to 6. In the first three graphics, we compare the different algorithms by varying the communication-computation ratio. The speed in the different graphs

are $[1, 2, 3]$ for the first case, $[10, 11, 12]$ and $[100, 101, 102]$ in the second and last case. The communication cost remains equal to 1. The BIL algorithm has very good results. It balances very well the work of the different processors. The PCT and the GDL algorithm are very close. They badly balance the load of the processors. Indeed, the most important part of the work is given to the fastest processor and the other processors have a small amount of computations to execute. Finally, $CPOP$ and $HEFT$ have very bad results. Because of the non-sense of a critical path for heterogeneous resources, they give all the computations to the same processor. Indeed, all the nodes of the $LAPLACE$ problem are on a critical path. When the communication-computation ratio becomes very important, only the BIL algorithm remains good. As the other algorithms give most computations to a single processor, they tend to the makespan of a sequential schedule. If we consider Figure 5, we see that for the BIL algorithm the results are worse. Indeed, as the average speed is higher, the expected execution time is more important because the load is quite well balanced. However, for the PCT algorithm the results become very bad. As the load is badly balanced, the slowest processors keep the fastest processor idle and so the expected execution time increases. For the other three algorithms, as the most part of the computations is given to the fastest processors, there is no difference between the two expected execution times. Finally, if we study the impact of the number of processors (Figure 6), we see that the $CPOP$ and $HEFT$ algorithm give most computations to the fastest processor and so the expected execution time doed not decrease with the number of processors. The expected execution time given by the other heuristics does decrease with the number of processors. We point out that in the case of the BIL algorithm, we obtain a minima with 50 processors. After this value, the expected execution time does not decrease.

5.2 LU problem

The results of the experiments on the LU problem are shown in Figures 7 to 11. A problem of size 100 consists of a graph with around 5000 nodes. Contrary to the $LAPLACE$ problem, there exists only one critical path. So we can assume that the $HEFT$ and $CPOP$ algorithms will be better than in the case of the $LAPLACE$ graph. As expected, these 2 heuristics present efficient results. With a poor or a high communication-computation ratio, they obtain the best results for this problem. The PCT algorithm has very similar results. For these three algorithms the research of a good scheduling is "guided" by the longest path in the graph. As they try to minimize the time needed to compute this path, the finally find a schedule with a good expected execution time. The GDL algorithm has very poor results. With only 3 processors, the schedule given by this heuristics leads to a bad expected execution time. And this remark remains true with a poor or a high communication-computation ratio. The performances of this algorithm starts to be interesting when the number of processor becomes very large. With around 95 processors, the results are equivalent to the results obtained by the other algorithms. For a higher number of processors, this heuristic is better than the others. The last heuristic, BIL, which was good for the $LAPLACE$ problem has surprising results. Its performances decrease with the size of the problem. This effect is more important when the computation-communication ratio is high. For a poor ratio (Figure 7 and 8), BIL obtains results very close to the best results. We could note a difference for a problem of size 100 but the difference is not very important. For a high ratio (Figure 9 and 10), the difference appears for a problem of size 50. And for a size 100 problem, the results are not efficient. Finally, as a last remark on the LU problem, we could note that the heuristics uses all the available resources as shown in figure 11. We obtain 5 curves for the execution time which all decrease with the number of processors.

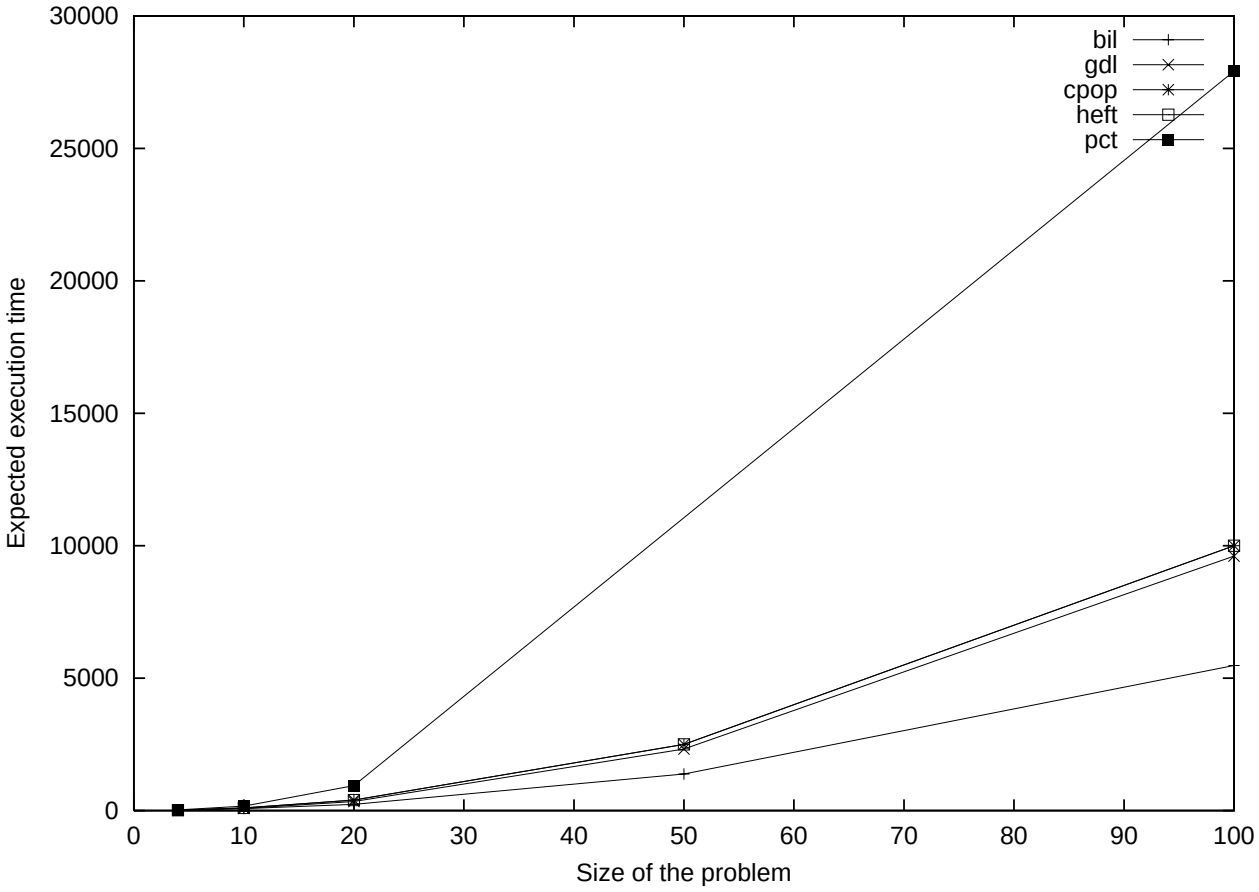


Figure 2: Comparison of the different heuristics for LAPLACE problem with speed equals to 1,2 and 3

5.3 *STENCIL* problem

The results of the experiments on the *STENCIL* problem are shown in Figures 12 to 16. As in the *LAPLACE* problem, each node belongs to a critical path. So, *HEFT* and *CPOP* assign each node to the fastest processor. We obtain the makespan of a sequential schedule. As we can see in Figures 12 and 13 where the communication-computation ratio is low *BIL*, *GDL* and *PCT* have poor results. Indeed, their expected execution time are worse than for sequential execution. In a same way, when the difference between processors speed is large (Figure 15), these 3 heuristics have not efficient results and *HEFT* and *CPOP* obtain better results. These bad results are explained by the very constrained nature of the *STENCIL* problem. However, when the communication-computation ratio is high and when the speed of the processors are close, the 3 heuristics are better than *HEFT* and *CPOP* but the gain is not very important. To see the impact of the number of processors, we choose the case where the communication-computation ratio is high and where the speed of the processors are close (Figure 16). As for the *LAPLACE* problem, the number of proceesors does not affect the results for *HEFT* and *CPOP*. The *BIL* algorithm present quite better results but it is not a very efficient heuristic for the *STENCIL* problem. The *PCT* and the *BIL* obtain the better results. However, to obtain a speed-up of 2 they need around 30 processors.

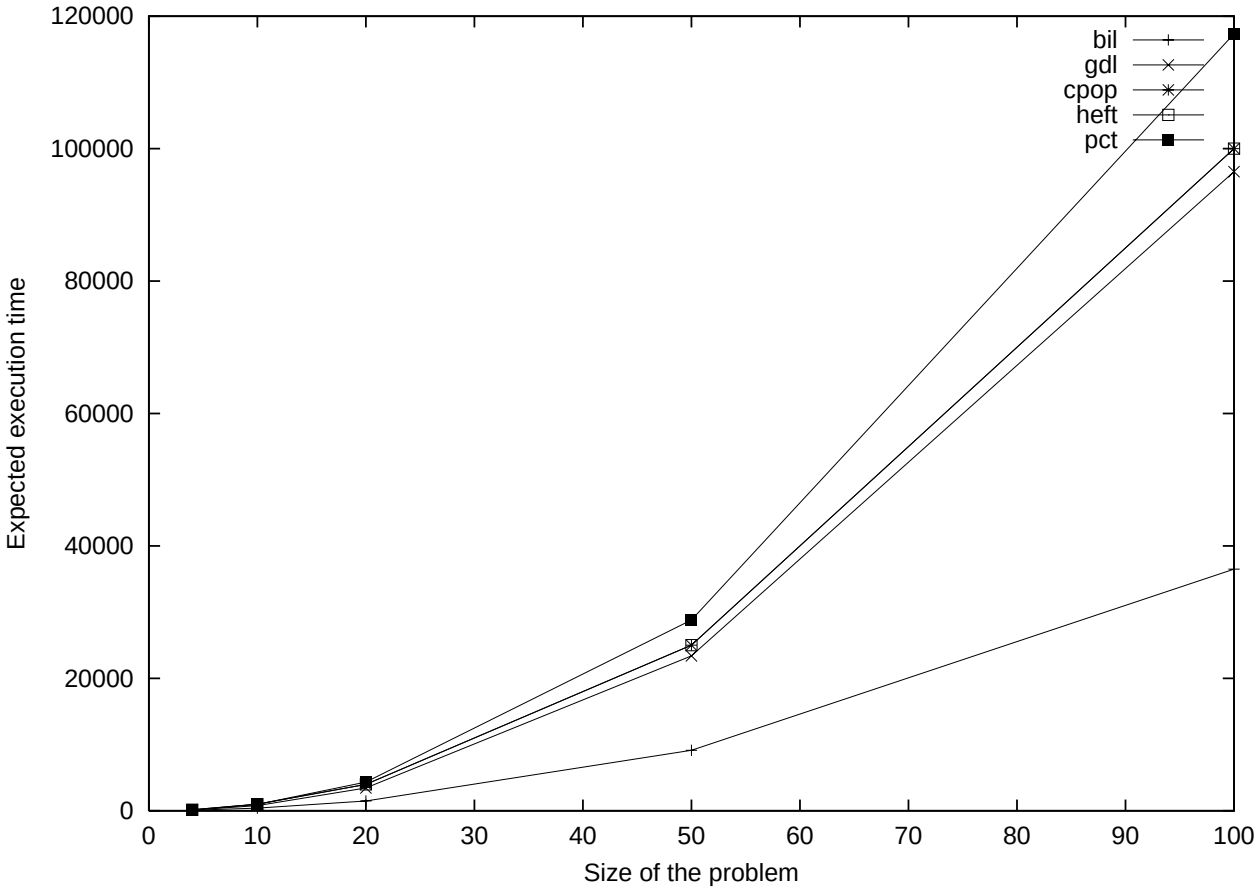


Figure 3: Comparison of the different heuristics for LAPLACE problem with speed equals to 10,11 and 12

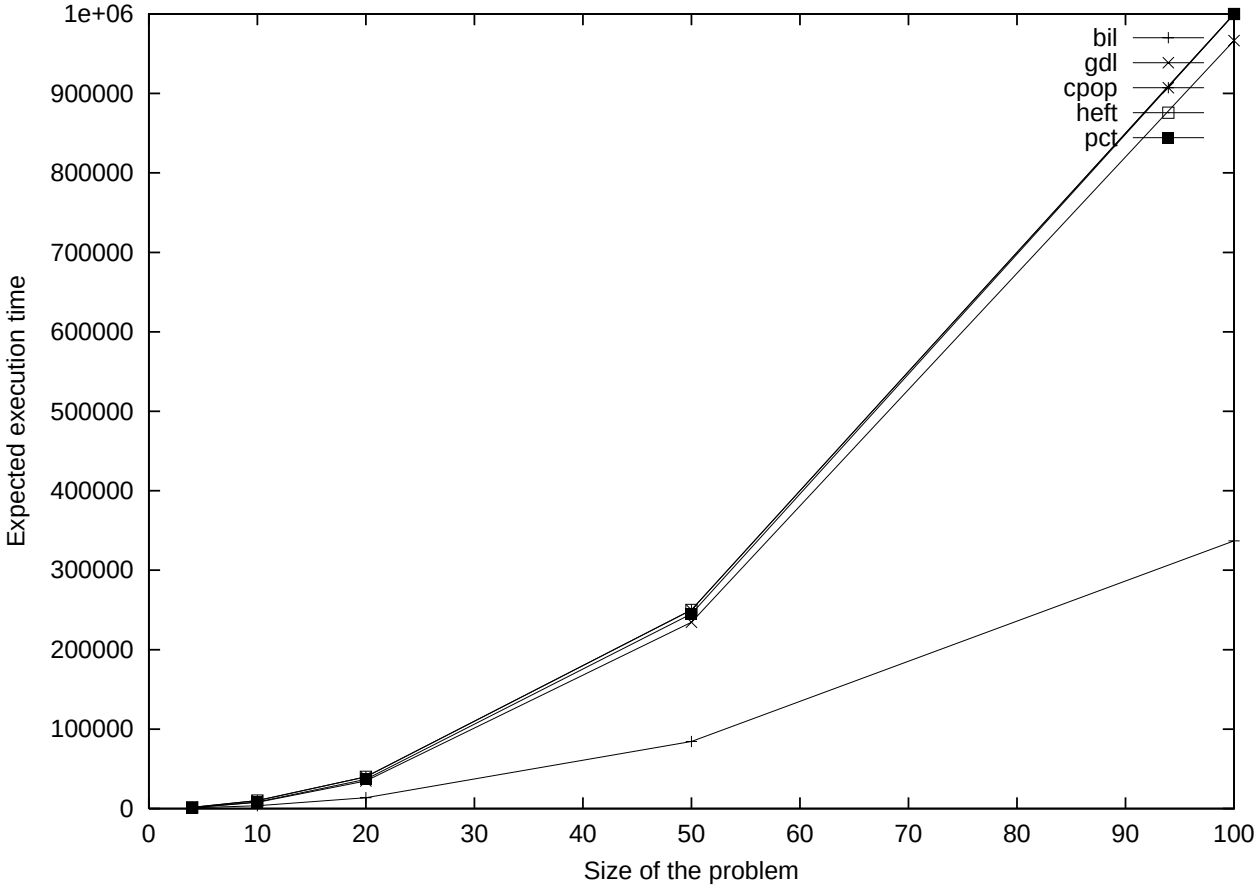


Figure 4: Comparison of the different heuristics for LAPLACE problem with speed equals to 100,101 and 102

5.4 $Fork - Join$ problem

Consider the results shown in Figures 17 to 21. The main fact is that the communication-computation ratio and the standard deviation of the speed does not affect the results. Indeed, the first four graphics present very similar results. The BIL and the PCT heuristics have the best results. They balance very well the work of the different processors by using all the ressources of the architecture. Once again, $HEFT$ and $CPOP$ give all the computations to a single processor. They obtain the makespan of a sequential schedule. The GDL algorithm present better results than $HEFT$ and $CPOP$ but the gain is very low. However, when the number of processor increases, the results of GDL are better but remain worse than those of PCT and BIL. Indeed, the expected execution time given by PCT or BIL is around 3 times better than for the GDL heuristic. Once again, the number of processors does not affect the results of $CPOP$ and $HEFT$.

6 Conclusion

In this paper we focused on different heuristics for the task scheduling problem on heterogeneous platforms which are : the *minimum Partial Completion Time static priority* algorithm (PCT), the *Best Imaginary Level* (BIL) algorithm, the *Heterogeneous Earliest Finish Time* (HEFT), the

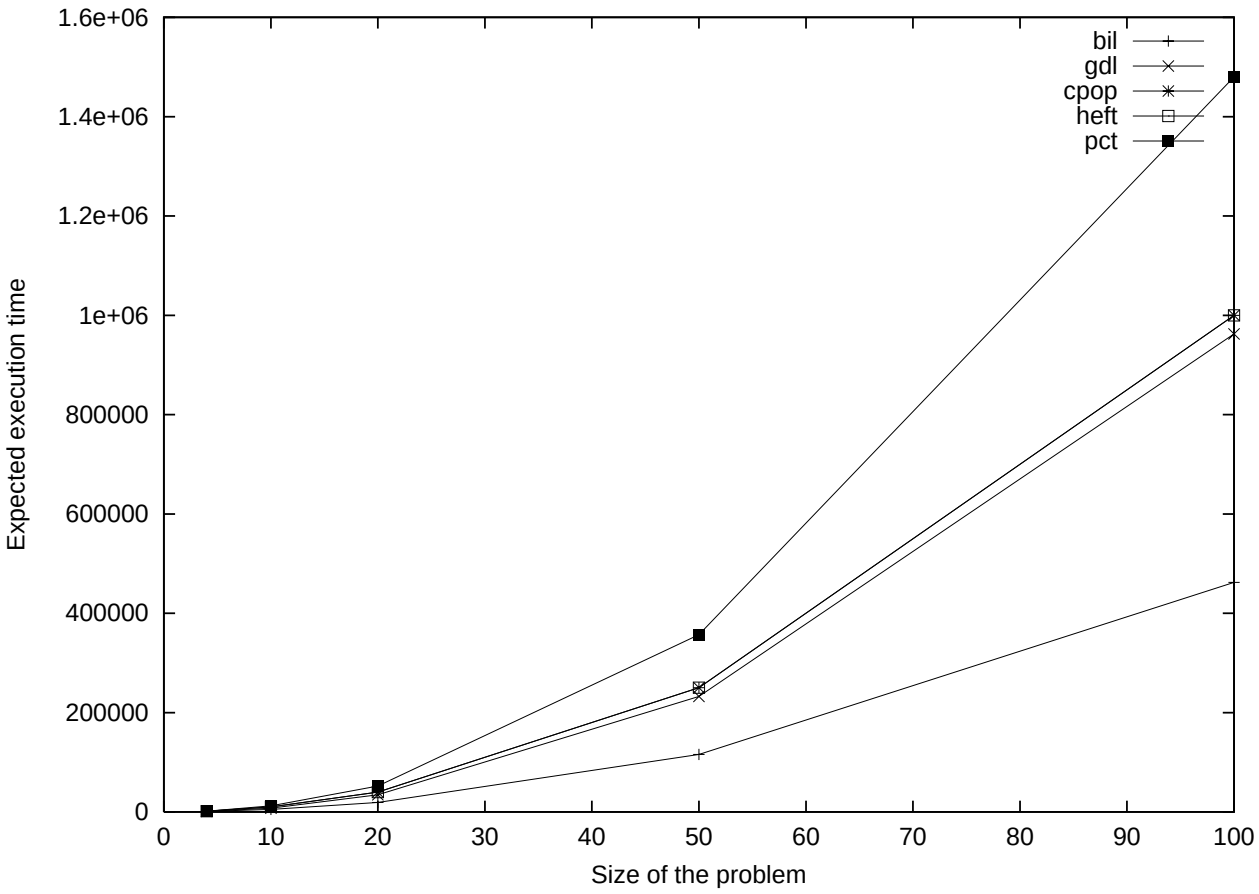


Figure 5: Comparison of the different heuristics for LAPLACE problem with speed equals to 100,150 and 200.

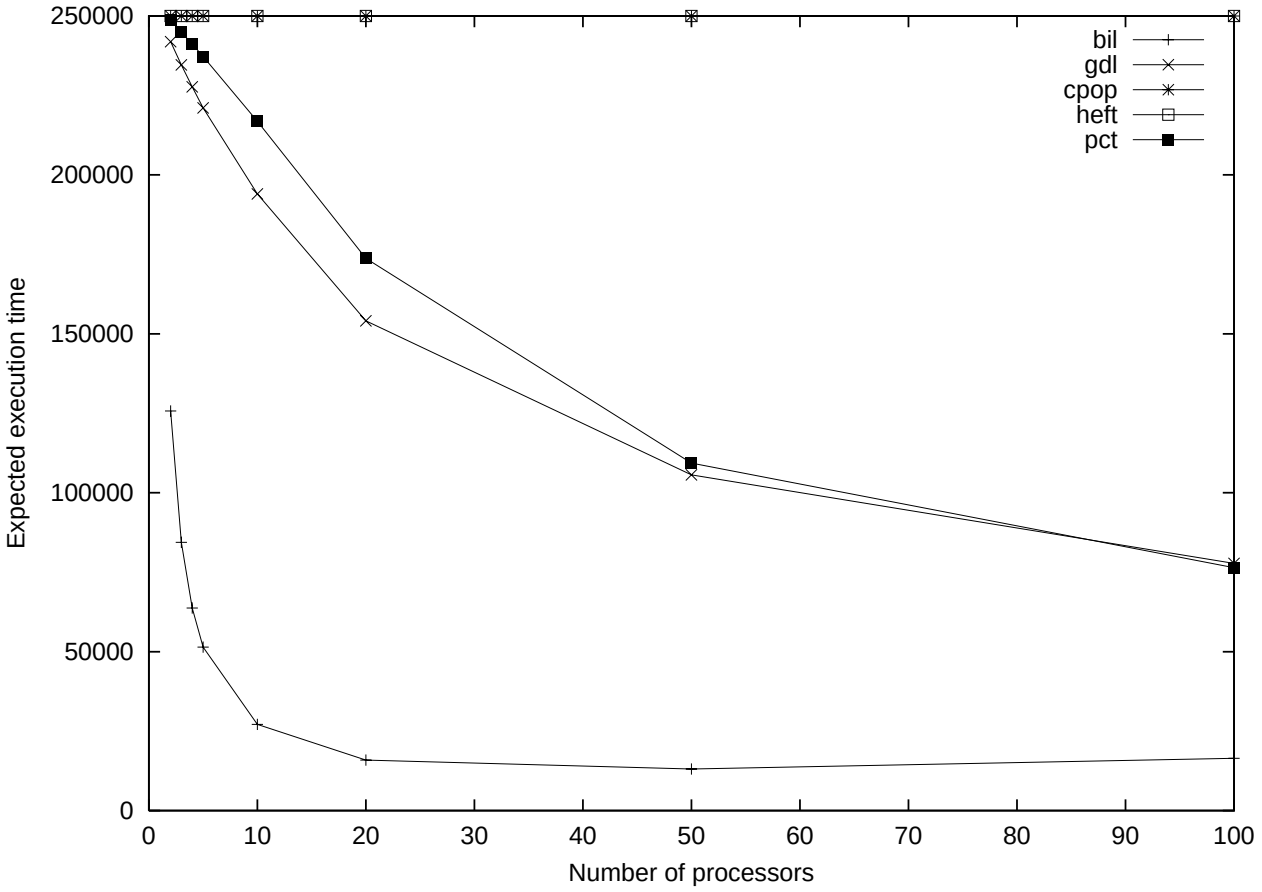


Figure 6: Impact of the number of processors for LAPLACE problem. The speed of the k^{th} processor is $100 + k$

Critical Path On a Processor (CPOP) and the *Generalized Dynamic Level* (GDL) algorithm. None of these heuristic presents good results in all testbeds. Some of them ($HEFT$ and $CPOP$) are bad for very regular problems (each node of the task graph is on a critical path $LAPLACE$, $STENCIL$, $Fork-Join$) and pretty good in the other cases. BIL is very good for $LAPLACE$ and $Fork-Join$ problems but relatively bad for $STENCIL$ and LU problems. PCT is the best for the $Fork-Join$ problem and is quite good for the LU problem. Finally, GDL never present good results, however, it is never the worst heuristic. It appears that the $STENCIL$ problem is one of the most difficult to solve (with $LAPLACE$) and none of the five heuristics is able to efficiently solve this problem.

References

[1] Ph. Chretienne. Task scheduling over distributed memory machines. In M. Cosnard, P. Quinton, M. Raynal, and Y. Robert, editors, *Parallel and Distributed Algorithms*, pages 165–176. North Holland, 1989.

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

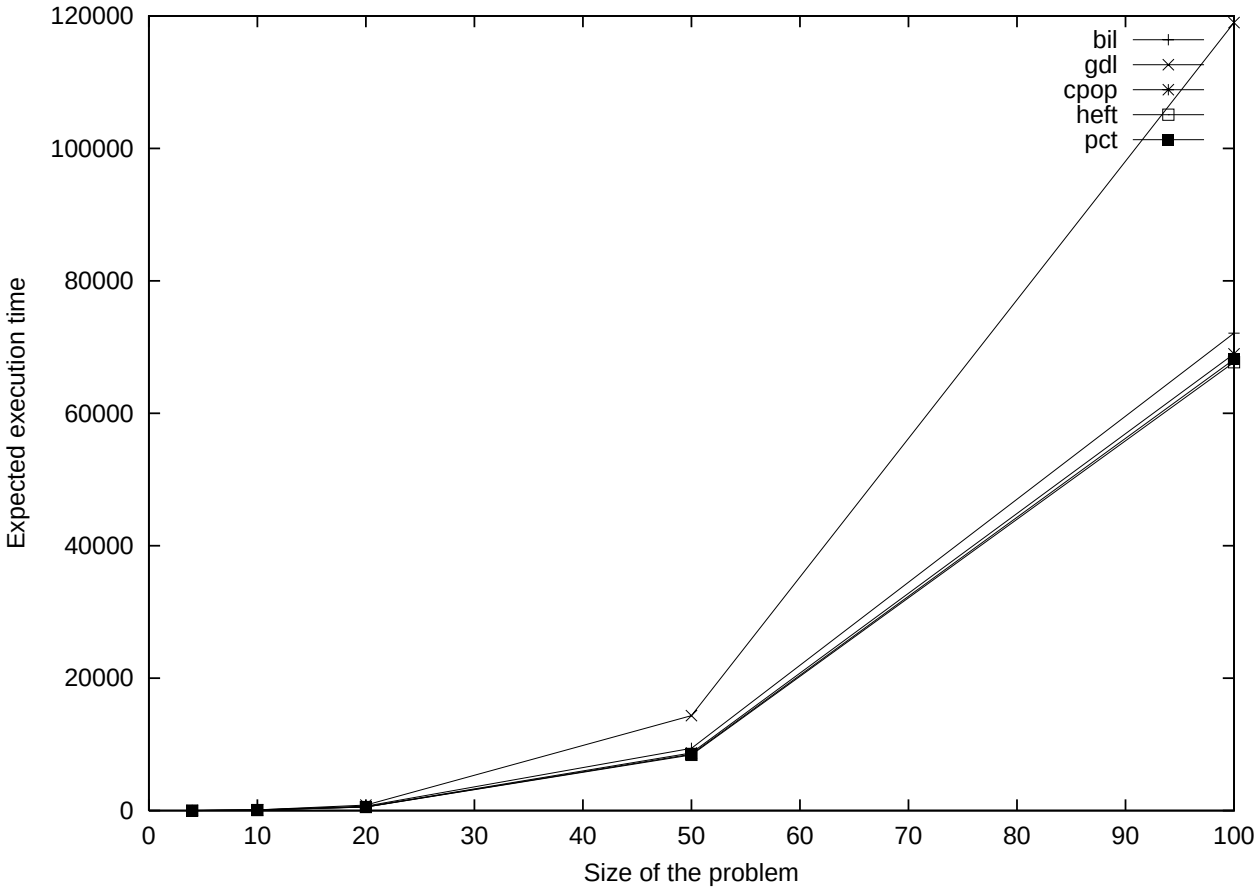


Figure 7: Comparison of the different heuristics for LU problem with speed equals to 1,2 and 3

Society Press, 1998.

[3] Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In *Proceedings of Europar'96*, volume 1123 of *LNCS*, Lyon, France, August 1996. Springer Verlag.

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

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

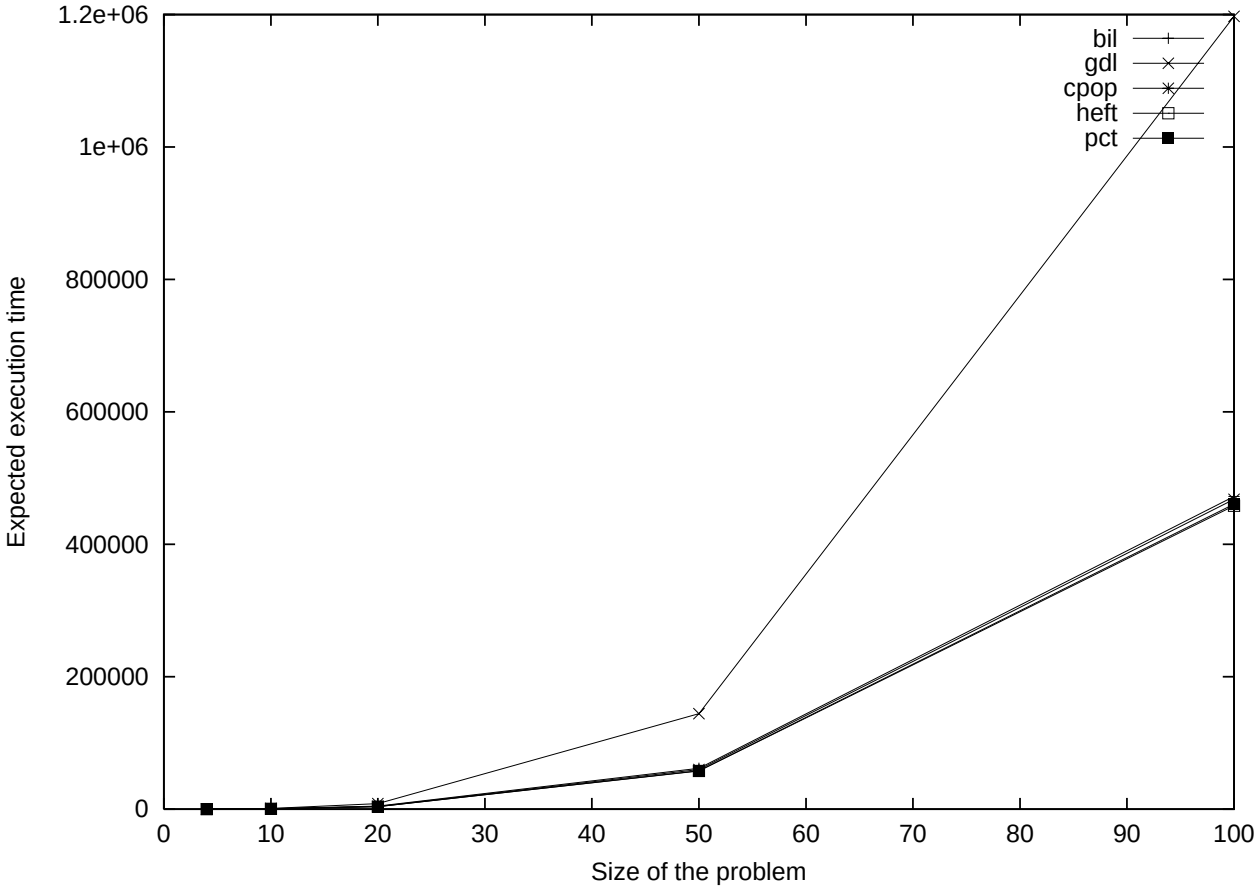


Figure 8: Comparison of the different heuristics for LU problem with speed equals to 10,11 and 12

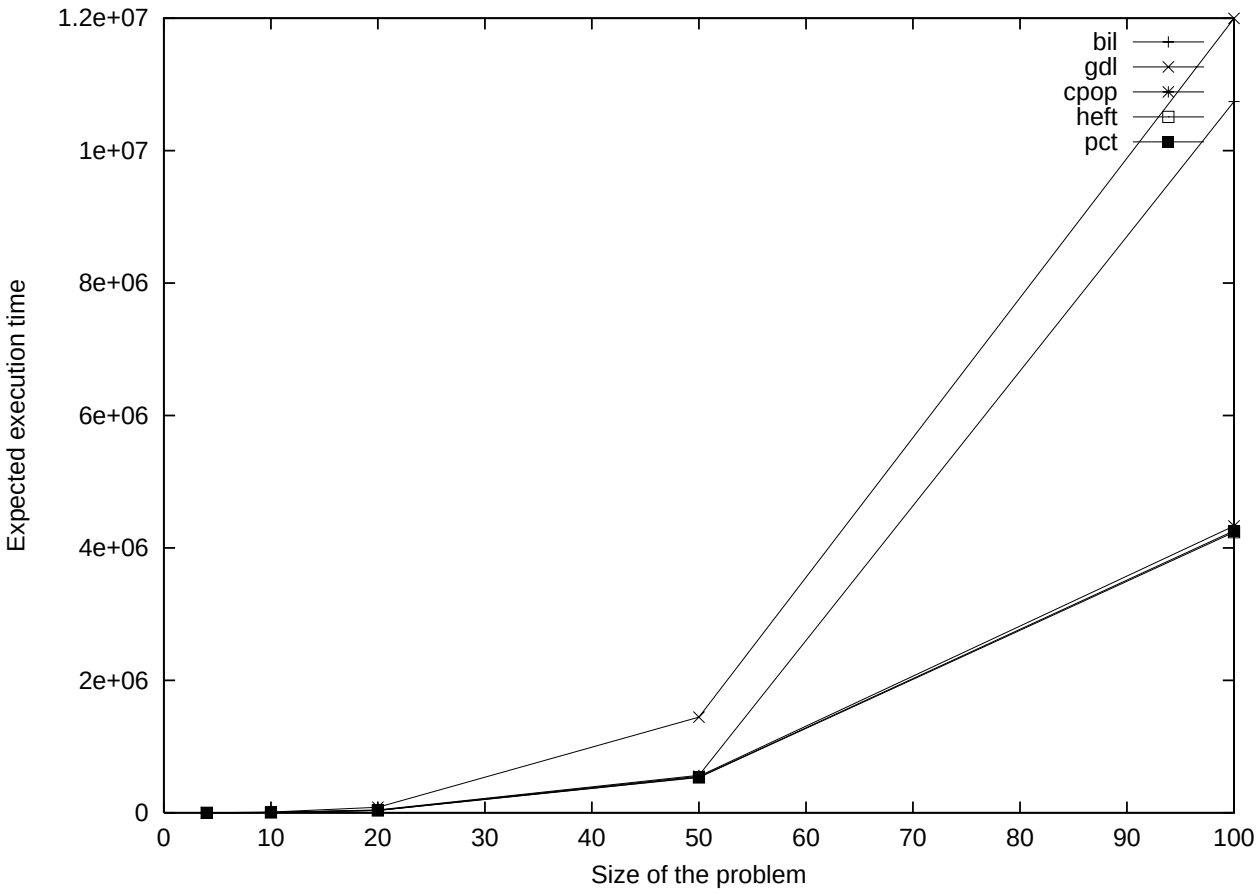


Figure 9: Comparison of the different heuristics for LU problem with speed equals to 100,101 and 102

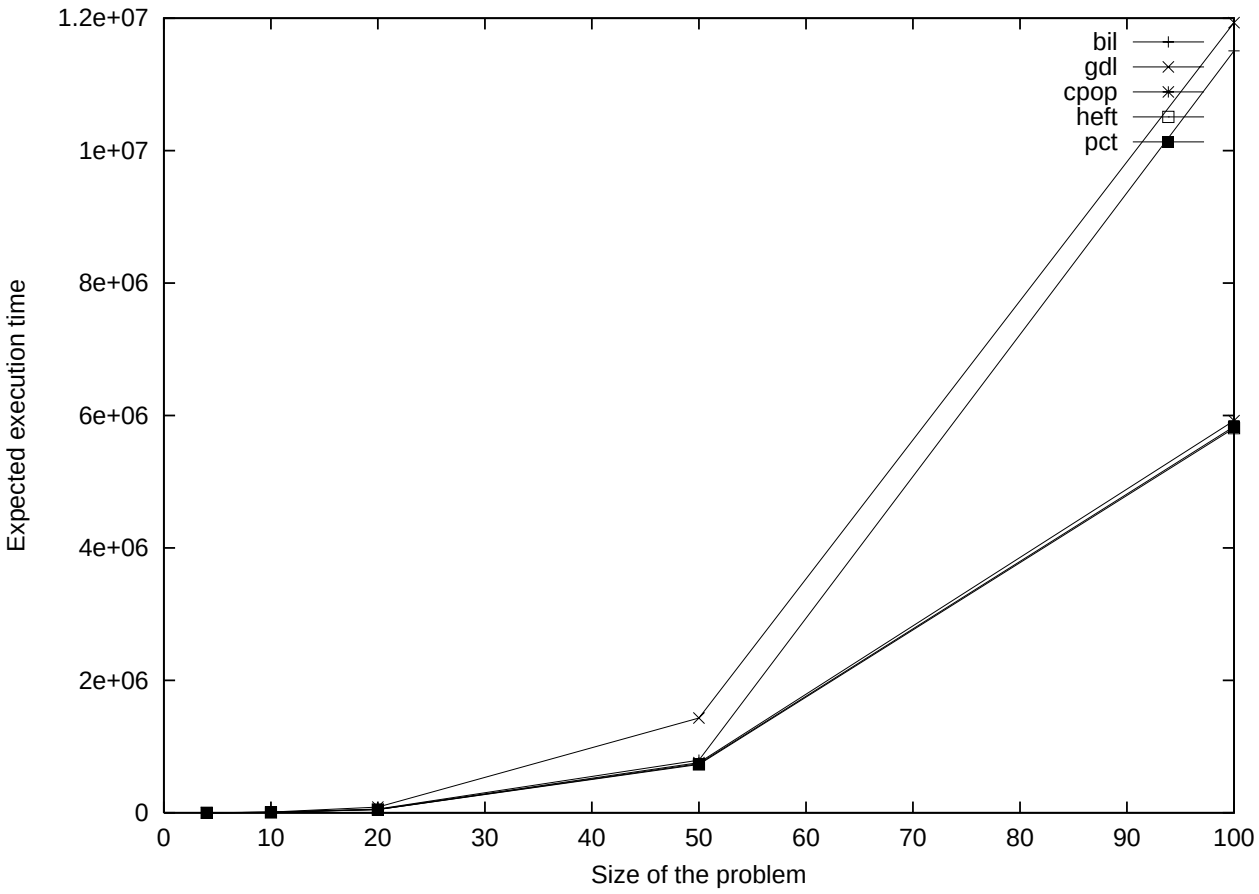


Figure 10: Comparison of the different heuristics for LU problem with speed equals to 100,150 and 200.

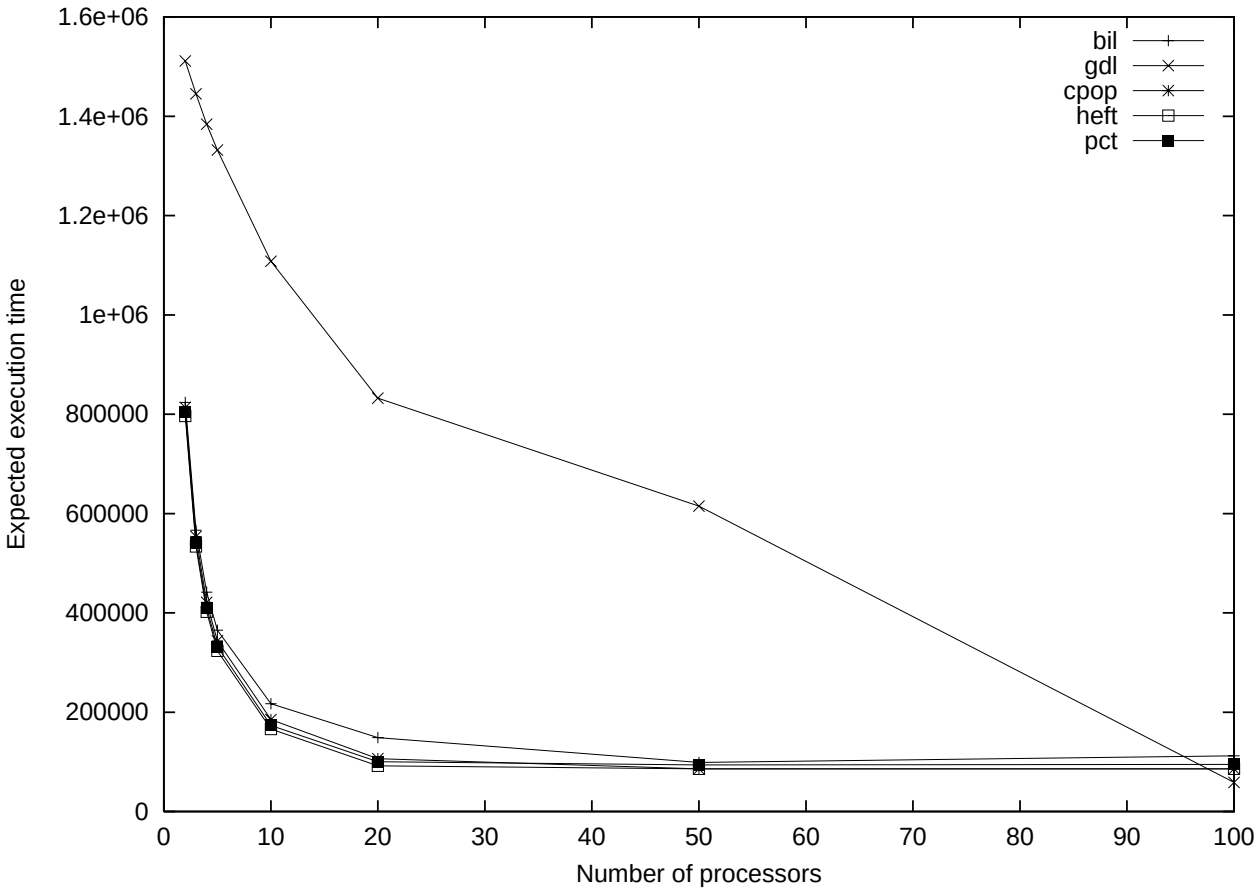


Figure 11: Impact of the number of processors for LU problem. The speed of the k^{th} processor is $100 + k$

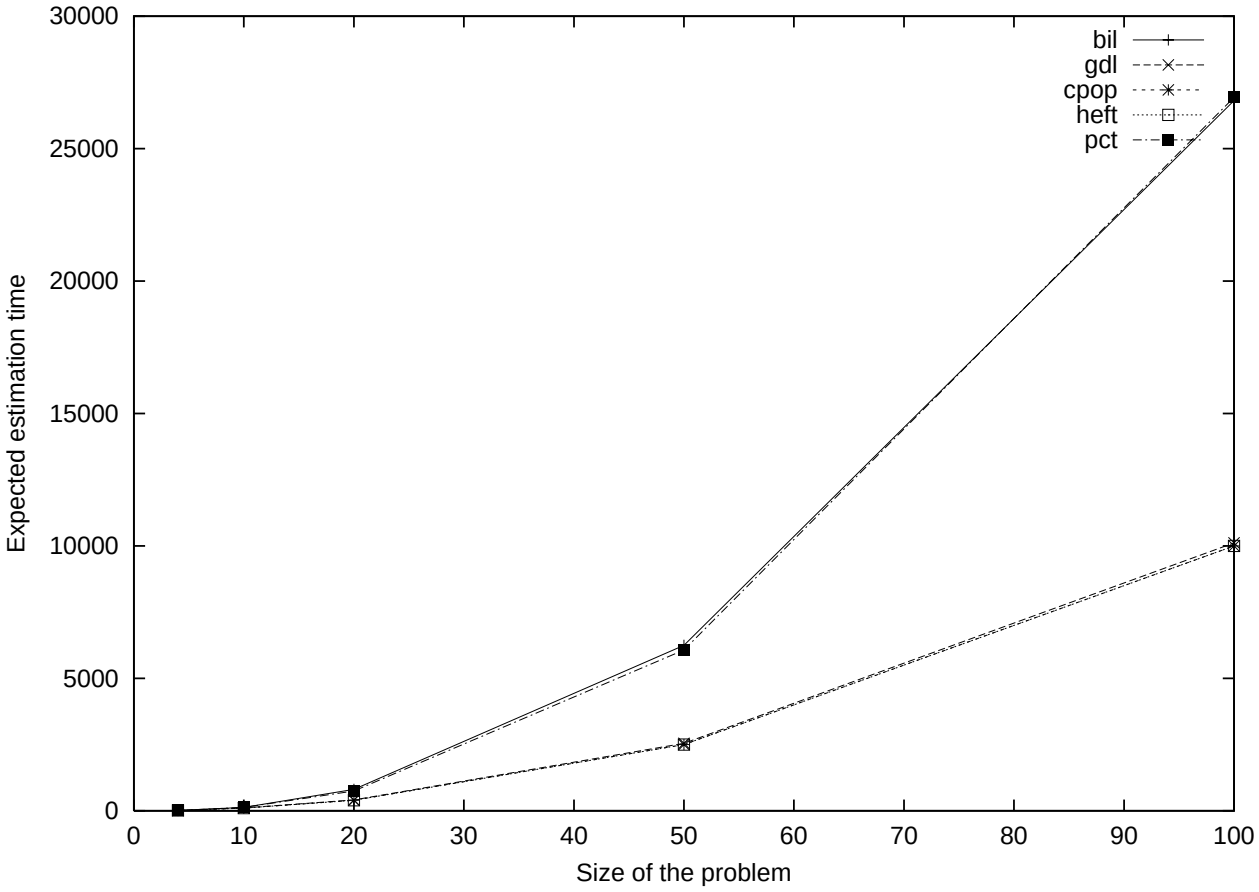


Figure 12: Comparison of the different heuristics for STENCIL problem with speed equals to 1,2 and 3

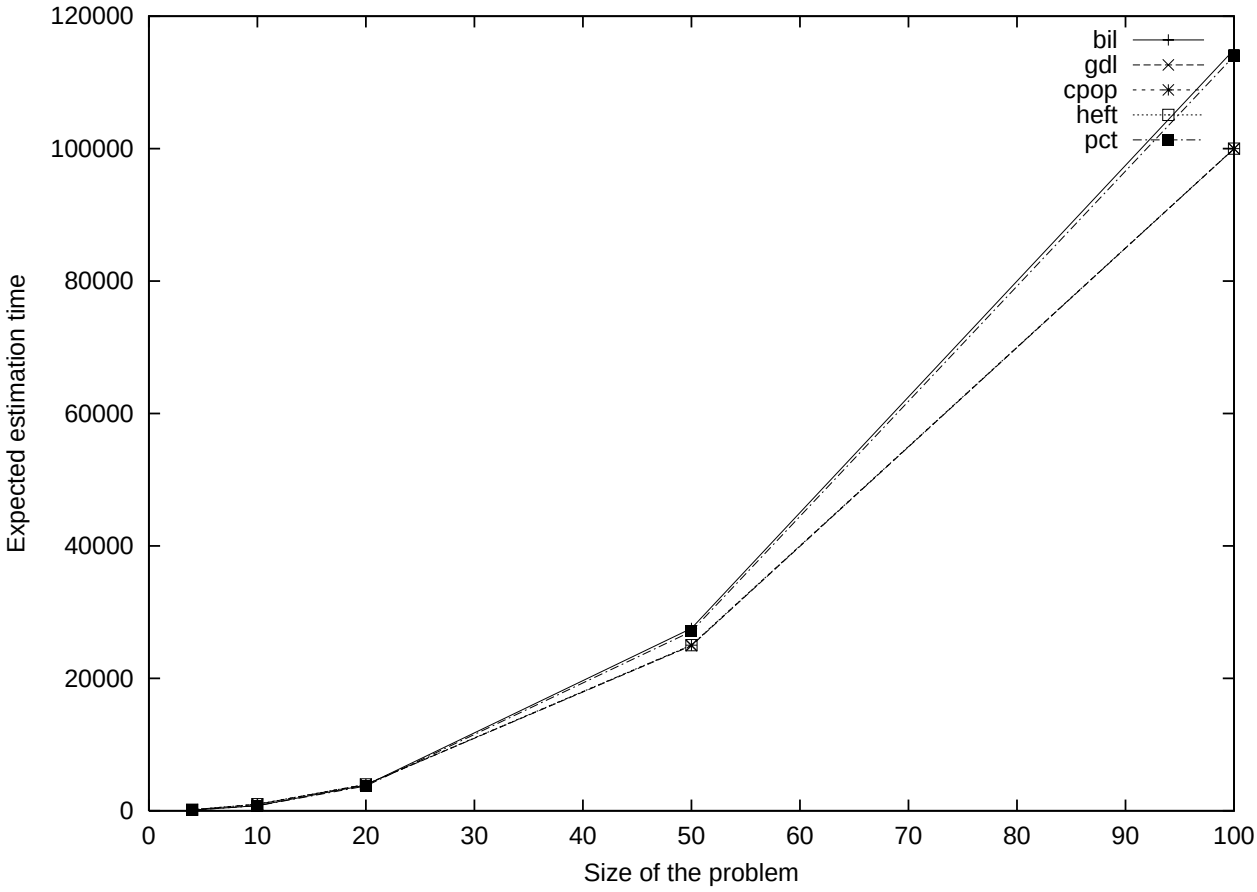


Figure 13: Comparison of the different heuristics for STENCIL problem with speed equals to 10,11 and 12

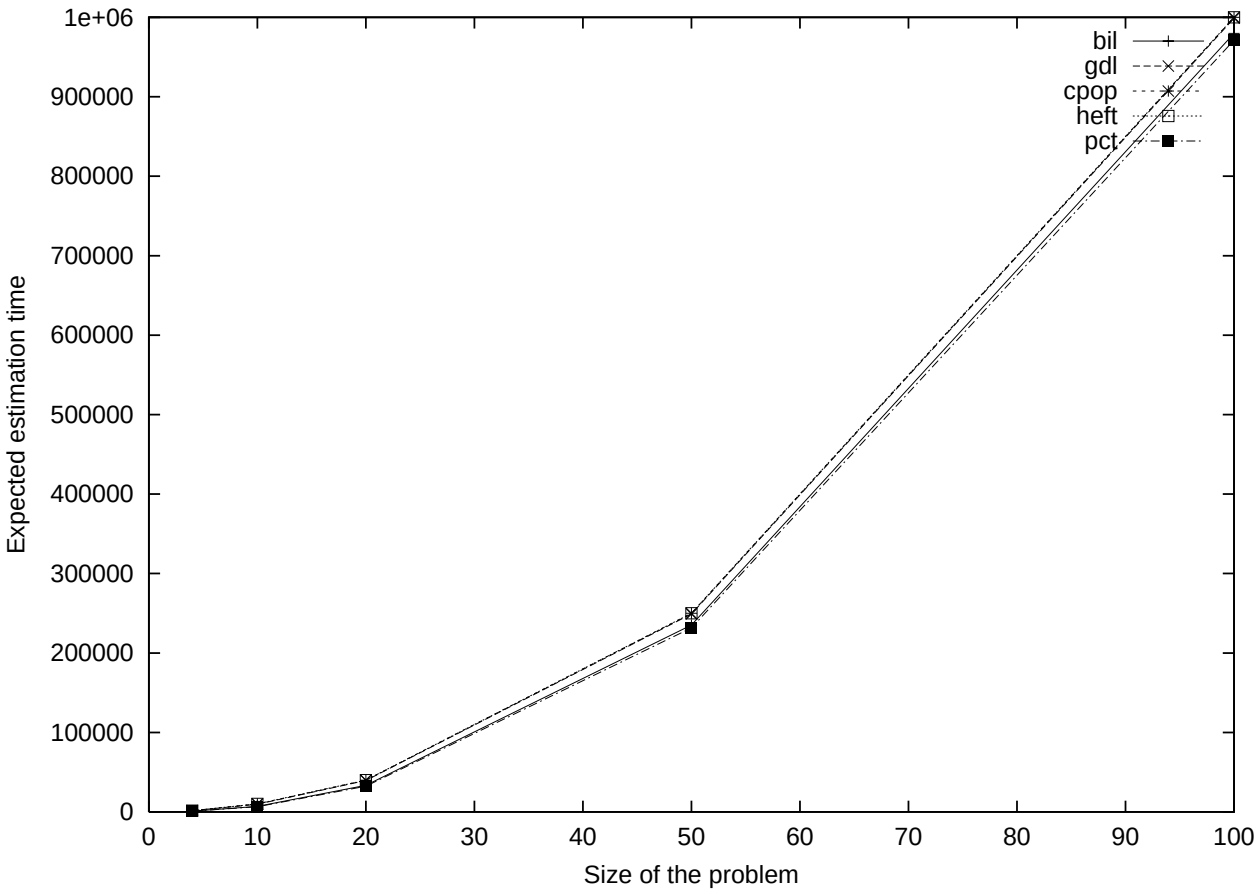


Figure 14: Comparison of the different heuristics for STENCIL problem with speed equals to 100,101 and 102

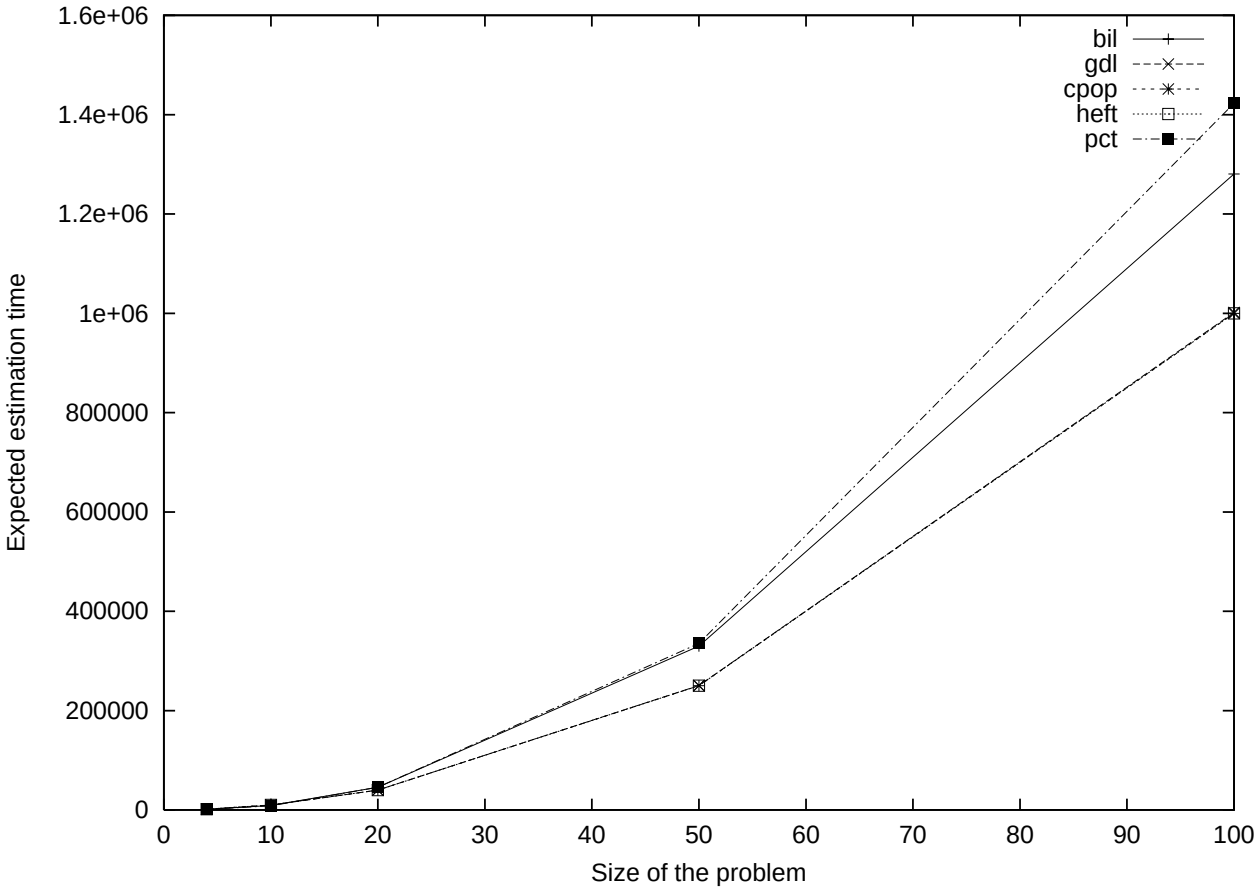


Figure 15: Comparison of the different heuristics for STENCIL problem with speed equals to 100,150 and 200.

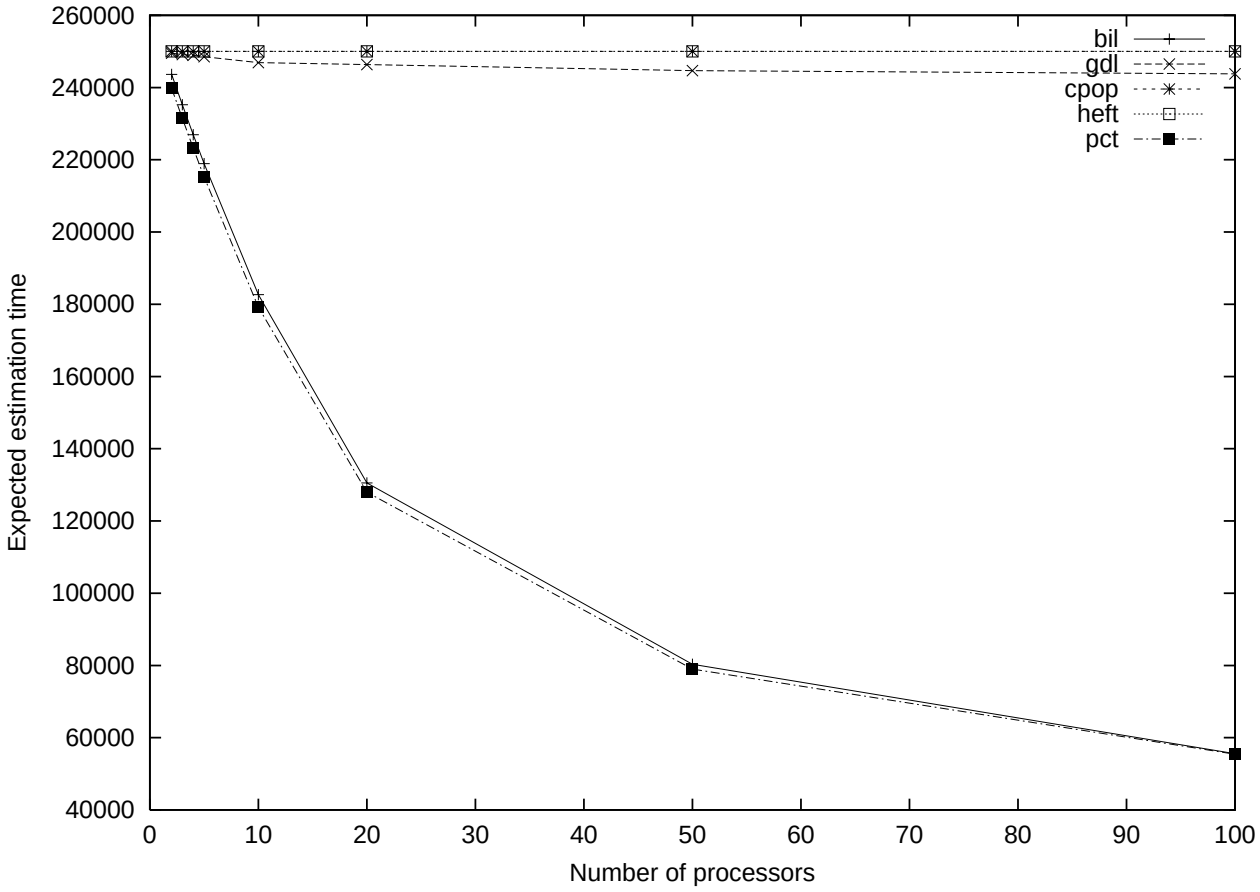


Figure 16: Impact of the number of processors for STENCIL problem. The speed of the k^{th} processor is $100 + k$

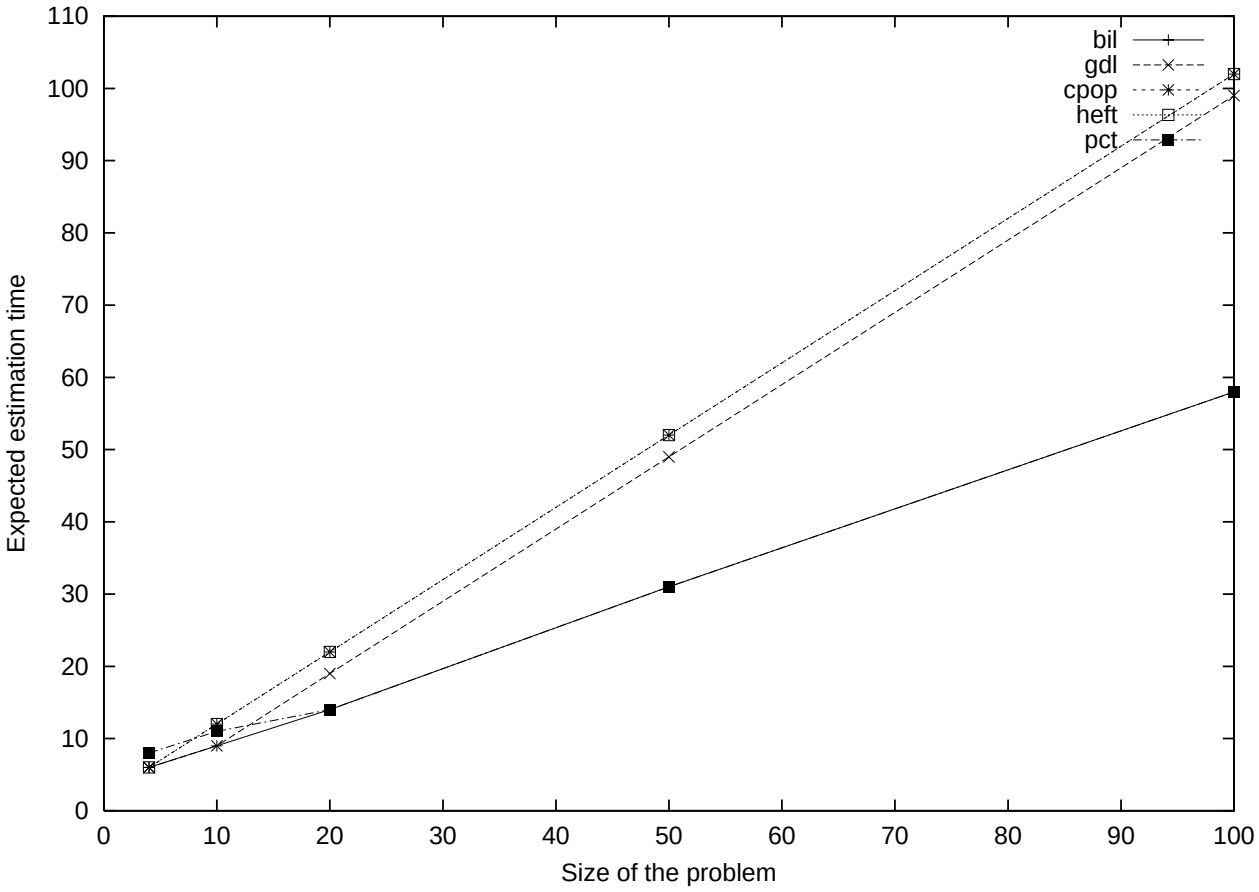


Figure 17: Comparison of the different heuristics for Fork-Join problem with speed equals to 1,2 and 3

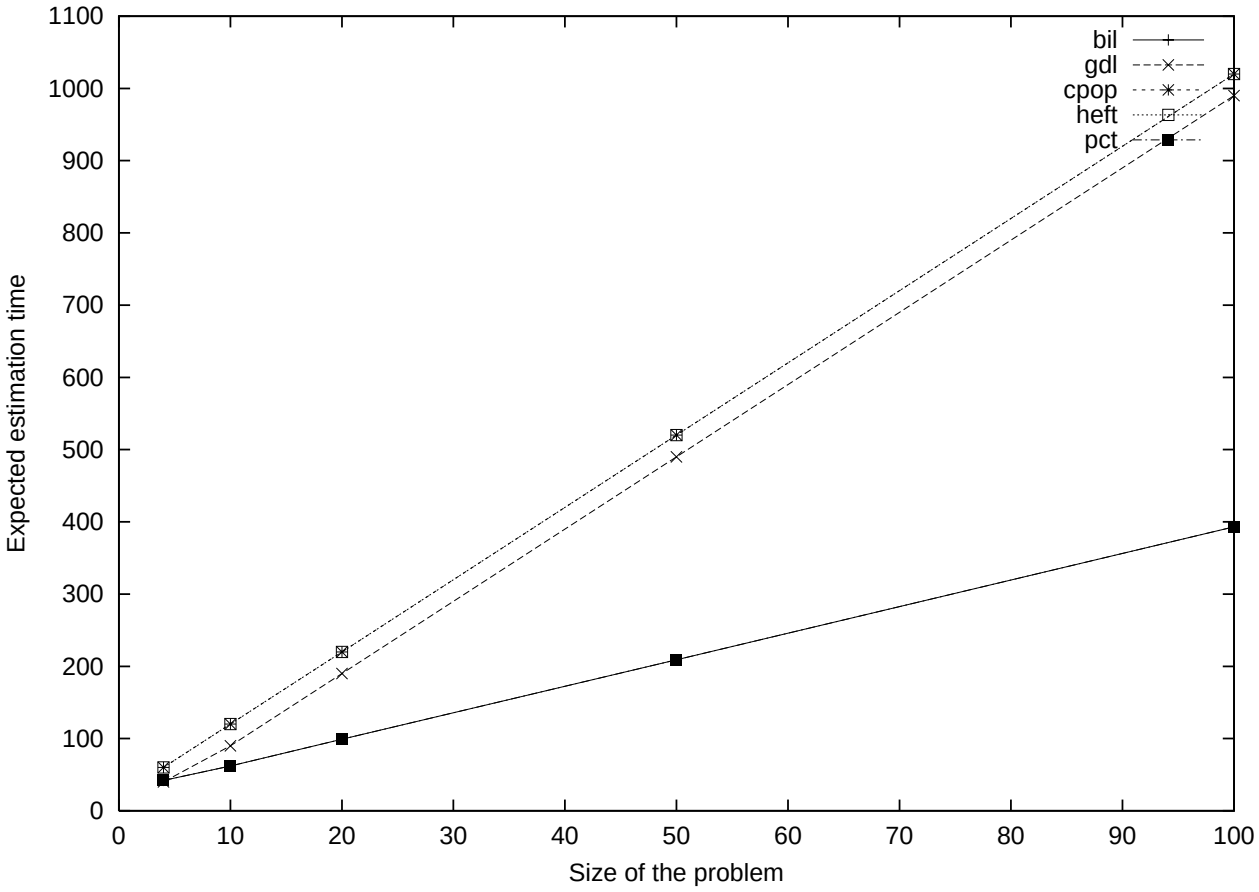


Figure 18: Comparison of the different heuristics for Fork-Join problem with speed equals to 10,11 and 12

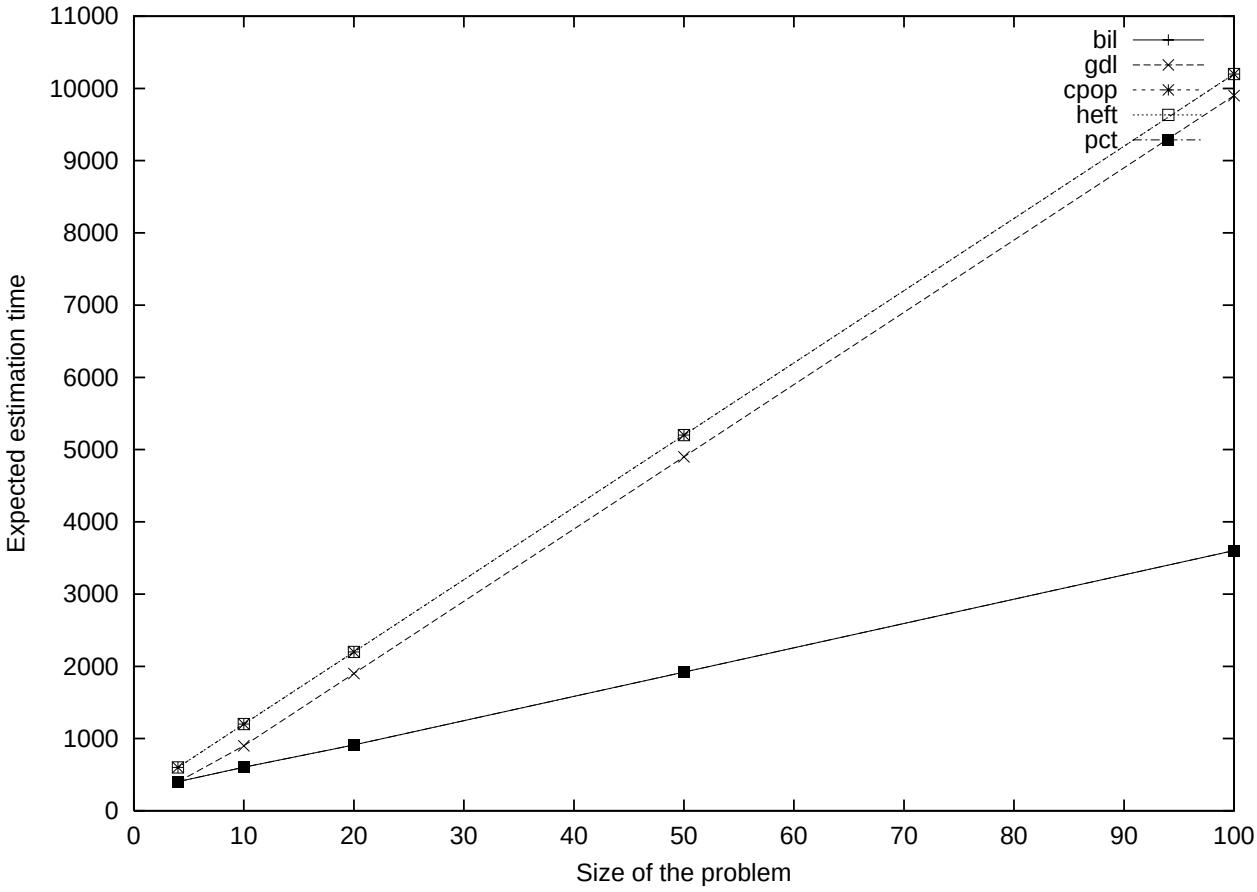


Figure 19: Comparison of the different heuristics for Fork-Join problem with speed equals to 100,101 and 102

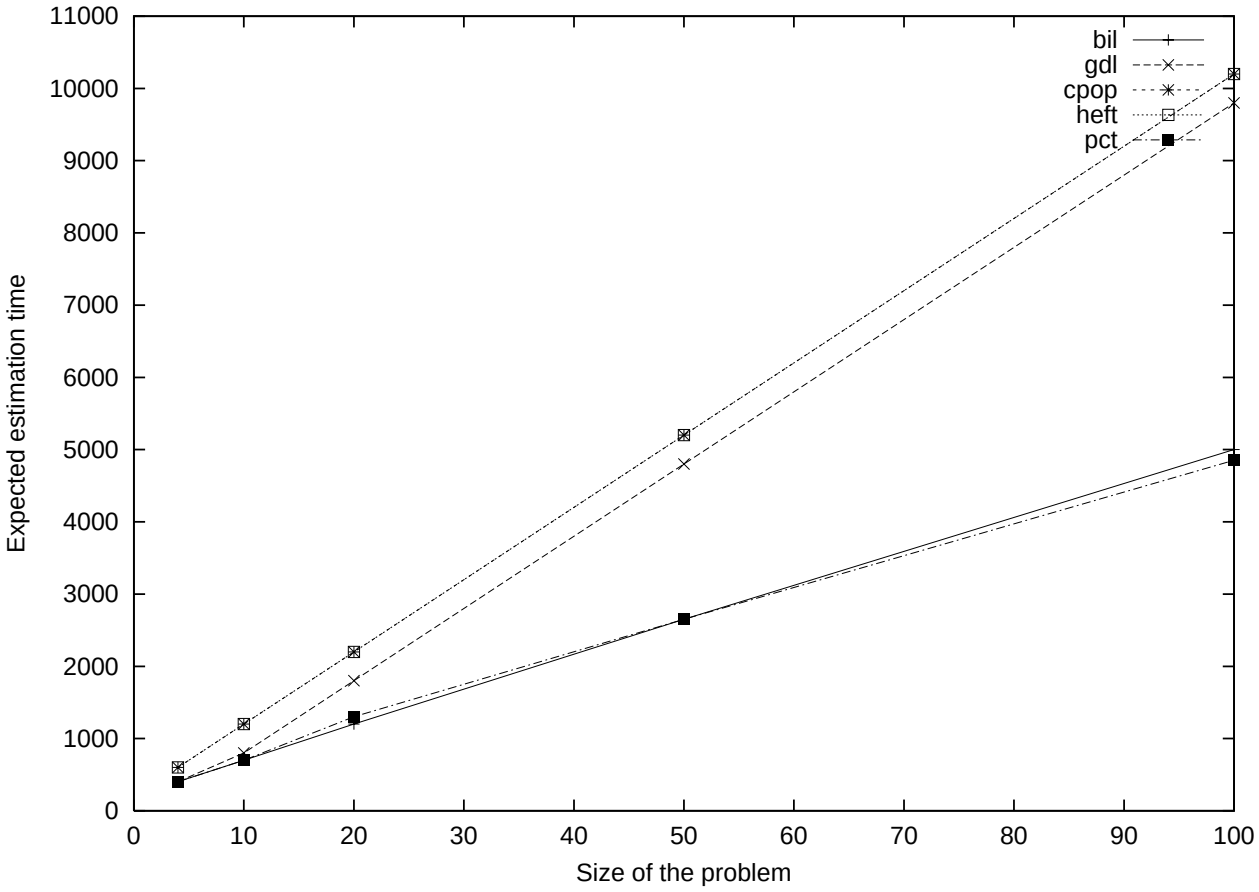


Figure 20: Comparison of the different heuristics for Fork-Join problem with speed equals to 100,150 and 200.

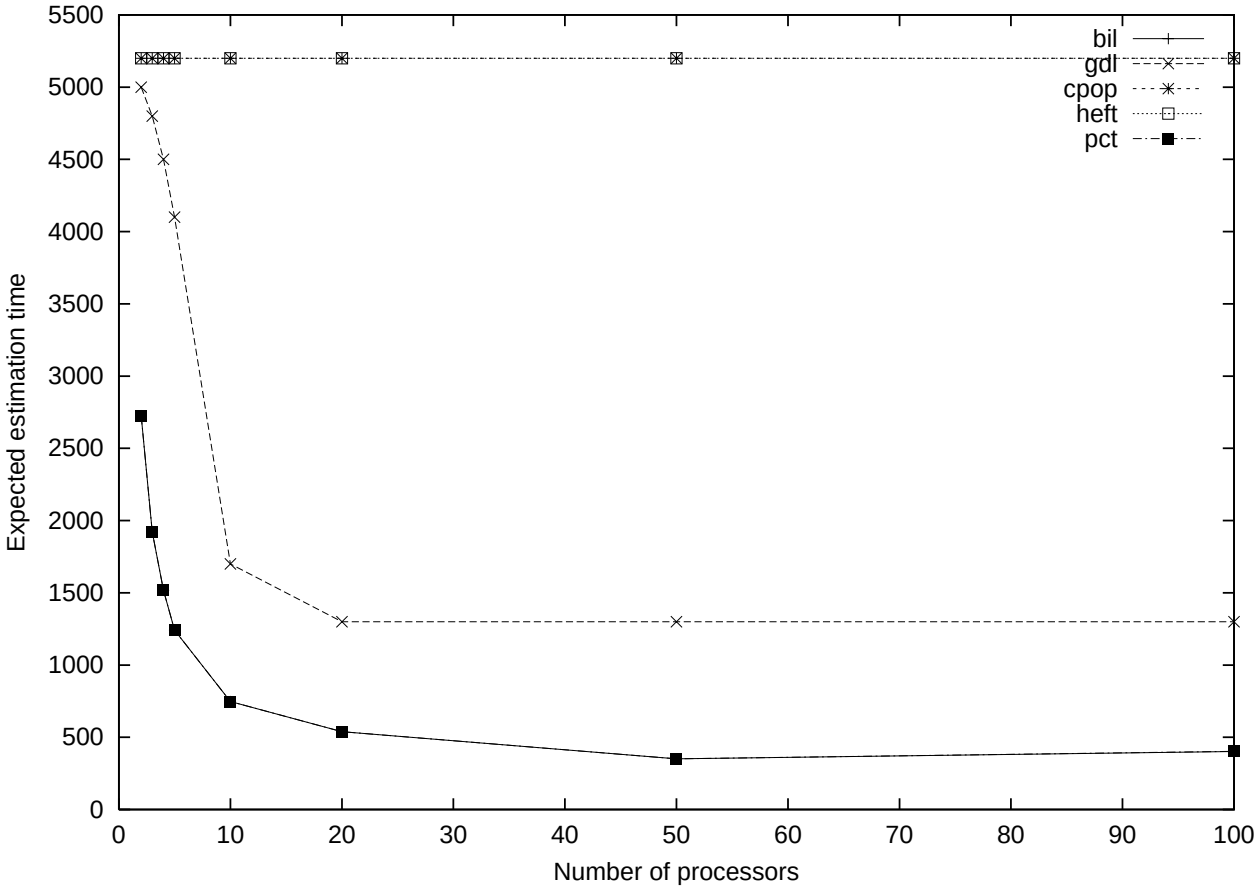


Figure 21: Impact of the number of processors for Fork-Join problem. The speed of the k^{th} processor is $100 + k$