MODELS FOR PARALLEL WORKFLOW PROCESSING ON MULTI-CORE ARCHITECTURES

Thomas Rauber
Department of Computer Science, University of Bayreuth, Germany

Gudula Rünger
Department of Computer Science, Chemnitz University of Technology, Germany

Keywords: Workflow interoperability, multicore processors, parallel workflow execution.

Abstract: The advent of multi-core processors offers ubiquitous parallelism and a new source of powerful computing resources for all kinds of software products. However, most software systems, especially in business computing, are sequential and cannot exploit the new architectures. Appropriate methodologies and models to includes parallel features into business software are required.
In this article, we consider workflow software systems using explicit workflow descriptions and explore the possibilities to define parallel and concurrent executions in business processes for an implementation on multi-core systems. The article also presents a parallel execution model for workflows and addresses the scheduling of workflow tasks for multi-core architectures.

1 INTRODUCTION

Many advances in software technology and business computing are enabled by a steady increase in microprocessor performance and manufacturing technology. The performance increase will continue during the next years (Kuck, 2005; Koch, 2005). But technological constraints have forced hardware manufacturers to consider multi-core design to provide further increasing performance. For these multi-core designs, multiple simple CPU cores are used on the same processor die instead of a single complex CPU core. Multi-core designs today are used by many manufacturers like Intel, IBM or AMD and in 2007, 70% of new desktop computers were equipped with dual or quad core processors. This development will continue and it is expected that within a few years, a typical desktop processor provides 10s or 100s of execution cores which may be configured according to the needs of a specific application area (Kuck, 2005).

The design change towards multi-core processors represents a fundamental shift in computing, since the computing power of the new processors can only be utilized efficiently, if the application program provides coordination structures which enable a mapping of different execution threads to different cores. This is often described as a fundamental turning point in software development (Sutter and Larus, 2005; Sutter, 2005), in particular for mainstream software products.

To benefit from the performance increase provided by multi-core processors, the software has to be restructured towards a parallel execution. Such a restructuring provides the possibility to integrate new functionalities, e.g., by running useful tasks continuously, like an automatic backup utility ensuring that no work files are lost, or by including an intelligent workflow monitor anticipating user requirements and offering real-time information on demand (Reinders, 2006). This is also important for business software and the new development towards multi-core provides new challenges and opportunities.

The orchestration of services or tasks in business computing is often expressed as workflows which are used in a great variety of commercial software. Workflows are composed of tasks, and there may be dependencies between the tasks of a workflow. In business computing, a workflow is often processed sequentially by a workflow engine. Therefore, workflow software in its present form cannot benefit from the performance increase provided by the new multi-core architectures.

In this article, we explore the possibilities for a

Rauber T. and Rünger G. (2008).
MODELS FOR PARALLEL WORKFLOW PROCESSING ON MULTI-CORE ARCHITECTURES.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - ISAS*, pages 220-227
DOI: 10.5220/0001690402200227

parallel execution of workflows, based on different standard scenarios describing interactions and relationships between workflows. In particular, we consider the parallel execution of workflows for multi-core processors with a dense coupling of cores which enables a fast synchronization. The contributions of the article include:

- The exploration of the design space for a parallel execution of workflows on multi-core architectures; this includes the parallel execution of a single workflow as well as the concurrent execution of cooperating workflows with synchronization points. The exploration is based on standard interoperability models defined by the Workflow Management Coalition (WfMC).
- The provision of a concise parallel execution model for workflows capturing dependencies between workflow tasks; this model is based on an execution model for multiprocessor tasks and can be used as a basis for the scheduling of workflows on multi-core architectures.

The rest of the paper is organized as follows: Section 2 provides a short overview of multi-core architectures and describes expected implications for software development. Section 3 gives scenarios for the interoperability of workflows and discusses possibilities for a parallel execution for the different scenarios. Section 4 provides a parallel execution model for workflows and defines scheduling properties based on a cost model. Section 5 concludes the paper.

2 MULTI-CORE ARCHITECTURE AND IMPLICATIONS

The term multi-core processor architecture refers to architectures where two or more independent execution cores or computational engines are placed on a single processor chip (Intel, 2006). Multi-core architectures have been used for many years in the area of embedded systems and graphics engines. Since about two or three years these architectures are also used for commodity computing processors like Intel Core 2 architecture, AMD Opteron, or Sun T1. In this section, we give a short overview of multi-core architectures and discuss the implications of using these architectures for system and application software development.

2.1 Multi-core Architectures

Based on the steady increase of the number of transistors on a chip, the hardware manufacturers have been able to provide a significant performance increase during the last years: the average increase has been about 55% per year for integer and 75% per year for floating-point computations, based on benchmark programs like the SPEC benchmarks. The performance increase has been mainly achieved by the internal use of parallel processing like pipelined execution of instructions or the use of multiple functional units. But these traditional techniques have mainly reached their limits.

There are several driving factors that made the advent of multi-core architectures for commodity processors imperative (Dongarra et al., 2007; Held et al., 2006):

- The number of transistors on a chip doubles roughly every 18 - 24 months (Moore's law), thus providing more room for integrating functional units or improving the internal efficiency of instruction processing. The growing number of transistors on a chip is mainly achieved by increasing the transistor density. But this also increases the power density and heat production because of leakage voltage and power consumption.
- The speed of processor clocks also cannot be increased significantly because this also leads to an increase in power consumption and heat production beyond an acceptable limit.
- The number of pins of processor chips and therefore the bandwidth between CPU and main memory are reaching their limits, leading to a processor-to-memory performance gap ("memory wall"). This makes the use of high-bandwidth memory architecture with an efficient cache hierarchy necessary (Azimi et al., 2007).

To enable similar performance improvements as in the past, the processor manufacturers have started to put multiple cores on a single processor die. Today, dual-core and quad-core processors have become mainstream in desktop, mobile, and server platforms and according to Moore's law it is expected that the number of cores per processor will double every 18-24 months. Computer architects explore the design space for multi-core architectures with 10s or 100s of cores per processor. Important elements of these new architectures include (Azimi et al., 2007)

- a scalable, high-bandwidth, low-latency, and power-efficient interconnection to enable information exchange between computing elements and between computing elements and the memory system;
- a cache hierarchy to allow multiple computing elements to efficiently use the memory resources provided;

- a scalable, high-bandwidth memory architecture to bring data to the computing elements.

For the internal organization of multi-core processors, different design types have been proposed (Kogge, 2005):

- In **hierarchical designs**, multiple cores share multiple caches in a tree-like configuration such that the cache capacity increases towards the root. The root represents the off-chip external memory. This organization is used by many commodity multi-core processors like the Intel Core 2, the IBM Power, and the Sun UltraSPARC T1.
- In **pipelined designs**, data items are successively passed through different cores, and each core performs possibly different computation steps. Queues between the processing cores may be used to buffer data items before processing. Network processors and graphics engines are often based on this design and use up to 100s of cores.
- In **network designs**, an on-die network is used to connect the cores and their local caches with each other; data transfer between be cores is performed via the network. Basic requirements for the interconnection network are scalability to a large number of cores, partitionability for performance and fault isolation, fault tolerance to obtain a graceful degradation in the presence of faults, and support for testing and validation (Azimi et al., 2007). This design is especially useful for a large number of cores. It has been used for the Intel Tera-scale architecture with a 2D mesh topology.

For traditional uniprocessor, a cache hierarchy is used to reduce the average access time for data. For multi-core processors, a cache hierarchy is used with the same purpose, but different designs have been proposed and used. Usually, the first one or two levels in the cache hierarchy are private to each core. The last-level cache may be shared (e.g. Intel Core 2, IBM Power5, Sun UltraSPARC T1) or private (AMD Opteron, Intel Itanium Montecito). Also, hyper-threading technology may be used as an additional source of parallelism (Marr et al., 2002).

2.2 Implications for Software Development

The advent of multi-core architectures provides new opportunities and challenges for software development. In the past, improvements in semiconductor fabrication and processor architecture have produced a continuous increase in performance which could be exploited by sequential programs (Sutter and Larus, 2005). But for multi-core architectures, only concurrent or parallel applications can benefit from the increase in performance due to an increase in the number of execution cores per processor. Sequential desktop applications cannot run faster and may even run slower, since individual cores become simpler and use lower clock rates to reduce power consumption.

Many researchers consider the current hardware development towards multi-core as a fundamental turning point in software development (Sutter and Larus, 2005; Sutter, 2005), since new concurrent programming models have to be developed and applied also for mainstream software.

Programming models for multi-core architectures are often based on threads sharing a common address space. Threads can exchange data and information via shared variables and require synchronization to avoid deadlocks and race conditions. Lock synchronization is often considered as too low-level and other mechanisms like transactional memory have been proposed (Adl-Tabatabai et al., 2006; Asanovic et al., 2006). It is often argued that more abstract models are needed and should be embedded into programming languages and runtime systems in order to reach productivity also for multi-core systems.

Concurrency is a challenging issue in particular for client-side applications. For server applications, concurrency has often been applied to simultaneously respond to independent requests arriving from different clients or users. Web servers are a typical example. Often, a database is used for concurrent accesses to application data and data consistency is guaranteed by the database. Client applications, on the other hand, can be quite diverse and often perform a relatively small amount of computation. Thus, exploiting concurrency can only happen at a small granularity of computation and requires a dense coupling of the executing cores with a small synchronization overhead.

Client applications are often controlled by workflow executions. To obtain a concurrent execution, the potential for a parallel execution within the workflow has to be considered. This includes the parallel execution of a single workflow as well as the executions of workflows that are connected in some way. A systematic exploration in the following section addresses the potential of parallelism based on the workflow interoperability scenarios defined by the WfMC.

3 WORKFLOW INTEROPERABILITY

Workflows are a well-established concept for the combination of tasks or the orchestration of services.

Consequently, workflows are used in a great variety of software products from e-business, e-government, or e-science. The expressiveness of the workflows depends on the interoperability of workflows. The actual execution depends on the implementation of interoperability on parallel or distributed platforms. In this article, we concentrate on workflow software based on explicit workflows executed by a workflow engine embedded in a workflow management system. This approach has been standardized by the WfMC. After summarizing standard interoperability patterns, we explore implications for their implementation on recent and future multi-core systems.

3.1 Interoperability Scenarios

The Workflow Management Coalition (WfMC) has defined different scenarios for the interoperability of workflows which can operate at different levels from simple task passing between workflows to a complete interchange of process definitions (Hollingsworth, 1995). In this subsection, we give a short overview of the scenarios and discuss in the next subsection possibilities for an implementation on recent multi-core architectures.

Chained Service Model. This model allows the chaining of two workflows in the sense that a workflow A produces a work item (process instance, activity, or data) which is passed to a workflow B which then operates independently from A with no further synchronization, see Fig. 1 for an illustration. The connection points can be anywhere within A and B.

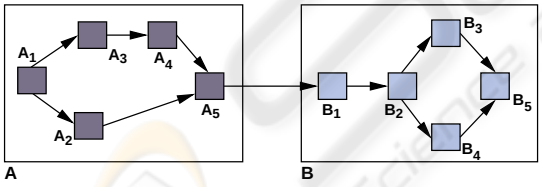

Figure 1: Illustration of chained service model.

Nested Subprocesses Model. This model allows the encapsulation of a single task of a workflow A and the migration of this task for execution in a different workflow domain B, see Fig. 2 for an illustration. Thus, there exists a hierarchical relationship between workflow A and the encapsulated task which is executed in workflow domain B. This hierarchical relationship may be continued across several levels and may also capture recursive relationships.

Peer-to-Peer. This model supports the execution of a workflow across multiple workflow engines by assigning different activities of the workflow to different servers, see Fig. 3 for an illustration. This assignment may require to pass application data to the executing server. The workflow engines must be able to cooperate and coordinate their execution and to exchange data as necessary; security and recovery issues have to be taken into consideration for the data exchange.

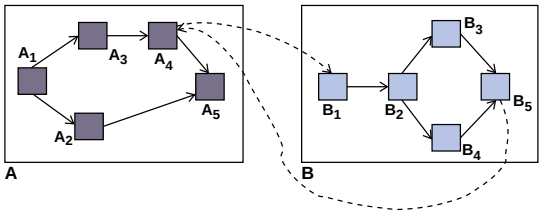

Figure 2: Illustration of nested subprocess model.

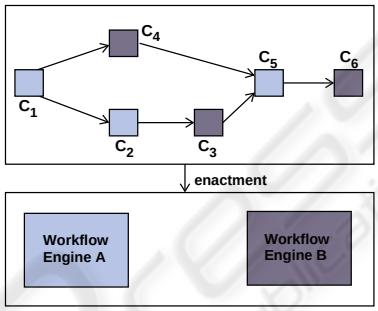

Figure 3: Illustration of peer-to-peer model.

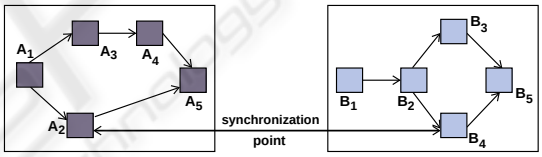

Figure 4: Illustration of parallel synchronized model.

Parallel Synchronized Model. This model allows the parallel and independent execution of two workflows using separate enactment services. The execution of the two workflows can be coordinated by using synchronization points between predefined tasks, see Fig. 4 for an illustration. Synchronization requires that a common event is generated when the workflow executions reach the predefined synchronization points within their execution sequences.

3.2 Parallel Execution of Workflows

The parallel computing resources of multi-core processors can be exploited by a parallel execution of workflows. The possibilities for a parallel execution

depend on the interoperability model used for the interaction between workflows. For the different models, we propose different parallel execution scenarios.

Pipelined Execution of Workflows. The chained service model allows a pipelined execution of workflows. If a workflow A produces data for a workflow B, A and B can be executed concurrently if A and B work on different data subsets: while B works on the data set produced by A in the previous time step, A starts to execute on the next data set, see Fig. 5 for an illustration. There has to be a synchronization point to ensure that B does not start its execution before A has entirely finished its computations on the data set. Moreover, the data set computed by A has to be made available to B. If the executing cores share a common address space, this can also be performed by synchronization. Otherwise, communication has to be performed.

To avoid waiting times when transferring data from A to B, the execution of A and B should take about the same time. The pipelined execution model can be extended to an arbitrary sequence of chained workflows, such that each workflow is executed on a different core by a separate workflow engine.

Parallel Execution of a Single Workflow. The parallel execution of a single workflow can be obtained by assigning the tasks of the workflow to different workflow engines running on different cores of a multi-core processor.

Two tasks *t* and *s* of the same workflow A can be assigned to different workflow engines for a parallel execution, if there is no control or data dependence

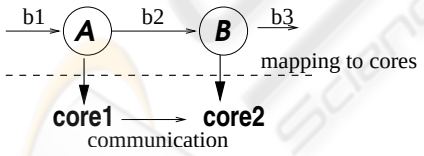

Figure 5: Pipelined workflow execution for the chained service model.

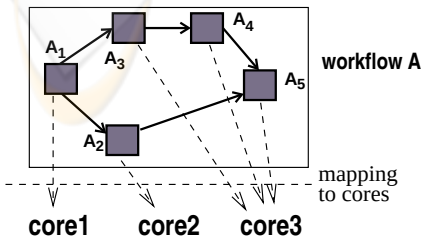

Figure 6: Parallel execution of a single workflow.

relation between *t* and *s* and if both *t* and *s* have to be executed, i.e. *t* and *s* are not in an exclusive choice relation. There is a data dependence between *t* and *s*, if *t* produces data which is used by *s* (or vice versa) or if *t* and *s* manipulate the same data. The maximum degree of parallelism of a workflow W can be defined as the maximum number of tasks of W that can be executed in parallel. This parallel execution model can be applied to separate workflows in the chained service model, the nested sub-process model, the peer-to-peer model, and the parallel synchronized model, see Fig. 6 for an illustration. The internal structure of the workflows can be described by workflow patterns like sequence, parallel split or exclusive choice (van der Aalst et al., 2003). For each of these patterns, a process-based description using CSP (Communicating Sequential Processes) (Hoare, 1985) can be given (Wong and Gibbons, 2007). Section 4 presents a parallel execution model for the parallel execution of the tasks of a single workflow.

Parallelism can also be exploited for the execution of a single task if its execution involves a significant amount of computation as it might be the case for tasks in an e-science workflow. In this case, the task is assigned to a group of execution cores instead to a single core. To exploit this kind of parallelism, the task has to have a suitable internal computation structure which allows a parallel execution. This can be a data parallel structure such that independent execution units work on different data, or if the task exhibits an internal task structure such that independent computations can be executed in parallel.

The parallel execution of a single task is also possible if the nested subprocess model is used: if a task of workflow A is encapsulated and executed as a separate workflow B, the execution of B can be performed in parallel, if B has a suitable structure with independent sub-tasks.

Concurrent Execution of Different Workflows. The parallel synchronized model allows the concurrent execution of different workflows using workflow engines on different execution cores. The workflow engines can work independently executing different workflows A and B. At synchronization points between predefined tasks of A and B, a common event is generated which stops the execution of A until the execution of B reaches the predefined synchronization point, and vice versa. Fig. 7 shows an illustration.

Synchronization may also be required when different workflows that are executed concurrently on different cores of the same processor access the same data sets that are stored locally; this access has to be synchronized if write accesses are performed. If all

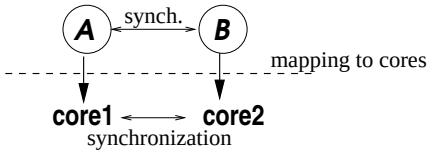

Figure 7: Concurrent execution of workflows with synchronization.

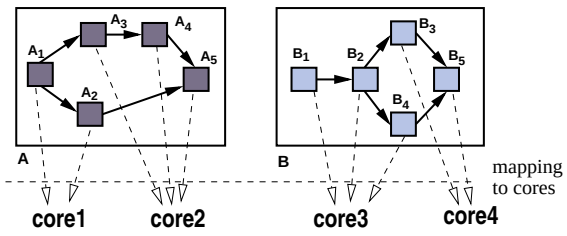

Figure 8: Mixed execution of workflows.

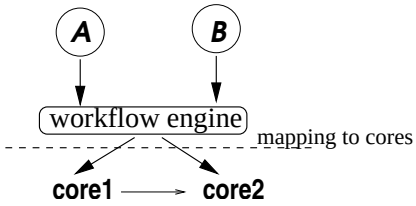

Figure 9: Parallelism within a workflow engine.

data accesses are performed via a database, the synchronization is usually performed in the database.

Mixed Executions. The different modes for a parallel execution can be mixed in different ways. For example, a concurrent execution of workflows can be mixed with a parallel execution of a single workflow in the sense that different groups of cores work on different workflows where each group executes its workflow in parallel by distributing the tasks to different cores for execution, see Fig. 8.

Similarly, a pipelined execution of workflows can be combined with a parallel execution of a single workflow by using groups of execution cores for each stage of the execution pipeline.

Parallelism within the Workflow Engine. The parallel execution of a single workflow can also be performed if the workflow engine is multi-threaded and can therefore execute different tasks by concurrent threads of control, see Fig. 9. Again, dependencies between tasks have to be taken into account. This kind of parallelism requires a parallel implementation of the workflow engine itself.

4 MODEL FOR PARALLEL WORKFLOW EXECUTION

This section gives a concise description of workflows and presents an execution model which can be used for the scheduling of the tasks of a workflow to different cores.

There are some similarities between workflow executions and the execution of multiprocessor task (M-task) graphs from scientific computing. Both problems deal with parallel execution of tasks with dependencies where each task can in principle be executed by several cores in parallel. The dependencies defined may require a sequential execution of specific tasks. There are several differences between workflows and M-task graphs: M-task scheduling usually assumes a distributed address space and, therefore, the execution model has to take redistribution operations into consideration, see (Rauber and Rünger, 1998) for a formal description. For workflow execution on multicore architectures, there is a shared address space and no communication for redistribution operations are required. Moreover, workflows may contain exclusive choice patterns, i.e. only one of two or more tasks without dependencies has to be executed. This does not occur in M-task graphs. In the following, we formulate the parallel execution of workflows with the mechanisms used for M-task graphs employing a useful adaption (Rauber and Rünger, 1998). This allows us to apply the scheduling theory for M-task programs to the parallel execution of workflows.

A workflow consists of a collection of nodes $J_1, \ldots, J_\Omega$ where each node represents a service or task to be executed. Nodes need not to be independent of each other but may be related by a *precedence* relation $\triangleright$ which is caused by a control or data dependency:

$J_i \triangleright J_k$ if there is a dependence from J_i to J_k.

The execution of different tasks of a workflow on different execution units has to guarantee that the input data required by a node J_k are available when the execution of J_k starts, i.e.:

(i) The input data must have been produced by predecessor J_i of J_k with $J_i \triangleright J_k$.

(ii) The input data must be available in the expected format, i.e. each execution unit must have the data that it needs for the execution of its nodes.

Item (i) is satisfied when executing the nodes according to the precedence relation. To satisfy (ii), a re-organization between nodes J_i and J_k with $J_i \triangleright J_k$ might be necessary to re-arrange the data that have been output by J_i in order to get the format expected by J_k.

A workflow can be represented as a DAG $G = (V,E)$. The nodes of the DAG are the nodes $V =$

$\{J_1,\ldots,J_\Omega\}$ and the edges are defined according to the precedence relation (i.e., $(J_i,J_k)\in E$ if $J_i \triangleright J_k$). In the following we identify a workflow with its DAG representation. We assume that $J_1,\ldots,J_\Omega$ are sorted topologically, i.e. if $J_i \triangleright J_k$ then $i<k$.

4.1 Parallel Execution of Workflows

A node of the workflow DAG can be executed on a single execution core or on multiple execution cores, if the node exhibits an internal structure as it is, e.g., the case for the nested sub-process model. In the following, we consider the general case that multiple execution cores may be used for a single node.

To capture the parallel execution of workflows, the nodes of a workflow DAG are attached with *cost functions* T describing the global execution time as a function of the number of execution cores $q\in\{1,\ldots,p\}$ executing $J\in V$:

$$T : V\times[1,\ldots,p]\rightarrow I\!R$$

$$T(J,q)=T_{comp}(J,q)+T_{admin}(J,q)$$

where $T_{comp}(J,q)=T_{comp}(J,1)/q+T_{overhead}$ is the core execution time. $T_{overhead}$ denotes the overhead caused by the parallel execution (e.g. waiting times) of a node on several execution cores. $T_{admin}(J,q)$ describes the costs for the administration which is necessary to arrange the execution of a node on several execution cores including the internal data accesses and communication.

In addition, there may be costs to perform re-arrangement between cooperating nodes. Such costs arise, e.g., if cooperating nodes require different arrangements for the same data. The necessary re-arrangements are performed by specific re-arrangement operations that have to be performed between the execution of nodes J and J' for $J\triangleright J'$. The costs are captured by a *re-arrangement function* $Re(J,J')$. The costs of the re-arrangement functions $T_{Re}: V^2\times[1,...,p]^2\rightarrow I\!R$ are attached to the edges of the workflow DAG. For $J\triangleright J'$, $T_{Re}(J,J',q,q')$ describes the re-arrangement costs between J and J' for the case that J and J' are executed by q and q' execution cores, respectively. $T_{Re}(J,J',q,q')=0$ holds if no re-arrangement is needed between J and J'.

4.2 Scheduling of Workflow DAGs

The parallel execution of a workflow can be captured by a *scheduling function* $\mathcal{S}$ performing an assignment of an execution time interval and a subset Π_i of the executing cores to each node. The execution time interval begins at a start time s_i:

$$\mathcal{S}:\{J_1,\ldots,J_\Omega\}\rightarrow I\!R\times 2^P$$

$$\mathcal{S}(J_i)=(s_i,\Pi_i)$$

Definition 1. (Feasible Scheduling) *A schedule for a workflow is feasible if for all $i,j=1,\ldots,\Omega$, the following conditions hold:*

(a) $s_i+T(J_i,|\Pi_i|)+T_{Re}(J_i,J_j)\leq s_j$ *for* $J_i\triangleright J_j$

(b) If $[s_i,s_i+T_g(J_i,\Pi_i)]\cap[s_j,s_j+T_g(J_j,\Pi_j)]\neq\emptyset$ *then* $\Pi_i\cap\Pi_j=\emptyset$ *where*

$$T_g(J_i,\Pi_i)=T(J_i,|\Pi_i|)+\sum_{Re\in\mathcal{RE}}T_{Re}(J_i,J_j).$$

$\mathcal{RE}$ contains the re-arrangement operations between J_i and its successors $\{J_j|J_i\triangleright J_j\}$.

Condition (a) expresses that dependent nodes have to be executed one after another according to the precedence relations. Condition (b) expresses that independent nodes must be executed on disjoint sets of execution cores if their execution time intervals overlap. This includes that the number of active execution cores can never exceed the total number of execution cores. The execution time intervals also comprise the time for the execution of re-arrangement functions to establish the correct data arrangement for subsequent nodes.

The total execution time Γ of a feasible schedule $\mathcal{S}$ for a workflow DAG is

$$\Gamma(\mathcal{S})=\max_{i=1,\ldots\Omega}\left\{s_i+T_g(J_i,\Pi_i)\right\}$$

The *scheduling problem* is to determine an optimal feasible schedule $\mathcal{S}$, i.e., a feasible schedule that minimizes the objective function $\Gamma(\mathcal{S})$.

A linear chain of nodes of a workflow always has to be executed sequentially one after another; usually the same execution unit should be used to avoid re-arrangement operations. We can therefore normalize workflow DAGs:

Definition 2. (Normalized DAG) *Let $G=(V,E)$ be a workflow DAG. A linear chain of nodes is a subgraph G' of G consisting of a subset of nodes $\{J'_1,\ldots,J'_\omega\}\subset V$ such that for $i=1,\ldots,\omega-1$:*

if there exists J with $J_i\triangleright J$ *then* $J=J_{i+1}$.

A maximum linear chain of nodes is a linear chain which is no proper subgraph of another linear chain of the DAG G. A normalized workflow DAG is a workflow DAG without linear chains of nodes.

By a suitable fusion of nodes, each workflow DAG can be transformed into a *normalized workflow DAG* that does not contain linear chain of nodes.

Definition 3. (Normalization of workflow DAGs) *Let G be a workflow DAG with m maximal linear chains $C_1,\ldots,C_m$. Each of the chains $C_j=\{J'_1,\ldots,J'_{\omega_j}\}$ is*

identified with one new node $\tilde{J}_j$, $j = 1,\ldots,m$, *which has costs*

$$\tilde{T}(\tilde{J}_j, |\Pi_j|) = \sum_{i=1}^{\omega_j} T(J'_i, |\Pi_j|).$$

The normalization of DAGs is a restriction of the potential solution space of feasible schedules. The restriction has the effect that the same execution cores are assigned to each node of a linear chain in the original DAG. This is a reasonable assumption, because a change in the executing cores may require a re-arrangement of data. The costs of such a re-arrangement is usually much larger than the benefits of a change in the number of executing processors.

Normalized workflow DAGs can be used for the scheduling of workflows on multiple execution cores, as they are provided by multi-core architectures. The model also captures a parallel workflow execution on distributed architectures where each site provides processors with multiple execution cores. Communication abstractions for the distributed execution of business processes are described in (Aldred et al., 2007).

5 CONCLUSIONS

The widespread use of multi-core processors provides new computing possibilities, since software developers can improve their applications with new functionalities which can be provided by separate threads of control running on a separate core of the processor. To exploit this improvement, parallel programming techniques must be applied. This is also the case for business software whose execution is often based on workflow model.

In this article, we have explored different possibilities for a parallel execution of workflow based on different interoperability models. The article provides a detailed model for the parallel execution of workflows and considers the scheduling of workflows based on this model.

REFERENCES

Adl-Tabatabai, A., Kozyrakis, C., and Saha, B. (2006). Unlocking concurrency. *ACM Queue*, 4(10):24–33.

Aldred, L., van der Aalst, W., Dumas, M., and ter Hofstede, A. (2007). Communication Abstractions for Distributed Business Processes. In *19th Int. Conf. on Advanced Information System Engineering (CAiSE 2007)*, Springer LNCS 4495, pages 409–423.

Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., and Yelick, K. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Azimi, M., Cherukuri, N., Jayasimha, D., Kumar, A., Kundu, P., Park, S., Schoinas, I., and Vaidya, A. (2007). Integration Challenges and Tradeoffs for Tera-scale Architectures. *Intel Technology Journal*, 11(03).

Dongarra, J., Gannon, D., Fox, G., and Kennedy, K. (2007). The Impact of Multicore on Computational Science Software. *CTWatch Quarterly*, 3(1).

Held, J., Bautista, J., and Koehl, S. (2006). From a Few Cores to Many – A Tera-Scale Computing Research Overview. Intel White Paper, Intel.

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice Hall.

Hollingsworth, D. (1995). The Workflow Reference Model. Technical report, The Workflow Management Coalition.

Intel (2006). Intel Multi-Core Processor Architecture Development Backgrounder. Technical report, Intel White Paper.

Koch, G. (2005). Discovering Multi-Core:Extending the Benefits of Moore's Law. Intel White Paper, Technology@Intel Magazine.

Kogge, P. (2005). An Exploitation of the Technology Space for Multi-Core Memory/Logic Chips for Highly Scalable Parallel Systems. In *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*. IEEE.

Kuck, D. (2005). Platform 2015 Software-Enabling Innovation in Parallelism for the next Decade. Intel White Paper, TechnologyIntel Magazine.

Marr, D., Binus, F., Hill, D., Hinton, G., Konfaty, D., Miller, J., and Upton, M. (2002). Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15.

Rauber, T. and Rünger, G. (1998). Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45:483–503.

Reinders, J. (2006). Sea Change in the Software World. *Intel Software Insight*, pages 3–8.

Sutter, H. (2005). The free lunch is over – a fundamental turn toward concurrency in software. *Dr.Dobb's Journal*, 30(3).

Sutter, H. and Larus, J. (2005). Software and the Concurrency Revolution. *ACM Queue*, 3(7):54–62.

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., and Barros, A. (2003). Workflow Pattern. *Distributed and Parallel Databases*, 14(3):5–51.

Wong, P. and Gibbons, J. (2007). A Process-Algebraic Approach to Workflow Specification and Refinement. In *Proceedings of 6th International Symposium on Software Composition*.