Flexible, Efficient and Robust Algorithm for Parallel Execution and Coupling of Components in a Framework

Gábor Tóth [a]

[a] *Center for Space Environment Modeling University of Michigan, 2455 Hayward, Ann Arbor, MI 48109*

Abstract

We describe a general algorithm suitable for executing and coupling components of a software framework on a parallel computer. The requirements of a flexible, efficient and robust algorithm are defined precisely, and the motivation for the requirements is demonstrated on several examples. In short, the requirements are the following: (i) the algorithm should allow arbitrary distribution of processors among the components, (ii) it should allow arbitrary coupling schedule between the components, (iii) it should not use any inter-processor communication other than already required by the components and their couplings, and (iv) it should never get into a dead-lock. We show that the proposed algorithm based on the Temporal and Predefined Ordering of Tasks (TPOT) satisfies all these requirements. The TPOT algorithm has been implemented in the Space Weather Modeling Framework. The flexibility and efficiency of the algorithm is demonstrated with several examples.

Key words: software framework; concurrent execution; parallel algorithm; space weather

1 Introduction

General frameworks are becoming more and more important in the numerical simulation of complex phenomena. Just in the areas of geophysics and plasma physics there are several frameworks under development [1–8]. Frameworks provide a way of executing and coupling multiple software components.

[1] This work has been partially supported by the Hungarian Science Foundation (OTKA, grants No. T037548 and T047042)

Each software component is responsible for solving some subtask in a relatively independent manner, the data exchange between the framework and the components and/or between the components should happen via standardized interfaces. The framework concept allows independent development of the software components, it allows replacement of a component with alternative versions, and it allows assembling a coupled system which can model more complex phenomena than the individual components are able to.

The development of the general execution and coupling algorithm, which is the subject of the present paper, has been motivated by the development of the Space Weather Modeling Framework (SWMF) [1,2]. In general terms, space weather is the dynamic interaction of solar and terrestrial phenomena which can affect human life. The SWMF aims at simulating and predicting space weather by combining a multitude of components, which model various physics domains spanning from the solar corona to the heliosphere, magnetosphere, ionosphere and upper atmosphere. The models have mostly been developed independently over several years, they use different numerical schemes to solve different equations. The components use different programming styles, which excludes combining them into a tightly integrated software.

In the SWMF the execution of the components is tied together by the notion of *simulation time*. In the typical simulations all the components should model the same time interval. They should interact when their simulation times are the same (or close). In general it is allowed to run the components in different time intervals. It is also possible that some (one directional) interaction between two components has some delay in time, but that can be hidden from the framework if the sending or the receiving component stores the data until it is needed. In other applications, the simulation time can be replaced with any monotonically growing quantity, like iteration number, or the amount of data processed, or the line number in some text etc. The only assumption we make is that there is a monotonically increasing quantity which measures the progress of the components and which can be used for scheduling the interaction between the components. For sake of simplicity we will refer to this quantity as *simulation time*, or simply *time*.

For sake of a clear distinction, the actual time of running the framework will be referred to as *wall clock time*, while the *CPU time* is the wall clock time multiplied by the number of CPU-s working on some problem. A parallel component shows ideal *speed up* if the CPU time is independent of the number of CPU-s, ie. the wall clock time is inversely proportional with the number of CPU-s. In this paper the word “scaling” will be used in this meaning as well, i.e. speed up for a fixed problem size.

One of the many challenges in creating a framework is the design of a flexible, efficient and robust algorithm for the execution and coupling of the components.

- **Flexibility** means that the framework should allow more or less arbitrary processor layout and coupling schedules between the components.
- **Efficiency** means the minimization of the total execution time of the coupled system on a given number of processors. An efficient algorithm should allow parallel components to execute on an ideal number of processors and it should minimize the idle time of processors.
- **Robustness** means that the execution should always complete, the processors should not get into a dead-lock, when all processors are waiting for information from another processor.

In the next session the problem is defined in mathematical terms, and we show how some simple and/or naive approaches fail to satisfy the above criteria. In session 3 the proposed algorithm, named Temporal and Predefined Ordering of Tasks (TPOT, to be pronounced like 'tea-pot') is described and its properties are analyzed. In particular, it is proven rigorously that the TPOT algorithm is robust, i.e. it never gets into a dead-lock. The usefulness and efficiency of the algorithm will be demonstrated with some example runs in Section 4. The final section concludes the paper with some generalizations and outlook to related problems.

2 Definition of the Problem

This section defines the problem of parallel execution and coupling of components in mathematical terms. The clear definitions will help to discuss and solve the various difficulties which arise in a complex framework.

2.1 Basic Definitions

To facilitate a short and accurate description of the problem, we begin with some basic definitions of processor layout, time stepping, coupling and the initial and final state of the simulation.

2.1.1 Components and Processor Layout

The set of components is defined as

$$C := \{c \mid 1 \leq c \leq N_C, \quad c \in \mathbb{N}\} \tag{1}$$

where N_C is the number of components. The identification of the components with an integer makes the ordering of components simple. The set of processors available for the framework is defined as

$$P := \{p \mid 1 \leq p \leq N_P, \quad p \in \mathbb{N}\} \tag{2}$$

where N_P is the number of processors. Each component is running on a non-empty subset of the processors

$$P_c \subset P \qquad\qquad c \in C \tag{3}$$

For serial components P_c has exactly one element, while for parallel components $N_{P_c} \geq 1$. Similarly each processor is working on a subset of the components

$$C_p \subset C \qquad\qquad p \in P \tag{4}$$

In principle C_p can be empty, which means that some processors do not take part in the execution of the components. If the number of components $N_{C_p} \leq 1$ for all processors, then the processor layout contains no overlap. In general, however, N_{C_p} can range from 0 to N_C.

2.1.2 Time Stepping and Coupling

At any given point in the execution of the framework, each component has a unique simulation time

$$t_c \in \mathbb{R} \qquad\qquad c \in C \tag{5}$$

which is known by the processors in the subset P_c. This simulation time must be advanced by the component on all the processors in a consistent manner. The advancing of the component by one time step on processor $p \in P_c$ consists of the following stages:

$$S_c := \text{start advancing component } c \tag{6}$$
$$F_c := \text{finish advancing component } c \tag{7}$$
$$t_c := t_c + \Delta t_c \tag{8}$$

where $\Delta t_c > 0$ is the same for all processors $p \in P_c$, but it may vary from time step to time step. The first stage acts as a block on processor p: once the time step is initiated no other work can be done until it completes. Since the components normally perform communication among the processors in P_c

during the time step, the time step will complete if and only if S_c is executed on all $p \in P_c$:

$$F_c \text{ if and only if } S_c \text{ on all } p \in P_c \tag{9}$$

The coupling between two components can be identified with a set of two component indexes. Not all pairs of components are coupled, so it is useful to introduce the set of couplings K as a subset of all possible pairs:

$$K \subset \{\{c,d\} \mid c,d \in C\} \tag{10}$$

Note that the interaction of more than two components can be replaced with a series of pairwise couplings which take place at the same simulation time.

The coupling between two components is scheduled to take place at time

$$t_{\{c,d\}} \in \mathbb{R} \qquad \{c,d\} \in K \tag{11}$$

which must be known on the processors p in

$$P_{\{c,d\}} := P_c \cup P_d \tag{12}$$

The coupling consists of the following substeps:

$$S_{\{c,d\}} := \text{start coupling components } c \text{ and } d \tag{13}$$
$$F_{\{c,d\}} := \text{finish coupling components } c \text{ and } d \tag{14}$$
$$t_{\{c,d\}} := t_{\{c,d\}} + \Delta t_{\{c,d\}} \tag{15}$$

where $\Delta t_{\{c,d\}} > 0$ is the same for all processors $p \in P_{\{c,d\}}$, but it may vary from one coupling to the next one. The first stage acts as a block on processor p: once the coupling is initiated no other work can be done until it completes. Since the components perform communication among the processors in $P_{\{c,d\}}$ during the coupling,

$$F_{\{c,d\}} \text{ if and only if } S_{\{c,d\}} \text{ on all } p \in P_{\{c,d\}} \tag{16}$$

2.1.3 Relations between time stepping and coupling

A necessary condition for running a component is that its simulation time is less than all the coupling times which involve the component. Formally the condition is

$$S_c \text{ only if } t_c < t_{\{c,d\}} \qquad \forall \{c,d\} \in K \tag{17}$$

The time step Δt_c is normally set by the component, and it may vary during the execution due to numerical stability conditions or accuracy requirements. It depends on the implementation if the component is allowed to exceed the next coupling time after its time step is finished, or the time step is truncated such that t_c becomes equal with the closest coupling time, i.e. the time step is limited by the condition

$$\Delta t_c \leq \min\{t_{\{c,d\}} \mid \{c,d\} \in K\} - t_c \tag{18}$$

If the above condition in enforced, the couplings will take place when both components have the same simulation time as the coupling time, i.e. $t_c = t_d = t_{\{c,d\}}$, which may improve the temporal accuracy of the solution. On the other hand, not all components may be able to limit their time steps, and it can also be inefficient if a large fraction of the time steps are reduced due to frequent couplings. In the SWMF implementation, condition (18) is generally enforced, but some components can be exempt, which is determined as an input parameter.

A necessary condition for coupling two components is that their simulation times have reached (or exceeded) the coupling time. For the coupling of components c and d on a processor $p \in P_c$ the condition is

$$S_{\{c,d\}} \text{ only if } t_{\{c,d\}} \leq t_c \tag{19}$$

while on processors $p \in P_d$ the condition is

$$S_{\{c,d\}} \text{ only if } t_{\{c,d\}} \leq t_d \tag{20}$$

Finally if there is an overlap between the layouts, then on processors $p \in P_c \cap P_d$ both conditions must hold.

2.1.4 Initial and Final Time

The framework starts the execution from simulation time t_{start} and finishes execution when all components reach (or exceed) the final time t_{finish}. This time interval may be divided into smaller parts, which we may call *sessions* which are delimited by some action that requires communication between all the components. In the SWMF, for example, the set of active components and the input parameters can be modified at the beginning of each sessions.

A session starts at time $t_{\min}$ and stops at $t_{\max}$. Initially

$$t_c \geq t_{\min} \qquad c \in C \tag{21}$$

$$t_{\{c,d\}} \geq t_{\min} \qquad \{c,d\} \in K \tag{22}$$

The framework successfully finishes the session if all components reach the final time and all couplings are complete:

$$t_c \geq t_{\max} \qquad \forall c \in C \tag{23}$$
$$t_{\{c,d\}} \geq t_{\max} \qquad \forall \{c,d\} \in K \tag{24}$$

It depends on the application if couplings scheduled for the final time $t_{\max}$ should be performed or not. Here we assume that such final couplings are not needed, but this is not a crucial assumption.

2.2 Flexibility and Efficiency Considerations

The number of processors selected for a component is restricted by several reasons:

- Not all components are parallel;
- The parallel components may require a minimum number of processors to fit into the memory;
- Parallel components may not work on arbitrarily large number of processors;
- Parallel components do not scale perfectly to arbitrary number of processors.

Therefore the algorithm has to allow that components use an arbitrary number of processors out of the total number used by the framework. Given limited computer resources the algorithm must also allow the overlap of the layouts of the components as well. The coupling of the components should be very flexible too. Some components need to communicate very often, others need to communicate less frequently. The optimal communication pattern depends on the particular simulation and cannot be hard coded.

The computational cost (total CPU time) of the components to simulate a certain interval of simulation time can vary by orders of magnitudes. Depending on the total number of processors, the optimal distribution of processors among the components vary.

2.2.1 Concurrent Execution

On a large number of processors we may split the set of processors P into disjoint parts and assign a number of processors to each component proportional to their computational cost as shown in the left panel of Figure 1. This way the wall clock time can be approximately equal, which minimizes the

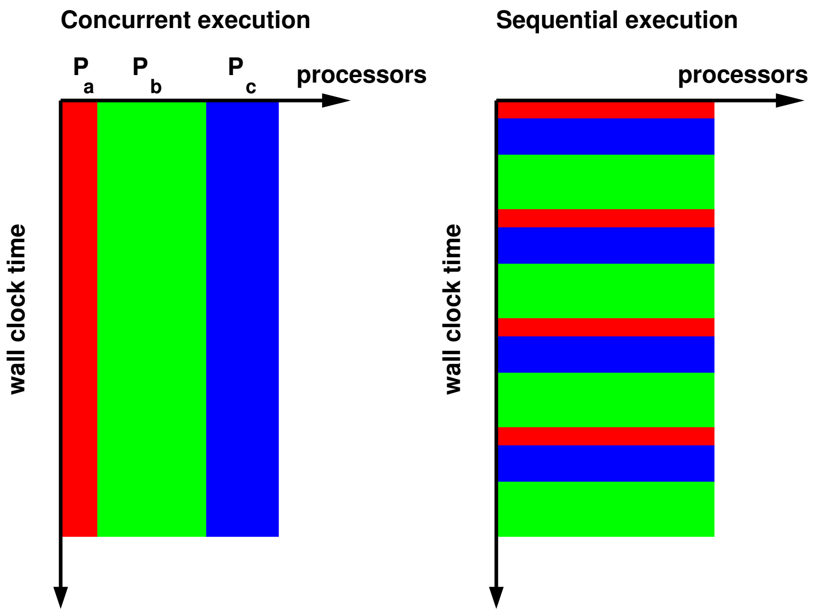

Fig. 1. Comparison of the purely concurrent (left) and purely sequential (right) execution models. The computational work is color coded by the component executing: a (red), b (green), c (blue). The areas covered by the three colors are the same in both execution models (which assumes perfect scaling)

idling time, and gives approximately optimal performance. On small number of processors, however, it may be impossible to split the number of processors proportionally to the computational work, or the component may need more memory than available on the subset of processors which would be optimal for speed.

2.2.2 Sequential Execution

On a smaller computer we may assign all the processors to all the components, so they can execute sequentially (in turns) as depicted in the right panel of Figure 1. As long as all components scale well to the number of available processors, this strategy can give nearly optimal performance. On large number of processors, however, the scaling will break down, and the sequential approach becomes inefficient.

2.2.3 Mixed Execution

When many components are included into the framework, each having its limitations for parallel execution, the optimal execution model on a limited number of processors is a mixture of the sequential and concurrent execution strategies. It is a very non-trivial problem to find the optimal processor layout and coupling schedule, and this paper does not attempt to solve the

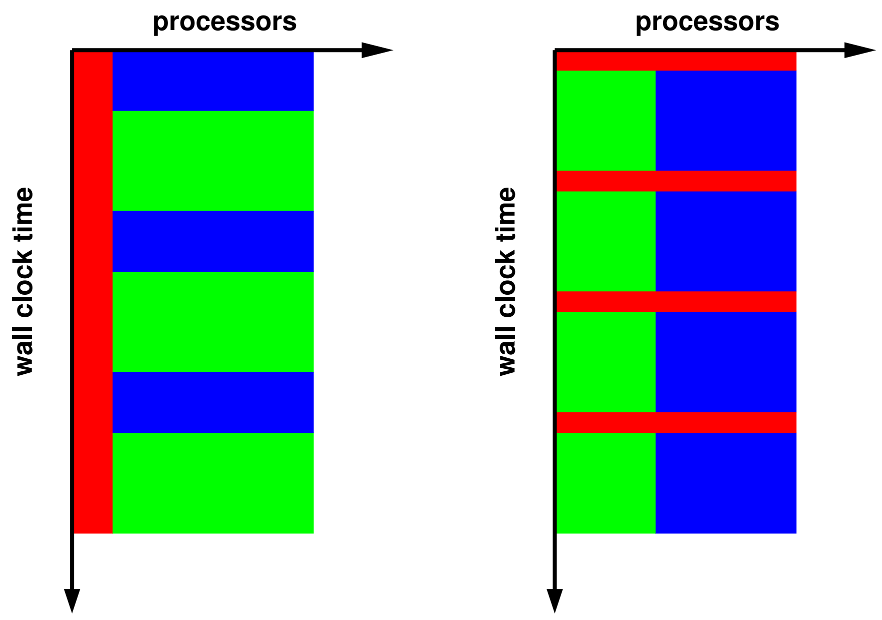

Fig. 2. Two examples of mixed execution models. The left panel shows how a serial component (red) can be run along two large parallel components (green and blue) which cannot run concurrently. The right panel shows how a very large component (red) which requires all the processors can be run together with two parallel components (green and blue) which can run most efficiently on a subset of the processors.

optimization problem. On the other hand an efficient algorithm must allow a mixture of concurrent and sequential execution, so that the optimal speed can be achieved. Figure 2 shows two simple examples for mixed execution model.

2.2.4 Minimal Communication

All communications between processors act as a block of execution. In case the processors need different times to reach the communication point, the block leads to idling processors, which is a waste of computational resources. Therefore an efficient algorithm must minimize the inter-processor communication. Ideally the framework should not add any extra communication relative to the unavoidable communication required by the execution and coupling of components. This means that an efficient algorithm should use locally available information only.

2.2.5 Fine Grained Time Stepping

It is tempting to design an algorithm which allows the components to progress uninterrupted as long as they do not require coupling with other components. This approach corresponds to a *coarse grained time stepping.* Although such an approach is conceptually simple, it can lead to an inefficient execution pattern. Assume, for example, that components b and c need to be coupled at

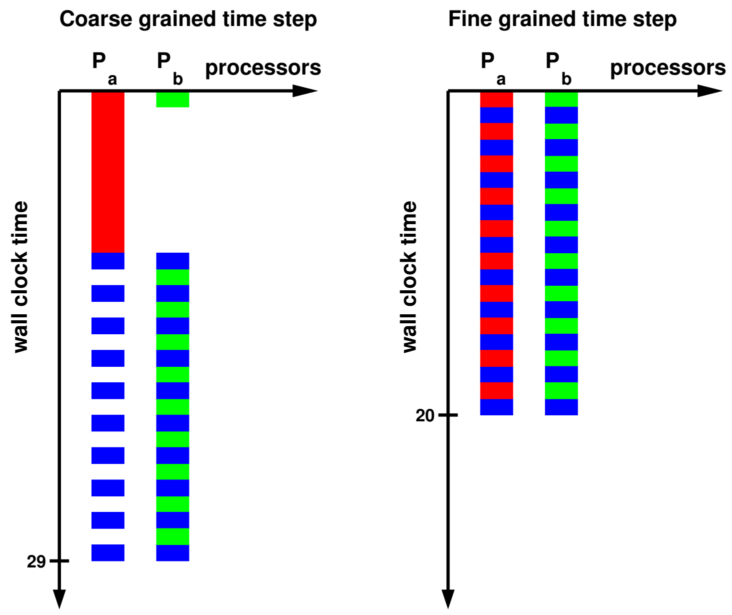

Fig. 3. Comparison of the efficiency of coarse grained (left) and fine grained (right) time stepping. The computational work is color coded by the component executing: a (red), b (green), c (blue). Components a and b use the disjoint processor subsets P_a and P_b, respectively, while component c uses the union set $P_a \cup P_b$. For coarse grained time stepping, the processors are idle (no color) for a significant fraction of the wall clock time and the overall execution time (29) is slower than for the fine grained time stepping (20).

every second of simulation time, while component a is coupled (for example with component b) at every 10 seconds only. Assume further that components a and b run on disjoint processor sets P_a and P_b ($P_a \cap P_b = \emptyset$), while component c uses all the processors used by a and b ($P_c = P_a \cup P_b$). For sake of simplicity let us take the wall clock time spent to be equal to the simulated time interval for all three components.

With coarse grained time stepping, component a is advanced by 10 seconds at a time and as can be seen from the left panel of Figure 3, it will take 29 seconds wall clock time to advance all 3 components by 10 seconds of simulation time, while the processors will be idle for 9 seconds. On the other hand, if fine grained time stepping is used and component a is advanced with 1 second (or smaller) time steps, then it is possible to complete the 10 seconds simulation time in 20 seconds wall clock time. In this case there is no idle time at all as shown by the right panel of Figure 3.

The conclusion is that an efficient algorithm should use fine grained time stepping to advance the components.

2.3 Robustness Considerations

A robust algorithm should be able to complete the simulation for an arbitrary processor layout and coupling schedule. This requires that the framework should never get into a dead-lock, when all the processors are waiting for other processors in a circular fashion. In this subsection various types of dead-lock situations will be shown. The examples will motivate the choices made in the design of the proposed robust algorithm.

2.3.1 Dead-lock due to inconsistent order of time stepping

One could design an algorithm where the order of time stepping the components varies from processor to processor. For example one could measure the wall clock time for each component, and advance the component which used up the least wall clock time so far. Let us examine if this approach can lead to problems. Assume that we have two components a and b which use the same subset of the processors ($P_{a,b} := P_a = P_b$) and they execute on more than one processors ($N_{P_{a,b}} > 1$). Since the measured wall clock time is not necessarily the same on all processors, it can happen that on processor $p \in P_{a,b}$ the time step of component a is started (S_a), while on processor $q \in P_{a,b}$ the time step of component b is started. Using the notation defined in (6), the algorithm does the following:

Step	p	q
1	S_a	S_b
2	idle	idle

where 'Step' monotonically increases with the wall clock time, and serves as an identifier to the sequence of algorithmic steps on the processors. Since operations S_a and S_b are blocking the execution, and neither S_a nor S_b are executed on *all* processors of $P_a = P_b$, the time steps cannot complete, and the algorithm is in a dead-lock.

The conclusion is that the order of the time stepping of any two components must be deterministic and identical on processors used by both components.

2.3.2 Dead-lock due to incorrect order of couplings

When there are several couplings scheduled for the same simulation time, it is important to execute them in an order which does not allow a dead-lock.

Assume, for example, that components a, b, and c are executing concurrently on the disjoint subsets P_a, P_b and P_c and they are to be coupled pair-wise at the same simulation time $t_{\{a,b\}} = t_{\{b,c\}} = t_{\{a,c\}}$. If the algorithm is not designed correctly, it can happen that processors in P_a start the coupling $S_{\{a,b\}}$ first, the processors in P_b execute $S_{\{b,c\}}$ first, and the processors in P_c start the couplings with $S_{\{a,c\}}$. Using the notation defined in (13) the incorrect parallel algorithm does the following:

Step	P_a	P_b	P_c
1	$S_{\{a,b\}}$	$S_{\{b,c\}}$	$S_{\{a,c\}}$
2	idle	idle	idle

The result is a circular dead-lock, since the start of the coupling blocks the execution, but there is no coupling which has been started on all the involved processors.

To avoid such dead-locks, the couplings must be ordered, and the couplings should be executed in that order on all processors. The ordering can be defined by a one-to-one correspondence (or bijection) between the elements of K and the integers between 1 and N_K:

$$f : K \leftrightarrow \{k \mid 1 \leq k \leq N_K, \quad k \in \mathbb{N}\} \tag{25}$$

where N_K is the number of couplings. The order of the couplings can be given as an input parameter for the framework.

For the above example we may assume that $f(\{a,b\}) < f(\{b,c\}) < f(\{a,c\})$. If the algorithm starts the couplings in the order defined by increasing values of f, the execution will progress the following way

Step	P_a	P_b	P_c
1	$S_{\{a,b\}}$	$S_{\{a,b\}}$	$S_{\{b,c\}}$
2	$F_{\{a,b\}}$	$F_{\{a,b\}}$	idle
3	$S_{\{a,c\}}$	$S_{\{b,c\}}$	idle
4	idle	$F_{\{b,c\}}$	$F_{\{b,c\}}$
5	idle	progress	$S_{\{a,c\}}$
6	$F_{\{a,c\}}$	progress	$F_{\{a,c\}}$
7	progress	progress	progress

where F means that the coupling is finished as defined in (14).

2.3.3 Dead-lock due to rushing ahead

We show that condition (17) for running a component (executing S_c) is not sufficient to avoid potential dead-locks. Let us assume that components a and b are running on disjoint processor subsets P_a and P_b, while component c is using the processors $P_c = P_a \cup P_b$. At the beginning let all components start from time $t_a = t_b = t_c = 0$. Component a, b and c take time steps $\Delta t_a = 1$, $\Delta t_b = 2$ and $\Delta t_c = 10$, respectively. Components a and b are scheduled for coupling at $t_{\{a,b\}} = 5$, and component c is not coupled to anything up to $t = 30$.

The algorithm will run component b for one time step, where it will start coupling $\{a, b\}$, while component a would need to do five time steps to reach the coupling time. According to the condition (17), however, processors on P_a will advance both a and c in turns. Since c is running on P_b as well which is blocked by b, it will not be able to complete the second step, and a dead-lock occurs. Using the notation introduced in (6)$-$(13), the following happens:

Step	P_a	P_b
1	S_c	S_c
2	F_c	F_c
3	$t_c := 10$	$t_c := 10$
4	S_a	S_b
5	F_a	F_b
6	$t_a := 1$	$t_b := 5$
7	S_c	$S_{\{a,b\}}$
8	idle	idle

Note that the processors on P_b cannot tell when component a will reach the coupling time $t_{\{a,b\}}$, because there is no communication with processors P_a prior to the coupling. Therefore the dead-lock has to be resolved by not allowing component c to 'rush ahead' even if it satisfies condition (17), i.e. it has not reached any coupling time which involves component c. The extra condition should be based on information available on P_a, which includes the component time t_a and the coupling time $t_{a,b}$.

3 Algorithm

Based on the considerations and lessons learned in sections 2.2 and 2.3, we construct a scheduling algorithm which uses a *Temporal and Predefined Ordering of Tasks* (TPOT). In the following subsections the TPOT algorithm is defined, an implementation is provided and then the correctness and robustness are proved.

3.1 Mathematical definition of TPOT

The basic idea of the algorithm is to execute all tasks (couplings and time-stepping) in the order of simulation time, and if the simulation times are equal, then in a predefined order.

In general the framework has to execute the following tasks: couplings between components and time stepping components. We can define the set of current tasks W (stands for 'work-to-do') as executing the next coupling for all pairs of components in K and doing the next time step for all components in C.

Since there is exactly one task for each element of K and C, and K and C are disjoint, the set W can be defined as

$$W := K \cup C \tag{26}$$

For each element $w \in W$ we can assign a unique integer index in the range $1 \ldots (N_K + N_C)$ by generalizing the coupling order function f defined in (25) to an order function for all tasks:

$$g : w \to \begin{cases} f(w) & \text{if } \; w \in K \\ w + N_K & \text{if } \; w \in C \end{cases} \tag{27}$$

Note that the coupling tasks are indexed with smaller numbers than the time stepping tasks. For each processor p the subset of couplings it is involved in is

$$K_p := \{\{c, d\} \mid (c \in C_p \text{ or } d \in C_p) \text{ and } \{c, d\} \in K\} \tag{28}$$

and then the subset of tasks that processor p is involved in is defined as

$$W_p := C_p \cup K_p \tag{29}$$

The TPOT algorithm can now be described in a single sentence: **each processor should execute its tasks in the order of the simulation times associated with the tasks, and if the simulation times coincide, then in the order defined by the order function g**. More precisely, for tasks $w, z \in W_p$

$$S_w \text{ should preceed } S_z \text{ if } \; (t_w < t_z \text{ or } \; (t_w = t_z \text{ and } g(w) < g(z))) \tag{30}$$

3.2 An implementation of TPOT

Here we describe a possible implementation of the TPOT algorithm both in form of text and in form of a pseudo-Fortran code.

Each processor $p \in P$ on which there is at least one component ($N_{C_p} > 0$) executes the following steps

(1) Set the minimum local time τ_p to the minimum of the simulation times t_w associated with the local tasks $w \in W_p$
(2) It τ_p has reached the final time $t_{\max}$ exit.

```
 (1) TIMELOOP: do
 (2)    $\tau_p = \min\{t_w \mid w \in W_p\}$
 (3)    if ( $\tau_p \geq t_{\max}$ ) exit TIMELOOP
 (4)    COUPLELOOP: do $k = 1,\ N_K$
 (5)       c = couple_order(k,1); d = couple_order(k,2)
 (6)       if ( ($c \notin C_p$ .and. $d \notin C_p$) ) cycle COUPLELOOP
 (7)       if ( $t_{\{c,d\}} == \tau_p$ ) then
 (8)          call couple_comp(c,d)
 (9)          $t_{\{c,d\}} = t_{\{c,d\}} + \Delta t_{\{c,d\}}$
(10)       endif
(11)    enddo COUPLELOOP
(12)    STEPLOOP: do $c = 1,\ N_C$
(13)       if( $c \notin C_p$ ) cycle STEPLOOP
(14)       if( $t_c == \tau_p$ ) then
(15)          call run_comp(c)
(16)          $t_c = t_c + \Delta t_c$
(17)       endif
(18)    enddo STEPLOOP
(19) enddo TIMELOOP
```

Fig. 4. Pseudo-Fortran implementation of the TPOT algorithm. The line numbers are shown for sake of easy references.

(3) Start and finish all couplings $\{c, d\} \in K_p$ with $t_{\{c,d\}} = \tau_p$ in the order determined by the coupling order function f. Increase the coupling time by $\Delta t_{\{c,d\}}$ when coupling is done.
(4) Start running all components $c \in C_p$ with $t_c = \tau_p$ in the order determined by the component indexes. Increase the component time by Δt_c when the time step is done.
(5) Go back to step 1.

Figure 4 shows a pseudo-Fortran implementation of the algorithm. The two-dimensional integer array `couple_order` provides the two component indexes in the order of the couplings. In essence `couple_order(k,:)` $= f^{-1}(k)$. The array should be initialized the same way on all processors. The subroutines `couple_comp` and `run_comp` implement the $S_{\{c,d\}}$, $F_{\{c,d\}}$ and S_c, F_d steps defined in (13)−(14) and (6)−(7), respectively.

3.3 Proof of correctness

This subsection shows that the TPOT algorithm satisfies the necessary conditions defined in (17), (19)−(20), and (23)−(24). Each statement will be accompanied with a reference to the corresponding line number in Figure 4.

PROOF. A coupling between components c and d is started (line 8) on processor p only if $\{c, d\} \in K$ (line 5) and ($c \in C_p$ or $d \in C_d$) (line 6). These conditions are the same as $\{c, d\} \in K_p$. A further condition is that $t_{\{c,d\}}$ equals τ_p (line 7), which implies conditions (19)−(20), since $t_c \geq \tau_p$ if $c \in C_p$ and $t_d \geq \tau_p$ if $d \in C_p$ (line 2). Thus no coupling is started before the local components have reached the coupling time.

The running of component c is started (line 15) on processor p only if $c \in C_p$ (line 13) and t_c equals τ_p (line 14). Since $t_{\{c,d\}} \geq \tau_p, \forall \{c, d\} \in K_p$ (line 2), this implies that

$$t_c \leq t_{\{c,d\}} \qquad \forall \{c, d\} \in K_p \tag{31}$$

Note that this is not the same (yet) as condition (17), which requires that $t_c < t_{\{c,d\}}$. Here we can exploit the feature of the algorithm, which orders the coupling tasks before the time stepping tasks. For all couplings $\{c, d\} \in K_p$ for which $t_{\{c,d\}}$ equals t_c, the $S_{\{c,d\}}$ must have been called (line 8) before S_c is called (line 15). Therefore when the time step gets started (line 15), the originally equal coupling times had already been increased (line 9) and condition (17) holds. Thus no extra time stepping is done after the coupling time has been reached.

The algorithm stops on processor p when $\tau_p \geq t_{\max}$ (line 3) which implies conditions (23)− (24), since $t_c \geq \tau_p, \forall c \in C_p$ and $t_{\{c,d\}} \geq \tau_p, \forall \{c, d\} \in C_p$, respectively (line 2). Consequently the algorithm does not finish without executing all couplings and time steps. □

3.4 Proof of robustness

In this subsection we prove that the TPOT algorithm cannot get into a dead-lock. We start with an intuitive and constructive proof which roughly follows the true execution of the algorithm, and finish with a more rigorous proof, which starts with the assumption that there is a dead-lock, and shows that such an assumption leads to contradicting.

PROOF 1 At any point in the execution, the component times and coupling times are consistent across all processors, and so is the task indexing function g defined in (27). Consequently the tasks are ordered the same way on all processors, although the processors only know about a subset W_p of the tasks. Let us select the task w which is the first in the global ordering (i.e. it has the smallest t_w, and among the tasks with the same simulation time it has the smallest index $g(w)$). For any processor $p \in P_w$, the minimum simulation time

Table 1
Simulation time steps (Δt) and CPU costs (ΔT) for the components of the SWMF

Component	ID	c	Δt_c	ΔT_c	$\Delta t_c/\Delta T_c$	$\max N_{P_c}$
Solar Corona	SC	1	0.397	11.48	0.035	22408
Inner Heliosphere	IH	2	60.000	81.81	0.733	2880
Solar Energetic Particles	SP	3	60.000	6.01	9.983	1
Global Magnetosphere	GM	4	4.000	125.54	0.032	2494
Inner Magnetosphere	IM	5	5.000	0.53	9.434	1
Radiation Belt	RB	6	300.000	17.79	16.863	1
Ionospheric Electrodynamics	IE	7	8.000	4.26	1.853	2
Upper Atmosphere	UA	8	10.000	38.16	0.262	32

must be $\tau_p = t_w$ since w is the task with the smallest time. Since the processors try executing the tasks in the same order, on all $p \in P_w$ the S_w will be called before starting any other (potentially blocking) tasks. Both for time stepping and coupling tasks the sufficient condition of completion is that all involved processors call S_w, therefore the task will be completed and the processors will progress to the next task. Since the number of tasks is finite, eventually all tasks will be completed and the algorithm will finish successfully. □

PROOF 2 An alternative proof starts with the assumption that the algorithm has reached a dead-lock. In that case all processors must have started some task w_p but never returned from S_{w_p}. We can now select task w as the first of the set of w_p tasks arranged by the task order. All processors involved in task w ($p \in P_w$) must have called S_w the last time, since S_w is called before any other of the unfinished tasks, and task w is not yet complete. According to the conditions (9) and (16), however, this means that S_w will complete successfully, which contradicts the assumption of the dead-lock. □

4 Application

To demonstrate the usefulness of the TPOT algorithm, we present some emulated timings for the SWMF involving 8 components with 9 pairwise couplings. The execution time is calculated by a Perl script which can emulate the parallel execution of the components, it takes the processor synchronizations into account, and adds up the CPU time spent on running the components and the idle times for each processor. The script uses timings from actual SWMF runs to estimate the execution time of the components and it uses Amdahl's law [9] to estimate the scaling behavior of the parallel components based on

Table 2
Coupling frequencies in the SWMF

Coupling	c	d	$\Delta t_{\{c,d\}}$
SC-IH	1	2	60.0
SC-SP	1	3	60.0
IH-SP	2	3	60.0
IH-GM	2	4	60.0
GM-IM	4	5	40.0
GM-RB	4	6	300.0
GM-IE	4	7	8.0
IM-IE	5	7	40.0
IE-UA	7	8	80.0

the timings of the components on different number of CPU-s. The time spent on the couplings is currently neglected. The script can give reasonable estimates for doing the simulations on different number of processors and with different processor layouts and coupling schedules. Using emulated timings instead of actual timings allows very fast evaluation of the various execution strategies, and the results are easier to interpret than the timings of actual SWMF runs which include coupling times, I/O operations and uncertainties due to the communication hardware. It has been verified in several runs that the emulated timings and the actual timings are reasonably close, and thus these idealized timing results provide a good basis of comparison.

Table 1 shows the typical time steps Δt and the associated CPU time costs (estimated for a single processor) ΔT on an SGI Altix supercomputer for each component. The single-processor performance (defined as simulation time divided by CPU time) varies between 0.032 and 17. This means that if we wish to get real time speed (wall clock time equal to simulation time), some components will have to run on at least 32 CPU-s, while other components cannot keep even a single CPU busy.

Not all the components of the SWMF are parallel. The last column in Table 1 shows the maximum number of processors that the component can use. In case of the truly parallel components the maximum number of processors is limited by the total number of blocks, which are the basic un-dividable units of the domain decomposition in those components. In practice the non-ideal scaling limits the maximum number of processors for the SC, IH and GM components to a few hundred, above which the performance (simulation time/wall clock time) starts to degrade.

The coupling frequencies are based on the time steps of the involved components as well as accuracy requirements. They vary between 8 seconds and five minutes as shown in Table 2. The simulation is done from $t_{\min} = 0$ to $t_{\max} = 600$ seconds simulation time. The total CPU time needed is

$$T_{\rm opt} = t_{\max} \sum_c \frac{\Delta T_c}{\Delta t_c} \approx 39768\,{\rm s} \tag{32}$$

This CPU time is a lower limit that can be achieved only when all components run on a single CPU and there is no idle time. We can define the optimal wall clock time on N_p processors as

$$S_{\rm opt} = \frac{T_{\rm opt}}{N_p} \tag{33}$$

and define the efficiency as

$$\varepsilon = \frac{S_{\rm opt}}{S} \tag{34}$$

where S is the actual execution wall clock time in seconds. If the number of processors exceed the number of components, the efficiency must drop below the maximum 100% because of the non-perfect scaling of the parallel components. In addition to that there may be idle times on some of the processors due to the imperfect load balancing and/or coupling schedules.

Let's compare various execution strategies for $N_p = 128$ processors. The optimal wall clock time is $S_{\rm opt} \approx 310$ seconds. In purely sequential execution mode, where all components use as many processors as they can and the processor layouts are overlapped, SWMF will finish the simulation in about $S = 870$ seconds, which corresponds to an efficiency $\varepsilon \approx 36\%$. If we modify the layout such that the components with limited number of CPU-s do not overlap, but the SC, IH, and GM components use all the CPU-s, the total time reduces to 794 seconds. A pure concurrent layout should be based on the relative performance of the parallel components. The serial components (SP, IM and RB) occupy 3 CPU-s. The IE component could use 2 CPU-s, but it is fast enough on 1 CPU. The remaining 124 CPU-s should be divided between IH, UA, SC and GM such that they run at approximately the same speed. Based on the performance numbers in Table 1 and the scaling behaviors, we assign 3 processors to IH, 6 processors to UA, 58 processors to SC and 57 processors to GM to achieve an optimal load balance. This fully concurrent layout results in a wall clock time of 410 seconds with an efficiency $\varepsilon \approx 76\%$. The efficiency is mostly limited by the scaling behavior of the GM and SC components. If the total CPU time T_{opt} would use the costs of GM and SC on about 50 processors (as opposed to 1), the efficiency would be $\varepsilon \approx 92\%$.

Table 3
Wall clock time S and efficiency ε on 128 processors

Layout and number of processors used by	UA	IH	SC	GM	S	ε
overlap	32	128	128	128	870	36%
SP,IM,RB,IE,UA disjoint, SC-IH-GM all	32	128	128	128	794	39%
disjoint	6	3	58	57	410	76%
disjoint except for SC-IH overlap	6	61	61	57	410	76%

In case the IH component does not fit onto 3 processors, it may be overlapped with other components. Since IH is coupled to SC and SP only, it is natural to overlap IH with SC, so the two components do not have to wait for each other. This results in a mixed layout with SC and IH overlapped on 61 processors, GM running on 57 processors, UA on 6 processors and SP, IM, RB and IE on 1 processor each. The resulting wall clock time is 410 seconds, which is the same as for the fully disjoint layout. In both cases the wall clock time is determined by the GM component running on 57 processors, but the SC and IH components are also utilizing their processors to almost 100%. The various layouts and the corresponding timings and efficiencies are summarized in Table 3.

Let's now compare the execution strategies for 32 processors. The optimal wall clock time is $S_{\rm opt} \approx 1240$ seconds. In the purely sequential execution mode the SWMF will finish in 1735 seconds. If the SP, IM, RB and IE components have non-overlapping layouts, while UA, IH, SC and GM use all 32 processors, the wall clock time improves only to 1704 seconds. The reason for the modest improvement is that the serial components cannot execute concurrently, because their time steps and costs are very different, and the couplings result in a lot of idle time. In a purely concurrent layout which is optimized for load balance, the SP, RB, IM, IE and IH components use 1 processor each, the UA uses 2 processors and the SC and GM components use 12 and 13 processors, respectively. The wall clock time is 1553 seconds, which is determined by the SC component, but GM is also running 98% of the time.

In practice, however, the IH component does not fit onto a single processor due to the memory requirements. If we overlap SC and IH on 13 processors, while GM still uses 13 processors, the overall wall clock time improves to 1523 seconds. The improvement is due to the better use of the processor which was occupied by IH in the concurrent layout (and was idle 50% of the time), and is shared by SC and IH (with almost no idle time) when the SC and IH components overlap. This change suggests that we can further improve the speed by eliminating idle time on the least used processors. Our final layout overlaps the SP, RB, IM and IE components on a single processor. The UA, IH and SC components are overlapped on 16 processors, finally the GM component uses the remaining 15 processors. With this layout the SWMF

Table 4
Wall clock time S and efficiency ε on 32 processors

Layout and number of processors used by	UA	IH	SC	GM	S	ε
overlap	32	32	32	32	1735	71%
SP,IM,RB,IE disjoint, UA-SC-IH-GM all	32	32	32	32	1704	73%
disjoint	2	1	12	13	1553	80%
disjoint except for SC-IH overlap	2	13	13	13	1523	81%
SP-IM-RB-IE overlap, SC-IH-UA overlap	16	16	16	15	1401	89%
same as above but large time steps	16	16	16	15	1691	73%

finishes the simulation in 1401 seconds which corresponds to an efficiency of $\varepsilon \approx 89\%$.

Finally we demonstrate that fine grained time stepping pays off in practice too. A simple way to mimic coarse grained time stepping is to increase the time steps Δt_c to be equal to the largest common denominator of the coupling frequencies $\Delta t_{\{c,d\}}$, and the computational cost ΔT_c is also increased proportionally. In effect the components will get from coupling to coupling in a single (or a few) time steps. For the purely disjoint or fully overlapping layouts this change makes no difference at all. For the mixed layouts, however there is a change in performance. In particular, let's take the best layout found for the fine grained time stepping on 32 processors: SP-RB-IM-IE overlapped on 1 processor, UA-IH-SC overlapped on 16 processors, and the GM component uses the remaining 15 processors. With the increased time steps the simulation takes 1691 seconds instead of the 1401 seconds obtained with the fine grained time stepping. The efficiency drops from 89% to 73%.

The various layouts, the timings and efficiencies are listed in Table 4.

5 Conclusions

This paper presented a general algorithm for scheduling the parallel execution and coupling of components in a software framework. The TPOT algorithm requires no extra communication between the processors, it allows arbitrary processor layouts and coupling schedules, and it cannot get into a dead-lock. These features allow to select optimal layout and coupling schedules for a given number of processors. The TPOT algorithm has been successfully used in the Space Weather Modeling Framework.

The SWMF is implemented as a single executable. For a multiple-executable framework the TPOT algorithm will work, but there may be alternatives. If each executable corresponds to a component, and multiple components can run on the same processor, then some of the potential dead-lock situations may be resolved by the multitasking environment. For example starting the execution of component a will not block the execution of component b on the same processor, so the dead-lock situations described in subsections 2.3.1 and 2.3.3 will be avoided. This means that a multiple-executable approach may allow more flexibility in the scheduling algorithm. Note, however, that even in the multiple-executable case it is possible to get into trouble: for example the situation described in subsection 2.3.2 with the circular couplings will lead to a dead-lock.

We have not attempted to solve the general optimization problem, which can be formulated in various manners. For example one can look for the optimal processor layout for a given coupling schedule and number of processors, or one can try to find the minimum number of processors that achieves a required execution speed. In case the cost of the execution of the components or the cost of the couplings change significantly during the run, it may be necessary to do dynamic load balancing. This dynamic load balancing could use the timings made during the run, in particular the idle times should be minimized. The cost of the load balancing itself should also be taken into account.

Even in the idealized case, when the parallel components scale perfectly and the couplings have zero cost, it is not trivial to find the optimal processor layout. When the non-ideal scaling and the cost of the couplings (which also depends on the number of processors involved) is taken into account, the optimization becomes even more difficult. It is likely that the optimal layout can only be determined by a numerical algorithm, and due to the very large number of combinatorial possibilities, the algorithm has to be able to search for the optimal layout in an intelligent manner.

References

[1] G. Tóth, I. V. Sokolov, T. I. Gombosi, D. R. Chesney, C. R. Clauer, D. L. De Zeeuw, K. C. Hansen, K. J. Kane, W. B. Manchester, R. C. Oehmke, K. G. Powell, A. J. Ridley, I. I. Roussev, Q. F. Stout, O. Volberg, R. A. Wolf, S. Sazykin, A. Chan, B. Yu, J. Kóta, Space Weather Modeling Framework: A New Tool for the Space Science Community, *J. Geophys. Res. (Space Physics)* in press

[2] G. Tóth, O. Volberg, A. J. Ridley, T. I. Gombosi, D. L. De Zeeuw, K. C. Hansen, D. R. Chesney, Q. F. Stout, K. G. Powell, K. J. Kane, R. C. Oehmke, A Physics-Based Software Framework For Sun-Earth Connection Modeling, in *Multiscale Coupling of Sun-Earth Processes*, Elsevier Publ. Co., Amsterdam, The Netherlands, A. T. Y. Lui, Y. Kamide, and G. Consolini, eds., (2005) 383.

[3] C. Hill, C. DeLuca, V. Balaji, M. Suarez, A. da Silva, and the ESMF Joint Specification Team, The Architecture of the Earth System Modeling Framework, *Computing in Science and Engineering*, **6** January/February (2004).

[4] Gurnis, M.; Aivazis, M.; Tromp, J.; Tan, E.; Thoutireddy, P.; Liu, Q.; Choi, E.; Dicaprio, C.; Chen, M.; Simons, M.; Quenette, S.; Appelbe, B.; Aagaard, B.; Williams, C.; Lavier, L.; Moresi, L.; Law, H., GeoFramework: A Modeling Framework for Solid Earth Geophysics, *American Geophysical Union, Fall Meeting* (2003) abstract #NG12D-06

[5] Luhmann, J. G., S. C. Solomon, J. A. Linker, J. G. Lyon, Z. Mikic, D. Odstrcil, W. Wang, M. Wiltberger, Coupled model simulation of a Sun-to-Earth space weather event, *J. Atm. S.-T. Phys.*, **66** (2004), 1243

[6] Buis, S.; Declat, D.; Gondet, E.; Massart, S.; Morel, T.; Thual, O., PALM : A dynamic parallel coupler for Data Assimilation, *EGS - AGU - EUG Joint Assembly, Nice, France* (2003) abstract #54

[7] Williams, Timothy J.; Crotinger, James A.; Cummings, Julian C.; Lin, Zhihong, GTC++: An Object-Oriented, Parallel, Gyrokinetic PIC Simulation, *American Physical Society, Division of Plasma Physics Meeting, New Orleans, LA* (1998) abstract #F3Q.12

[8] Leboeuf, Jean-Noel; Decyk, Viktor; Dimits, Andris; Shumaker, Dan, Gyrokinetic calculations of ITG turbulence in general toroidal geometry within the Summit Framework, *American Physical Society, 45th Annual Meeting of the Division of Plasma Physics, Albuquerque, New Mexico* (2003) abstract #LP1.064

[9] G. Amdahl, Validity of the Single Processor Approach to Achieving Large-Scale *Computing Capabilities, AFIPS Conference Proceedings*, **30** (1967) 483-485,