Dynamic Program Description as a Basis for Runtime Optimization *

Jörn Gehring

Paderborn Center for Parallel Computing (PC^2), D-33095 Paderborn, Germany,
joern@uni-paderborn.de

Abstract. Dynamic load-balancing is an important topic for network-computing. In order to get the best performance out of a networked pool of dynamic resources, it is important to use as much information as possible for load-balancing decisions. Parallel programs are usually no "ex-and-hop" applications and a load-balancer that treats every execution as if it had never seen this application before, cannot achieve optimal results. We present an algorithm for generating formal descriptions of the dynamic execution behavior of distributed programs. These descriptions can then be used for a variety of tools, such as schedulers, load balancers, or routing systems.

1 Introduction

Currently there exists a variety of cluster management systems that support the distribution of sequential jobs on hundreds or thousands of networked computers. However, there is only rudimentary support for the distribution of parallel applications [6, 1]. Since communicating distributed programs need additional effort in scheduling, mapping, dynamic load-balancing, and message routing, much research has been done in these fields. However, most strategies either ignore the communication structure of the applications completely [3] or use a static representation [7]. The latter approach achieves better response times but it is still a simplification of the real problem. Fig. 1 depicts an example of three communicating processes. Looking at the dynamic program description, it is immediately clear that all three processes should be mapped onto one single machine. However, this cannot be derived from the static description. Therefore, newer policies

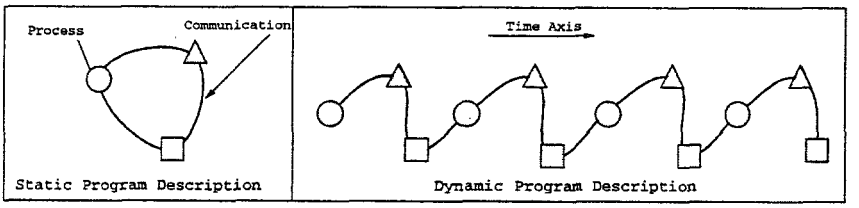

Fig. 1. Structure information of static vs. dynamic program description

* This work was supported by the Ministry for Science and Research of Northrhine-Westfalia, project *Metacomputing*

try to consider the dynamics of parallel applications as well. [9] and [2] are some examples of these. Both require dependency graphs which describe the dynamic behavior of the program during a certain time period. However, the user needs to provide these graphs for those sections of the algorithm that are worthwhile the effort of specific optimizations.

With the concept of heterogeneous metacomputing the impact of improper process assignments has increased essentially [5]. Consequently, the availability of formal descriptions of dynamic program characteristics is important for the performance of large WAN-distributed applications and thus for the acceptance of metacomputing at all.

In the following we present a concept for describing characteristics of the dynamic execution behavior of distributed communicating programs automatically. Since the general problem of extracting all features is known to be NP-complete, we reduce the complexity by identifying the characteristics usable for optimization purposes in Section 2. From this we derive an algorithm in Section 3 which provably finds all usable traits of a distributed message-passing program in polynomial time. This concept has been implemented in the context of the MARS [5] project which is part of the international *"Metacomputer Online"*[8] initiative.

2 Dependency Graphs and Phases

A trace of a parallel program can be represented by a directed graph with the nodes representing sequential computations and the edges standing for communications. In the following, we show how this representation can be used for deriving information about the internal structure of the parallel application.

2.1 Dependency Graphs

We consider a parallel program as a number of sequential computations interleaved by "critical" communications. These are blocking receive operations, blocking sends, waiting for termination of non-blocking sends/receives, or blocking collective communications (broadcast, gather, ...). For convenience, we also consider start and termination of a process as critical communications. The block of sequential statements between any sequence of two communications is called an *independent block.*

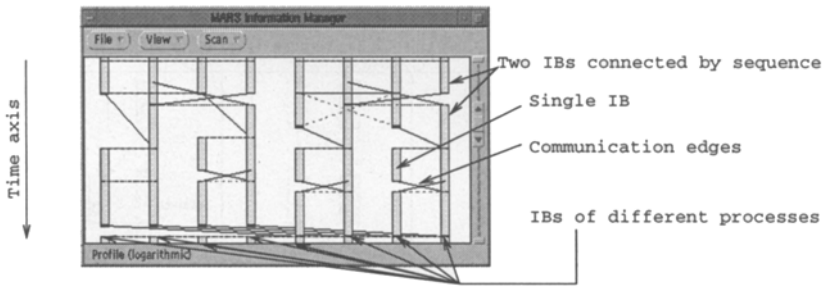

Fig. 2. Example for a dependency graph (sequence edges omitted for the sake of clarity)

Definition 1 *Let I be a sequence of instructions to be executed within a single process. I is called an* "independent block" *(or IB for short), iff the statements to be executed immediately before and after I are critical communications and there are no critical communications in I.*

Thus, all IBs are disjunct, no IB can start before all of its predecessors have finished, and once the execution of an IB has started, it can continue until completion. (See Fig. 2 for an example of a parallel sorting algorithm.)

Definition 2 *A directed graph is called a* "dependency graph", *if its nodes are independent blocks and for any two nodes α and β there is an edge from α to β iff at least one of the following conditions holds:*

1. *Both belong to the same process and β follows immediately after α (α "precedes" β). These edges are called "sequence edges". (Fig. 3.1)*
2. *The instruction after α is a blocking send which is received by the instruction directly before β. The latter is either a blocking receive or a wait for the termination of a non-blocking receive. (Fig. 3.2)*
3. *The instruction after α and that before β are both part of the same collective communication. (Fig. 3.3)*
4. *β waits for the termination of a blocking or non-blocking send initiated by a preceding IB and received immediately after α. (Fig. 3.4)*

In [5] we have shown, how dependency graphs can be collected from program runs without significant impact on the execution speed. Note that applications produce different dependency graphs in different execution runs. These have to be merged together for consolidating the statistical data of the application [4].

2.2 Phases

Our strategy is to perform offline analysis on dependency graphs in order to detect repeating patterns and store them into a knowledge base. In [5] we have shown, how this knowledge can be used for speeding up distributed programs by dynamic migration decisions. During the offline analysis we are looking for "phases" in the dependency graphs. A phase is a subgraph of a dependency graph that represents a set of instructions performing a closed subtask. It is considered to be of high quality, if it occurs frequently and covers a large amount of the program's overall resource usage. Concerning sequential or data parallel

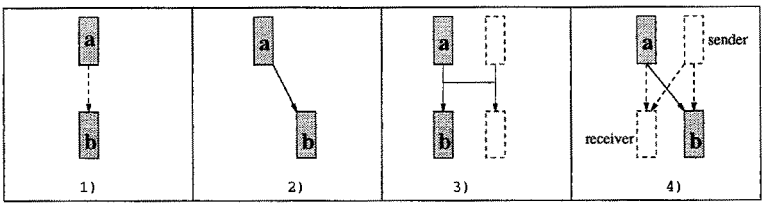

Fig. 3. Different edges of a dependency graph

programs, high quality phases are closely related to loop- and function-bodies and can therefore be detected during compile time. This is not possible for distributed MIMD programs, because the dynamic interaction between different modules can usually not be predicted by the compiler.

Since we are going to use phases for the prediction of runtime behavior, a phase which has started execution must not be interrupted by communications with IBs that are out of scope of this phase. I.e., the predecessors of an IB are either all inside the phase or all outside. This leads us to the following formal definition:

Definition 3 *Let $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ be a dependency graph with $\mathcal{V}$ being the set of nodes and $\mathcal{E}$ being the set of edges. A connected subgraph $\mathcal{P} = (\mathcal{V}_\mathcal{P}, \mathcal{E}_\mathcal{P})$ of $\mathcal{D}$ is a* "phase of $\mathcal{D}$"*, iff*

1. *For all $v, u \in \mathcal{V}_\mathcal{P}$: If $(v,u) \in \mathcal{E}$, then $(v,u) \in \mathcal{E}_\mathcal{P}$*
2. *For all $(u,v) \in \mathcal{E}_\mathcal{P}$, $u' \in \mathcal{V}$: If $(u',v) \in \mathcal{E}$, then $u' \in \mathcal{V}_\mathcal{P}$ and $(u',v) \in \mathcal{E}_\mathcal{P}$.*

Consequently, a single IB is also a phase, which we call an "atomic phase".

3 Detecting Phases

In the following we introduce an order on the quality of phases and describe a deterministic algorithm that extracts all valuable phases from a dependency graph. Although there may be a total of $2^{|\mathcal{V}|}$ phases in a dependency graph, we will prove that at most $O(|\mathcal{V}|^2)$ of them are usable for optimization purposes.

3.1 The Quality Function

As mentioned in Section 2.2, a phase is "good", if it describes a large part of the internal structure of the program. I.e., it occurs frequently and covers a high percentage of the overall resource usage. The algorithm presented in Section 3.3 constructs large phases out of smaller ones. Thus, the quality function has to make small frequently occurring phases attractive during the first iterations of the algorithm, while larger phases ought to become better to the end (see Sec. 3.4). This leads us to the following definition:

Definition 4 *Let $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ be a dependency graph and $A \subseteq \mathcal{D}$ a phase. We define the* "quality of A" *$Q(A)$ to be $Q(A) := (\, Occ(A)\,,\ |A|\,)$ with*

$$Occ(A) := \begin{cases} 0 & , \text{ if } A = \emptyset \\ |\{A' \subseteq \mathcal{D} \mid A, A' \text{ are isomorph}\}| & , \text{ else} \end{cases}$$

Two quality values are compared using lexicographical comparison.

Phase A is consequently considered to be of higher quality than B, if it appears more frequently or if A and B have the same number of instances but A is larger than B. In Section 3.4 we will show, how this definition directs the phase-detection algorithm on its search for valuable phases.

1) Let $i := 0$; $S_0 := \mathcal{V}$; $S_0^* := \mathcal{E}$
2) While $S_i^* \neq \emptyset$ Do
3) Find $(A_i, B_i) \in S_i^*$ with $Q(A_i \oplus B_i) = \max_{(\alpha,\beta)\in S_i^*}\{Q(\alpha \oplus \beta)\}$
4) Let $S_{i+1} := S_i \cup \{A_i \oplus B_i\}$
5) Let $T_i := \{\ (C, A_i \oplus B_i) \mid C \in S_i \backslash (A_i \oplus B_i) \quad \wedge$
$[\exists \alpha \in C, \beta \in (A_i \oplus B_i) : (\alpha,\beta) \in \mathcal{E}]\ \} \quad \cup$
$\{\ (A_i \oplus B_i, C) \mid C \in S_i \backslash (A_i \oplus B_i) \quad \wedge$
$[\exists \alpha \in (A_i \oplus B_i), \beta \in C : (\alpha,\beta) \in \mathcal{E}]\ \}$
6) Let $U_i := \{(F, G) \in S_i^* \mid F \subseteq A_i \oplus B_i \wedge G \subseteq A_i \oplus B_i\}$
7) Let $S_{i+1}^* := (S_i^* \cup T_i) \backslash U_i$
8) Let $i := i + 1$
9) Let $I := i$

Fig. 4. The phase-detection algorithm

3.2 Creating Complex Phases

Complex phases are created by the concatenation of at least two simpler phases. Due to Def. 3 the union of two phases does not have to be a phase by itself. Other IBs that precede one phase but are not part the other one may have to be added. These additional IBs are covered by the following relation:

Definition 5 *Let A and B be phases of a dependency graph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ and let $\gamma \in \mathcal{V}$ be a node of $\mathcal{D}$. We say $\gamma \rightarrow (A \cup B)$ ("γ feeds $(A \cup B)$"), if γ has to be added to $A \cup B$ in order to make it a phase:*

$$\gamma \rightarrow (A \cup B) \Leftrightarrow \exists\, \alpha, \beta \in \mathcal{V} \text{ with } \alpha \in A \wedge \beta \in B \wedge (\alpha,\beta) \in \mathcal{E} \wedge (\gamma,\beta) \in \mathcal{E}$$

We can now define the concatenation of two phases A and B as:

$$A \oplus B := \begin{cases} \emptyset & ,if\ \nexists\, \alpha, \beta \in \mathcal{V} \text{ with } \alpha \in A \wedge \beta \in B \wedge (\alpha,\beta) \in \mathcal{E} \\ A \cup B \cup \{\gamma \in \mathcal{V} \mid \gamma \rightarrow (A \cup B)\} & ,otherwise \end{cases}$$

Thus, $A \oplus B$ is the smallest possible phase that includes $A \cup B$.

3.3 The Phase-Detection Algorithm

The phase-detection algorithm starts with a set S containing all atomic phases. It then successively merges the best pair out of all known phases and inserts it into S. Thus, after the algorithm has terminated, S will hold all extracted phases. In order not to do superfluous work, we exclude pairs lying entirely within a phase that has already been created. Fig. 4 gives a formal description of the algorithm which takes a dependency graph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ as input.

S_i contains all phases known after the i-th iteration and S_i^* holds all pairs of phases in S_i that may have to be considered during later iterations. Lines 5) and 6) may look somewhat complicated, but their purpose is simple. T_i includes all pairs to be added to S_i^* due to the creation of $(A_i \oplus B_i)$. U_i, on the other hand, incorporates all pairs having become superfluous.

3.4 A Closer Look at the Algorithm

Looking at Def. 4, it is obvious that $Occ(A)$ cannot increase, when phase A is extended. Furthermore, all pairs added to S^* in line 5) and 7) will be extensions of phases already contained in S_i^*. Thus, we can conclude that the greater the value of i is, the less often the newly constructed phases occur in $\mathcal{D}$. Since $Occ(A) \in \{0, \ldots, |\mathcal{V}|\}$ for any phase A, we have now shown that

$$\exists\, c_{|\mathcal{V}|}, \ldots, c_0 \in \{0, \ldots, I-1\} \text{ with } 0 = c_{|\mathcal{V}|} \leq c_{|\mathcal{V}|-1} \leq \ldots \leq c_0 = I \text{ and}$$
$$\forall_{j \in \{1,\ldots,|\mathcal{V}|\}} \; : \; \left[\forall_{k \in [c_j; c_{j-1}[} \; : \; Occ\left(A_k \oplus B_k\right) = j\right] \quad (1)$$

I.e., from iteration c_j inclusive to iteration c_{j-1} exclusive the algorithm creates only phases occurring exactly j times within the dependency graph.

The phases constructed in line 4) are connected subgraphs of $\mathcal{D}$. Since $\mathcal{D}$ is finite, there exist at most $n_j \leq |\mathcal{V}|$ disjunct phases $P_1, P_2, \ldots, P_{n_j}$ which occur precisely j times in $\mathcal{D}$ such that

$$Occ\left(P_1\right) = \ldots = Occ\left(P_{n_j}\right) = j \text{ and } \left[\forall_{k,l \in \{1,\ldots,n_j\},\, k \neq j} \; : \; Occ\left(P_k \oplus P_l\right) < j\right] \quad (2)$$

During the c_j-th iteration, the algorithm constructs the so far largest phase appearing j times in $\mathcal{D}$. Because the quality function is defined to judge on the size of two phases, if their occurrence is identical, the algorithm will try to extend $A_{c_j} \oplus B_{c_j}$ during the next iteration. If this is not possible, because there is no phase C with $[(C, A_{c_j} \oplus B_{c_j}) \in S^*_{c_j+1} \; \vee \; (A_{c_j} \oplus B_{c_j}, C) \in S^*_{c_j+1}] \; \wedge \; Occ((A_{c_j} \oplus B_{c_j}) \oplus C) = j$, it tries extending another phase with the same number of instances that is not connected to $A_{c_j} \oplus B_{c_j}$.

Lines 6) and 7) remove all pairs of inner phases from S^* and therefore ensure that once a phase with occurrence j has been extended to its maximum, it will no longer be considered throughout iterations $[c_j; c_{j-1}[$. Thus, during the next iteration, the algorithm will continue extending the last phase or it starts inflating a completely new one. From Def. 4 we conclude that each IB $v \in \mathcal{V}$ belongs to at most one $P_i \in \{P_1, \ldots, P_{n_j}\}$ and during each iteration of $[c_j; c_{j-1}[$ at least one of these IBs will be assigned to its corresponding P_i. Lines 6) and 7) ensure that this IB will no longer be considered until at least iteration c_{j-1}. Since there are at most $|\mathcal{V}|$ IBs to assign, we have now shown that

$$\forall_{j \in \{1,\ldots,|\mathcal{V}|\}} \; : \; 0 \leq c_{j-1} - c_j \leq |\mathcal{V}| \quad (3)$$

Combining (3) and (1) we conclude that $I \leq |\mathcal{V}|^2$ and therefore the algorithm extracts $|S_I| \leq |\mathcal{V}| + I = O(|\mathcal{V}|^2)$ phases. Fig. 5 depicts an example run of the phase-detection algorithm. All IBs are the same and thus there is only one atomic phase with 12 instances. This phase is then iteratively extended until the algorithm has generated the last phase which is always the complete dependency graph.

Of course, the algorithm does not have to be implemented as it was described in Fig. 4. In [4] we show that it can be implemented with an overall time complexity of $O(|\mathcal{V}|^4)$. This sounds still very much, but experimental results have

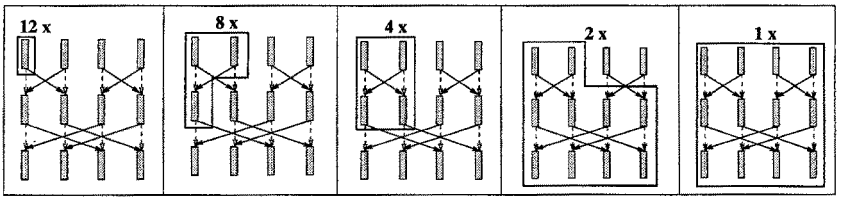

Fig. 5. Example run of the phase-detection algorithm

demonstrated that for two reasons the algorithm is usually much faster: First, the $\oplus$-operation most of the time produces larger phases than $A_i \cup B_i$ and second, not for every $j \in [1; |\mathcal{V}|]$ there exist A_i and B_i with $Occ(A_i \oplus B_i) = j$. Analyzing the dependency graph depicted in Fig. 5 for example, the algorithm generates only 26 occurrences of four valuable phases out of 12 IBs.

4 Experimental Results

Fig. 6 depicts the behavior of a distributed CG solver with different process mappings. We used two different clusters of four workstations each that were connected by a 32 Mbit/s wide area network. The bandwidth within the clusters was 155 Mbit/s. This configuration was used, because it demonstrates the absence of an optimal static mapping.

Both mappings shown in Fig. 6 are optimal for a particular range of problem sizes. Mapping A is optimal for problem sizes below 2^8 and mapping B is optimal for larger problems. Mapping A uses only one of the two clusters. Therefore, it is well suited for small problems, because the slow WAN connection cannot have any effect on the computation. For larger problems however, it is better to use all the computing power that is available. The optimal mapping for each problem size depends on the current network and computer configuration which changes dynamically in the shared environment. A process mapping tool that does not take advantage of any information about the program dynamics can only choose a mapping at random. Using dynamic program descriptions however, we first make a good guess according to the statistics stored in the knowledge base. I.e., if we have already seen a lot of runs with large problem sizes, we choose mapping B. Then, the application is monitored during the first few phases (iterations) and its behavior is matched against the knowledge base. From these

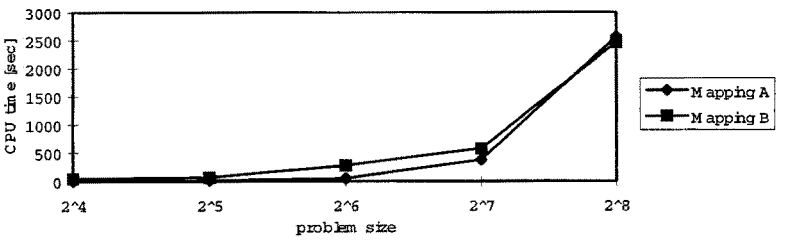

Fig. 6. Two different task mappings of a distributed CG solver

observations we decide, if B was the correct choice. Since the knowledge base provides estimations of the expected execution time, too, the load balancer can decide, if the application is likely to run long enough to be re-mapped. (More detailed experiments can be found in [5].)

5 Summary

We have demonstrated how characteristic features of a distributed program can be detected by analyzing dependency graphs. We presented a deterministic algorithm which provably detects all valuable characteristics within polynomial time. The algorithm considers results from previous runs of the same application and thereby produces better results each time the application is invoked. The extracted features are stored in a knowledge base which can be exploited by a variety of supporting tools like schedulers, load balancers, or routing systems for minimizing the response time. The consideration of characteristic phases enables these tools to adapt their optimization strategies to the dynamic communication behavior of the application. This is especially important for WAN-distributed applications arising in metacomputing environments.

References

1. M.A. Baker, C.G. Fox, and H.W. Yau. Cluster computing review. Technical report, Syracuse Univ., nov 1995.
2. W. Becker and G. Waldmann. Exploiting inter task dependencies for dynamic load balancing. In *Proc. HPCN-95*, pages 407–412. Springer LNCS 919, 1995.
3. T. Decker, R. Diekmann, R. Lüling, and B. Monien. Towards developing universal dynamic mapping algorithms. In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing*, pages 456–459, 1995.
4. J. Gehring. Dynamic program description as a basis for runtime optimization. Technical Report PC2/TR-002-97, Paderborn Center for Parallel Computing (PC^2), 1997.
5. J. Gehring and A. Reinefeld. Mars - a framework for minimizing the job execution time in a metacomputing environment. *Future Generation Computer Systems*, 12(1996)(1):87–99, may 1996.
6. J.A. Kaplan and M.L. Nelson. A comparison of queueing, cluster and distributed computing systems. Technical memo, NASA, jun 1994.
7. Soo-Young Lee and J.K. Aggarwal. A mapping strategy for parallel processing. *IEEE Transactions on Computers*, C-36(4):433–442, apr 1987.
8. A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, F. Ramme, T. Römke, and J. Simon. The MOL project: An Open, Extensible Metacomputer. In *Heterogenous computing workshop HCW'97 at IPPS'97*, 1997.
9. 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, feb 1993.