J. Parallel Distrib. Comput. 73 (2013) 207–219

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Parallel partitioning for distributed systems using sequential assignment

Simon Spacey *, Wayne Luk, Daniel Kuhn, Paul H.J. Kelly

Department of Computing, Imperial College London, London, UK

ARTICLE INFO

Article history:
Received 23 January 2012
Received in revised form
16 August 2012
Accepted 27 September 2012
Available online 5 October 2012

Keywords:
Parallel partitioning
Sequential assignment
Write-Only Architecture
Heterogeneous computing
High-performance computing

ABSTRACT

This paper introduces a method to combine the advantages of both task parallelism and fine-grained co-design specialisation to achieve faster execution times than either method alone on distributed heterogeneous architectures. The method uses a novel mixed integer linear programming formalisation to assign code sections from parallel tasks to share computational components with the optimal trade-off between acceleration from component specialism and serialisation delay. The paper provides results for software benchmarks partitioned using the method and formal implementations of previous alternatives to demonstrate both the practical tractability of the linear programming approach and the increase in program acceleration potential deliverable.

1. Introduction

There are two main branches in the current computational acceleration research domain: parallel partitioning and sequential assignment. In parallel partitioning multiple coarse-grained tasks are accelerated by executing them in parallel on parallel computational components [6,19,35,56,47,55,12,52] and in sequential assignment a single task is accelerated by dividing it into sections which are executed on components specialised for different types of code in a heterogeneous architecture [48,30,29,40,5,45,46,44]. The two branches have previously been considered distinct with authors concerned with problems characterised by multiple threads of largely independent tasks or single tasks with running internal data dependencies focusing on one branch or another.

In this paper we seek to join these two branches to obtain the benefits of both parallelism and code-sign specialism for general software partitioning. To do this however is not quite as simple as, say, just identifying the best sequential assignments then running them in parallel — because such an isolated task approach could violate hardware space constraints and would not try to optimise task interactions on serial components. If we hope to join the branches together in a way that optimises the use of shared heterogeneous hardware, it soon becomes evident that we need to deal with the problem of execution sequence uncertainty [46].

Sequence uncertainty is not an issue for parallel partitioning where full sequence information is available or for sequential assignment where components are not shared by parallel tasks and in this paper we present our solution to the sequence uncertainty problem for parallel partitioning using sequential assignment and in so doing contribute:

1. a formal model for partitioning coarse-grained parallel tasks to shared distributed hardware components using fine-grained sequential assignment.
2. insights to simplify the formal model for solution using standard solvers.
3. experimental results demonstrating the performance improvements possible over previous approaches for a suite of software benchmarks.

The paper begins with an overview of related work in Section 2. In Section 3 our formal model is presented along with simplifying insights. Section 4 then provides results that demonstrate the performance improvements achievable over previous methods together with quality bounds and solution timing information to demonstrate the practical utility of the formal model. The paper concludes with a high-level review and areas for future work in Section 5.

2. Related work

Previous software partitioning work can be classified as either parallel or sequential in focus based on the characteristics of Table 1. Parallel methods deliver their program acceleration by executing multiple code sections simultaneously on parallel execution units to exploit data independence and sequential partitioning methods execute one code section at a time on different hardware locations to exploit component specialism in heterogeneous architectures.

* Corresponding author.
E-mail addresses: simon.spacey@sase.biz (S. Spacey), wl@doc.ic.ac.uk (W. Luk), dkuhn@doc.ic.ac.uk (D. Kuhn), p.kelly@doc.ic.ac.uk (P.H.J. Kelly).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.09.019

Table 1
Characteristics of parallel and sequential partitioning methods compared with those of our approach. The Code Sections Executing row lists to the number of code sections (at the assignment granularity) that can be executing at any one time with the methods.

	Parallel	Sequential	This paper
Code sections executing	Multiple	1	Multiple
Graph type partitioned	DAG	CDFG	DAG and CDFG
Granularity	Coarse	Fine	Multi-level

A typical example of parallel partitioning work is [52] in which data dependent processes are assigned to distributed components for parallel execution. Parallel techniques require detailed dependency sequence information to take advantage of temporal assignment flexibility in parallel architectures. The problem that parallel techniques face is that the dependency sequence information they require must be provided at the same granularity as the assignment they hope to achieve and, at excessive granularity, loops and cycles can expand out fine-grained sequence traces to the point where analysis quickly becomes intractable [41,42,47]. To make parallel partitioning practically tractable, implementations tend to work at a coarse-grained Directed Acyclic Graph (DAG) level consolidating data dependent branches and cycles into a dependent sequence of assignable coarse-grained "tasks" [6] at the process [52], thread [19], function [12] or loop [35,55] level. Although it is often possible to achieve some benefits through careful task placement [52,35,24,49,26,55], the consolidation of fine-grained program parts into coarse-grained tasks means that traditional parallel assignment approaches cannot take full advantage of the fine-grained computational specialisation present in modern heterogeneous architectures [56].

A typical example of sequential partitioning work is [40] in which program basic blocks are assigned to heterogeneous components for sequential execution. Sequential techniques require detailed execution timing information for each component in an architecture to take advantage of component specialism [44]. The problem facing sequential techniques is that component specialism advantages are often only applicable to small code segments which is why sequential implementations typically work at the fine-grained Control/Data Flow Graph (CDFG) level. CDFGs compress out sequence information from traces rather than sacrificing graph granularity like DAGs and allow sequential assignment to be performed at the program kernel (tight loop) [48,30], the basic block [40] and even at the assembly code [5] level. While it is clear that a set of tasks could be executed one after another on heterogeneous hardware to accelerate multiple tasks with sequential assignment, the loss of parallel task execution advantages and the added overhead of hardware reconfiguration times [53] can outweigh the benefits of component specialism for some architectures as we will show in Section 4.

In this paper we present a hybrid partitioning technique called Multi-level Assignment Partitioning (MAP). Multi-level assignment techniques have proved beneficial in various fields of computing recently [28,19,18,38,8,32] and in this paper we introduce a formal multi-level approach to assign sets of coarse-grained tasks for parallel execution on shared distributed heterogeneous hardware components. Our multi-level approach uses coarse-grained DAG level task sets like parallel partitioning [6,19,35,52,55,12,56,47] as well as fine-grained CDFG level computation and communication information like sequential techniques [48,30,29,40,5,45,46,44] and in so doing is able to produce better execution times for static task assignments [1,24,7] than either of the previous methods alone as we will demonstrate in Section 4 after we detail our multi-level approach in Section 3 next.

3. Methodology

3.1. The Multi-level Assignment Problem

Our Multi-level Assignment Partitioning (MAP) approach is summarised in Fig. 1. Like traditional parallel partitioning we start with a set of coarse-grained tasks that can run in parallel. However unlike traditional parallel partitioning we use fine-grained timing information to assign subsections of the tasks to shared heterogeneous components to achieve benefits from both parallelism and component specialism. The problem of identifying parallel task sets has been dealt with elsewhere [20,6,26,13] and in this paper we focus on the last steps of the MAP process: the assignment of parallel coarse-grained tasks to shared components using fine-grained characteristics.

Before we begin detailing our methodology, let us note that we will be introducing variables at various points throughout this paper the most important of which we have summarised in Table 10 for your reference convenience. Let us now start by defining $\mathbb{X}$ as the set of feasible assignments of a group of static parallel tasks to locations that:

1. assign every code section for every task exactly once somewhere (see [46,21] for the alternative).
2. do not violate hardware size constraints.
3. assign code sections that are called by or call external code (for example system libraries) to appropriate locations (e.g. CPUs running an OS).

If we define $p \in \mathbb{P}^i$ to be an assignable code unit in a task i, $\mathbb{P}^i_e$ to be the set of assignable code sections that require external code in task i, δ^i_{pl} to be the size of the assignable unit p on location $l \in \mathbb{L}$ and Δ_l to be the total size capacity of location l, the three feasibility constraints above can be expressed in Linear Programming (LP) form as:

$$\sum_l x^i_{pl} = 1 \quad \forall i \in \mathbb{T}, p \in \mathbb{P}^i \tag{1}$$

$$\sum_{i \in \mathbb{T}, p \in \mathbb{P}^i} x^i_{pl} \delta^i_{pl} \leq \Delta_l \quad \forall l \in \mathbb{L} \tag{2}$$

$$\sum_{l_e \in \mathbb{L}^p_e} x^i_{pl_e} = 1 \quad \forall i \in \mathbb{T}, p \in \mathbb{P}^i_e \tag{3}$$

with $x^i_{pl} \in \{0, 1\}$ task code section level location assignment indicators and $\mathbb{L}^p_e$ the set of computational locations classified as having access to the external code section $p \in \mathbb{P}^i_e$.

Now, with reference to Fig. 1(d), it is clear that the total execution time for a particular parallel assignment x from the set of feasible assignments $\mathbb{X}$ can be defined as:

$$t_x = \max_{i \in \mathbb{T}} t^i_x \tag{4}$$

that is, the task group's execution time is the time t^i_x required for the slowest task i in the parallel DAG task set $\mathbb{T}$ to complete for the assignment x.

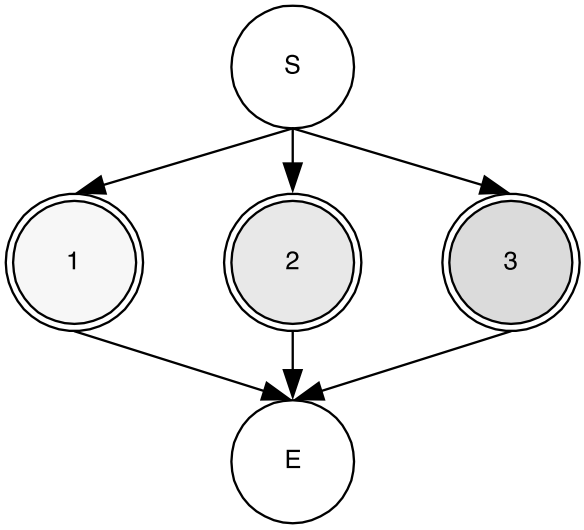
Our aim in multi-level partitioning is to find a multi-task assignment $x^* \in \mathbb{X}$ that produces the minimum execution time for the set of parallel tasks. To find x^* we need to solve the problem of Definition 1 below which takes advantage of the outer minimisation to convert Eq. (4) to the set of constraints (6).

Definition 1 (*Multi-level Assignment Problem*)**.**

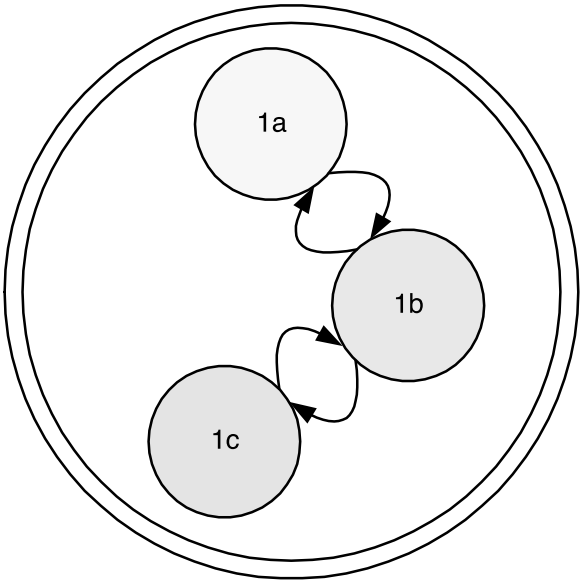
$$\min_{x \in \mathbb{X}} t_x \tag{5}$$

$$s.t.\ t_x \geq t^i_x \quad \forall i \in \mathbb{T} \tag{6}$$

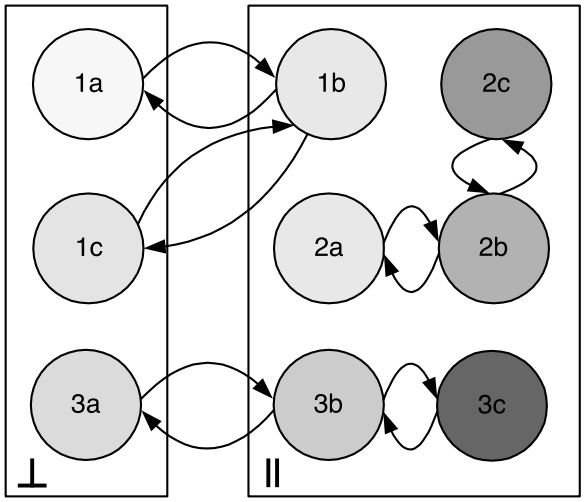
where $\mathbb{T}$ is the parallel task set, i a task identifier, t^i_x the time task i takes to execute given the multi-task assignment x from the set of feasible assignments $\mathbb{X}$.



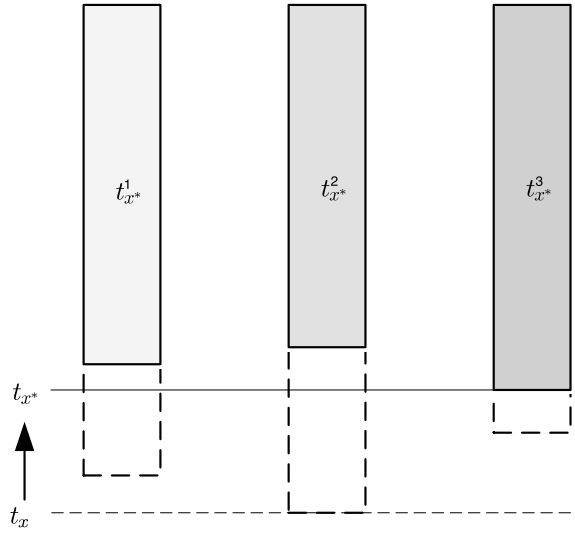
(a) Coarse-grained parallel task DAGs are identified.



(b) Fine-grained CDFG information is obtained for each coarse-grained task.



(c) Feasible fine-grained task assignments to shared sequential (⊥ e.g. CPU core) and parallel (‖ e.g. GPU or FPGA) locations $x \in \mathbb{X}$ are examined.



(d) The assignment execution times (including potential serialisation delays) t_x are used to identify the optimal assignment x^* for the combined task set.

Fig. 1. The Multi-level Assignment Partitioning (MAP) approach of combining coarse-grained parallel task partitioning with fine-grained sequential assignment. In the absence of sequence information, the fine-grained assignment time has to take into account all feasible serialisation conflicts across the task set as discussed in the text.

Concentrating for a moment on constraints (6) of Definition 1, we see from Fig. 1(c) that an individual task's execution time t^i_x for a particular assignment $x \in \mathbb{X}$ is at least the sum of the code section execution times $t^i_{pl|x}$ for each of the task's code sections p running on their assigned execution location l. For example, for task 1 of the assignment x shown in Fig. 1(c) we have:

$$t^1_x \geq t^1_{a\perp} + t^1_{b\|} + t^1_{c\perp}$$

which can be generalised to:

$$t^i_x \geq \sum_{pl} x^i_{pl} t^i_{pl|x} \tag{7}$$

where $t^i_{pl|x}$ is the time for code section p from task i to execute at location l including appropriately proportioned communication costs (we expand this statement in Section 3.3) between the task's code sections given the assignment x.

The inequality of Eq. (7) reflects the fact that serialisation conflicts on sequential execution components such as CPU cores can increase a task's elapsed execution time. Re-writing Eq. (7) to include an explicit slack variable for task level serialisation delays $t^i_\otimes \geq 0$ we have:

$$t^i_x = \sum_{pl} x^i_{pl} t^i_{pl|x} + t^i_\otimes. \tag{8}$$

Referring again to the three-task assignment example of Fig. 1(c) we see that task 2 has no serialisation delays because it does not spend any time on a shared serial component or serial communications channel with the assignment. Tasks 1 and 3 on the other hand could experience serialisation delays. Any serialisation delays for tasks 1 and 3 will depend on the exact execution sequence which is compressed out of the CDFG for practical complexity reasons as discussed in Section 2 (note Fig. 1(c) does

not define whether task 3 has the sequence $a \rightarrow b \rightarrow c$ or $a \rightarrow b \rightarrow a$ at points in its execution path).

As sequence information is compressed out of CDFGs, the best we could hope for when estimating $t^i_\otimes$ for an assignment is a robust solution based on feasible sequence paths. However obtaining such a feasible sequence aware robust solution is complex in both formulation and computation time as the authors have already shown in [46]. Consequently, in this work we use a path unaware robust bounding approach for $t^i_\otimes$ which we will show produces efficient multi-level assignments while remaining practically tractable with current solvers for a wide range of problems in Section 4.

The ultimate *lower bound* on a task's possible serialisation delays would clearly be $\underline{t}^{\,i}_\otimes = 0$ which assumes there are no clashes on sequential components for the task. The corresponding *upper bound*, and the robust form we use in this paper, would assume that the task is delayed by all other task executions on shared sequential components in a similar manner to [1], which can be defined algebraically as:

$$\bar{t}^{\,i}_\otimes = \sum_{j \neq i} \sum_{r \in \mathbb{P}^j} \sum_{l \in \mathbb{L}_\perp} \alpha^i_{l|x} x^j_{rl} t^j_{rl|x} \tag{9}$$

where i and j are task identifiers with $j \neq i$, r is an assignable code section in the task j, l is a location in the set of sequential execution locations $\mathbb{L}_\perp$ and $t^j_{rl|x}$ is the time for code section r of task j to execute on the sequential component l (with appropriately proportioned communication times). $\alpha^i_{l|x}$ is a new indicator variable that is 1 if any code unit of i is assigned to location l and 0 otherwise. Taking advantage of the minimising objective of Definition 1, $\alpha^i_{l|x} \in \{0, 1\}$ can be defined in LP form with:

$$\alpha^i_{l|x} \geq x^i_{pl} \quad \forall p \in \mathbb{P}^i \tag{10}$$

which can only be 0 if no part of i is assigned to location l (i.e. $\exists x^i_{pl} = 1 \Longrightarrow \alpha^i_{l|x} = 1$).

With our $t^i_\otimes$ upper bound now defined in terms of isolated task execution times in Eq. (9) we only need to detail the isolated fine-grained task execution timing contributions $t^i_{pl|x}$ and the set of sequential locations $\mathbb{L}_\perp$ in order to quantify Eq. (8) and use Definition 1 to identify optimal robust parallel partitions with our fine-grained intra-task assignment approach.

3.2. Execution timing contributions

From the preceding discussion we know that MAP selects the shared hardware assignment with the minimum overall execution time taking into account possible serialisation delays. The MAP approach can be used with existing parallel task identification schemes [20,6,26,13] and sequential assignment methods [48,30,29,40,5,45,46,44] provided the fine-grained isolated intra-task execution times $t^i_{pl|x}$ are available.

Approaches to obtain $t^i_{pl|x}$ values, sometimes termed Expected Times to Compute (ETC) [2,39], include direct measurement [11] with a specific sequential assignment implementation for an architecture and general models [42] and in the remainder of this work we will expand $t^i_{pl|x}$ using the Write-Only Architecture (WOA) model [45]. The WOA provides the following general formula to quantify ETC figures which includes appropriately proportioned communication costs:

$$t^i_{pl|x} = \mu^i_{pl} + \sum_{qm} x^i_{qm} c^i_{pqlm} \tag{11}$$

where μ^i_{pl} is the total computation time of code section p on location l, x^i_{qm} is 1 if code section q from the task i is instantiated at location m and 0 otherwise and c^i_{pqlm} is the total communication time for activations sent from components p on l to q on m. Eq. (11) can be used to model a range of sequential methods through appropriate selection (or measurement) of μ^i_{pl} and c^i_{pqlm} as discussed in [45] and implies that the time summations $\sum_{pl} x^i_{pl} t^i_{pl|x}$ of Eq. (8) and (9) will generally need to include a quadratic communications cost component to quantify the $|x$ dependency in $t^i_{pl|x}$ as we will see when we expand the MAP formulation next.

3.3. Expanding MAP with timing paradigm and architecture specifics

In this section we expand the MAP problem of Definition 1 using the general WOA execution timing contribution equation from Section 3.2 and sequential hardware classifications.

Referring to the original timing equations of Section 3.1, we see that our initial analysis assumed execution times included "appropriately proportioned communication costs" together with core computation times. The assumption of associated communication costs simplified our initial analysis but it left us with an x dependency on our fine-grained assignment timing function $t^i_{pl|x}$. However, the WOA execution paradigm [45] introduced in Section 3.2 provides us with a means to expand the communication costs in $t^i_{pl|x}$ in terms of the assignment variables x which we can now use to detail $t^i_{pl|x}$ in the execution timing Eq. (8) associated with Definition 1 as shown below:

$$t^i_x = \sum_{pl} x^i_{pl} \mu^i_{pl} + \sum_{pqlm} x^i_{pl} x^i_{qm} c^i_{pqlm} + \bar{t}^{\,i}_\otimes \tag{12}$$

where x^i_{pl} is 1 if code section p of task i is instantiated at location l and 0 otherwise and μ^i_{pl} and c^i_{pqlm} are the linear computation and quadratic communication cost constants which can be obtained through various means [42,45,11] for a specific problem instance. We can use a similar approach to expand the $t^j_{rl|x}$ in the $\bar{t}^{\,i}_\otimes$ definition of Eq. (9) to obtain:

$$\bar{t}^{\,i}_\otimes = \sum_{\substack{j \neq i, r, \\ l \in \mathbb{L}_\perp}} \alpha^i_{l|x} x^j_{rl} \mu^j_{rl} + \sum_{\substack{j \neq i, r, s, \\ (l,m) \in \mathbb{M}_\perp}} \beta^i_{lm|x} x^j_{rl} x^j_{sm} c^j_{rslm} \tag{13}$$

with r and s communicating code sections in task $j \neq i$ and lm shared communication channels from the (newly required) set of shared channels $\mathbb{M}_\perp \subseteq \mathbb{L} \times \mathbb{L}$ on which link contentions could arise [22]. $\beta^i_{lm|x}$ is a new communications channel usage indicator similar in concept to $\alpha^i_{l|x}$ which is 1 if any code section in task i communicates over the shared channel lm in an assignment and 0 otherwise. Taking advantage of the minimisation objective of Definition 1 once again, $\beta^i_{lm|x} \in \{0, 1\}$ can be expressed as:

$$\beta^i_{lm|x} \geq x^i_{pl} x^i_{qm} \quad \forall p, q \in \mathbb{P}^i : \chi^i_{pq} + \eta^i_{pq} > 0 \tag{14}$$

where χ^i_{pq} and η^i_{pq} are the control and data flows between code sections p and q of task i which can be statically determined using characterisation frameworks such as 3S [41] as detailed in [42]. Note that $\beta^i_{lm|x}$ will be dragged to 0 outside of the inequality set (14) by the minimising objective (all our time contribution constants are positive) and so we do not need to specify the implied constraints:

$$\beta^i_{lm|x} = 0 \quad \forall p, q \in \mathbb{P}^i : \chi^i_{pq} + \eta^i_{pq} = 0. \tag{15}$$

The other variables in Eq. (13) are as previously described for Eq. (9) and detailed in Table 10.

The final piece of information required to make Definition 1 concrete is to define the $\mathbb{L}_\perp$ and $\mathbb{M}_\perp$ used in the expanded $\bar{t}^{\,i}_\otimes$ of Eq. (13). This involves the classification of hardware components and communication channels into sequential or parallel sets which

is relatively easy to do as a one-off manual process for most architectures.

For example, the Imperial Axel architecture [50] consists of CPU, GPU and FPGA computational components interconnected by PCIe and Gigabit Ethernet communication channels. The CPU cores can be classified as sequential as they can be expected to run only one task at a time while the GPUs and FPGAs are parallel by nature [10,53] and all the inter-component communication channel types are shared/sequential [16,51]. It is worth noting that we need to also classify intra-component communications and for the Axel architecture, all variants of the internal CPU↔CPU communications can be considered sequential while the GPU↔GPU and FPGA↔FPGA internal communications can take place for each task over a dedicated task internal bus simultaneously and so can be considered parallel. For other hardware architectures, thread limits can be modelled as multiple sequential cores or location-wise parallel task limit constraints, shared links and component group interactions can be modelled using appropriate mapping functions (see the Φ clash indicator in Section 4.3 for an example) and Δ_l dimensions can be added along with other minor modifications to the basic feasibility constraints of Eqs. (1)–(3) if required.

All that remains to complete this section is to collate the equations and definitions presented to generate the expanded formal model for MAP given in Definition 2 below. Note that Definition 2 has quadratic and cubic terms combining x^i_{pl}, $\alpha^i_{l|x}$ and $\beta^i_{lm|x}$. However as these variables are binary, their products can be linearised as shown in Section 3.4 to solve Expanded Map Problem instances directly with standard linear solvers if required.

Definition 2 (*Expanded MAP Problem*)**.**

$$\min_{x\in\mathbb{X}} \; t_x \tag{16}$$

$$s.t.\;\; t_x \geq t^i_x \quad \forall i \in \mathbb{T} \tag{17}$$

$$t^i_x = \sum_{pl} x^i_{pl}\mu_{pl} + \sum_{pqlm} x^i_{pl}x^i_{qm}c^i_{pqlm} + \bar{t}^{\,i}_{\otimes} \tag{18}$$

$$\bar{t}^{\,i}_{\otimes} = \sum_{\substack{j\neq i,r,\\ l\in\mathbb{L}_\perp}} \alpha^i_{l|x}x^j_{rl}\mu^j_{rl} + \sum_{\substack{j\neq i,r,s,\\ (l,m)\in\mathbb{M}_\perp}} \beta^i_{lm|x}x^j_{rl}x^j_{sm}c^j_{rslm} \tag{19}$$

$$\alpha^i_{l|x} \geq x^i_{pl} \quad \forall p \in \mathbb{P}^i \tag{20}$$

$$\beta^i_{lm|x} \geq x^i_{pl}x^i_{qm} \quad \forall p,q \in \mathbb{P}^i : \chi^i_{pq} + \eta^i_{pq} > 0 \tag{21}$$

with $x \in \mathbb{X}$ an assignment of fine-grained code sections to locations for all coarse-grained tasks $i \in \mathbb{T}$ satisfying the feasibility constraints of Section 3.1 with variables as detailed in Table 10.

3.4. Simplifying the formulation for practical software

In this section we draw on several insights from practical software partitioning problems to reformulate MAP. The MAP problem of Definition 2 is easily recognised as a strongly NP-hard problem through its quadratic communication costs c^i_{pqlm} and binary assignment variables x^i_{pl} [42,15,36] and the reformulation presented in this section for practical problems does not alter the theoretical complexity. However, the reformulation does reduce the number of binaries in the problem considerably and allows problem instances to be solved to optimality using CPLEX [9] in just a few seconds as we will see in Section 4.3 demonstrating the practical utility of the MILP formulation and that it is not always necessary to resort to heuristics for problem instances in the software partitioning domain [52,25,26,29,17,44,40,7].

Let us start by defining what we mean by a practical software partitioning problem in this work.

Definition 3 (*Practical Software Problems*)**.** *A practical software partitioning problem is an assignment of software sections to hardware with the following characteristics:*

1. *all computations and cross-component communications require some time to complete.*
2. *intra-component communication times are negligible (i.e. can be considered 0) in comparison to other costs.*
3. *sequential communication channels connect sequential components.*

With the characteristics of Definition 3 we have:

$$\mu^i_{pl} > 0 \quad \forall i,p,l$$

$$c^i_{pqlm} \begin{cases} > 0 & \forall i,p,q \; \forall l \neq m \\ = 0 & \forall i,p,q \; \forall l = m \end{cases}$$

$$\mathbb{M}_\perp \supseteq \{(l,m) : l \in \mathbb{L}_\perp \vee m \in \mathbb{L}_\perp\}.$$

Now let us start our practical reformulation exercise by defining two convenience variables. Firstly let us define $t^i_{lm|x}$ to be the time task i would spend on lm in isolation (i.e. without serialisation delays) for an assignment which is given by:

$$t^i_{lm|x} = \sum_p x^i_{pl}x^i_{pm}\mu_{pl} + \sum_{pq} x^i_{pl}x^i_{qm}c^j_{pqlm} \tag{22}$$

noting that, for singly instantiated code [46], $x^i_{pl}x^i_{pm} = 0 \;\; \forall l \neq m$ and so:

$$t^i_x = \sum_{lm} t^i_{lm|x} + \bar{t}^{\,i}_{\otimes}. \tag{23}$$

Next let us define a new auxiliary binary $\gamma^i_{lm|x} \in \{0,1\}$ as:

$$\gamma^i_{lm|x} \geq \frac{t^i_{lm|x}}{M^i_{lm}} \tag{24}$$

with M^i_{lm} a constant quantified later in this section to ensure $\frac{t^i_{lm|x}}{M^i_{lm}} \leq 1 \; \forall lm$ and note that for practical software partitioning problems with a minimising objective:

$$\alpha^i_{l|x} = \gamma^i_{ll|x}$$

$$\beta^i_{lm|x} = \begin{cases} \gamma^i_{lm|x} & \forall l \neq m \\ 0 & \text{otherwise,} \end{cases}$$

in accordance with the characteristics of Definition 3.

Looking again at Definition 2 we see that we can use $\gamma^i_{lm|x}$ to reformulate Eq. (19) for practical software partitioning problems to:

$$\bar{t}^{\,i}_{\otimes} = \sum_{\substack{j\neq i,\\ (l,m)\in\mathbb{M}_\perp}} \gamma^i_{lm|x}t^j_{lm|x} \tag{25}$$

using characteristic 3 of Definition 3. Eq. (25) is a sum of binaries $\gamma^i_{lm|x}$ multiplied by dependent variables $t^j_{lm|x}$ (not constants) and we cannot solve this problem form with standard linear solvers. However there is a simple reformulation to obtain:

$$\bar{t}^{\,i}_{\otimes} = \sum_{(l,m)\in\mathbb{M}_\perp} \bar{t}^{\,i}_{lm\otimes} \tag{26}$$

$$\bar{t}^{\,i}_{lm\otimes} \geq \sum_{j\neq i} t^j_{lm|x} - N^i_{lm}(1-\gamma^i_{lm|x}) \tag{27}$$

which is solvable with $\bar{t}^{\,i}_{lm\otimes} \geq 0$ and N^i_{lm} a task and location dependent big-M large enough to ensure $\bar{t}^{\,i}_{lm\otimes} = 0$ whenever $\gamma^i_{lm|x} = 0$.

We can now substitute the above equations into Definition 2 to obtain the practical MAP problem form of Definition 4.

Definition 4 (*Practical MAP Problem*).

$$\min_{x\in\mathbb{X}} t_x \tag{28}$$

$$s.t.\ t_x \geq t_x^i \quad \forall i \in \mathbb{T} \tag{29}$$

$$t_x^i = \sum_{lm} t_{lm|x}^i + \bar{t}_{\otimes}^{\,i} \tag{30}$$

$$t_{lm|x}^i = \sum_p x_{pl}^i x_{pm}^i \mu_{pl} + \sum_{pq} x_{pl}^i x_{qm}^i c_{pqlm}^i \tag{31}$$

$$\bar{t}_{\otimes}^{\,i} = \sum_{(l,m)\in\mathbb{M}_\perp} \bar{t}_{lm\otimes}^{\,i} \tag{32}$$

$$\bar{t}_{lm\otimes}^{\,i} \geq \sum_{j\neq i} t_{lm|x}^j - N_{lm}^i(1-\gamma_{lm|x}^i) \tag{33}$$

$$\gamma_{lm|x}^i \geq \frac{t_{lm|x}^i}{M_{lm}^i} \tag{34}$$

with the feasibility constraints of Section 3.1 and big-Ms as defined below.

Definition 4 of MAP is a Mixed Integer Quadratically Constrained Program (MIQCP) with $|\mathbb{T}||\mathbb{L}_\perp|$ less auxiliary variables than Definition 2. However, the real benefit of the reformulation only becomes apparent when we transform the problem for solution with standard linear solvers.

To solve problem instances of quadratic and higher-order formal models using standard linear solvers we need to linearise variable products. This can be performed by replacing each unordered combination of unique binary products $x_1x_2\ldots,x_K$ in the problem instance with a new linearisation variable $y_{\{1,2,\ldots,K\}} \in [0, 1]$ using the method presented in [43,33] generalised to:

$$y_{\{1,2,\ldots,K\}} \geq \sum_{k=1}^{K} x_k - K + 1. \tag{35}$$

For the MIQCP of Definition 4 we only need to linearise the unique indicator combinations in the quadratics of Eq. (31). For Definition 2 on the other hand, we need to linearise quadratic combinations of x_{pl}^i, $\alpha_{l|x}^i$ and $\beta_{lm|x}^i$ in Eqs. (18), (19) and (21) together with cubic x_{pl}^i and $\beta_{lm|x}^i$ combinations in Eq. (19). The result is that Definition 4 has the order of $|\mathbb{T}|^2|\mathbb{P}|^2|\mathbb{L}_\perp|^2$ less linearisation variables than Definition 2 (where $|\mathbb{P}|$ is the number of code sections in a "representative" task such that $|\mathbb{T}||\mathbb{P}| = \sum_{i\in\mathbb{T}}|\mathbb{P}^i|$).

The benefit of the reduction in linearisation variables in Definition 4 comes at the expense of the introduction of sets of big-M constants M_{lm}^i and N_{lm}^i. The big-Ms extend the range of values in problem instances, particularly as M_{lm}^i is used as a divisor and N_{lm}^i a multiple, and care must be taken to avoid solution accuracy issues as discussed in [43]. Solution accuracy issues can be avoided for practical problem instances by setting the big-Ms to tight bounds using, for example, Definition 4 with the objective modified as illustrated below[1]:

$$M_{lm}^i \geq \left\{\max_{x\in X} t_{lm|x}^i\right\}$$

$$N_{lm}^i \geq \left\{\max_{x\in X} \sum_{j\neq i} t_{lm|x}^j\right\}$$

or by using numerical emphasis or implementing the constraints of (33) and (34) with indicators where available in the solver [9].

[1] Actually, the big-M bounds can often be improved further than those shown above by setting all M_{lm}^i and N_{lm}^i values to any known upper-bound of min t_x with the domain of $\gamma_{lm|x}^i$ changed from $\mathbb{B}$ to $\mathbb{Z}_+$.

Table 2
Characteristics of the six software tasks partitioned in this work. The figures are summaries of 3S version 2.10 [41,42] measurements made at the basic block level of granularity for code compiled to x86 CPU instructions with `gcc -O3` and running with representative program inputs. The symbols correspond to Table 10 with $|\mathbb{P}^i|$ the number of basic blocks in the task, $\sum\delta_{pf}^i$ the total size of the tasks on an FPGA component of Fig. 2 in x86 Instruction Equivalents, $\sum\mu_{pc}^i$ the total execution cycles (in ms) for the task on a single CPU in Fig. 2, $\sum\chi_{pq}^i$ the sum of the inter-block control flows and $\sum\eta_{pq}^i$ the sum of the non-cached inter-block data flows (in bytes) for each task.

Task	$\lvert\mathbb{P}^i\rvert$	$\sum\delta_{pf}^i$	$\sum\mu_{pc}^i$	$\sum\chi_{pq}^i$	$\sum\eta_{pq}^i$
`dijkstra`	117	295	292	9 864 826	26 750 704
`fft`	130	438	200	3 670 310	11 470 084
`ispell`	1033	3242	188	5 856 315	35 981 124
`jpeg`	1776	7657	134	4 406 038	12 414 004
`sha`	68	816	203	6 609 432	82 246 556
`susan-e`	249	2491	164	4 719 519	28 121 940

4. Results

4.1. Experimental configuration

This section compares the execution performance of parallel tasks partitioned for multi-component architectures using MAP and alternative approaches. The parallel tasks were selected to cover the six MiBench benchmark categories [14] and intra-task software characteristics were obtained using 3S version 2.10 [41,42] at the fine-grained basic block level as summarised in Table 2.

The tasks will be partitioned for execution on the architecture illustrated in Fig. 2 which is based on the Imperial Axel [50] heterogeneous computational cluster. We will be assigning to the CPU and FPGA components of the Axel nodes which will be sufficient to demonstrate the benefits of MAP over alternative approaches for specific design-points as in previous work [19,35,52,24–26,47,48,30,40,5,31] and we will discuss the general significance of our specific experimental results in Section 4.3.

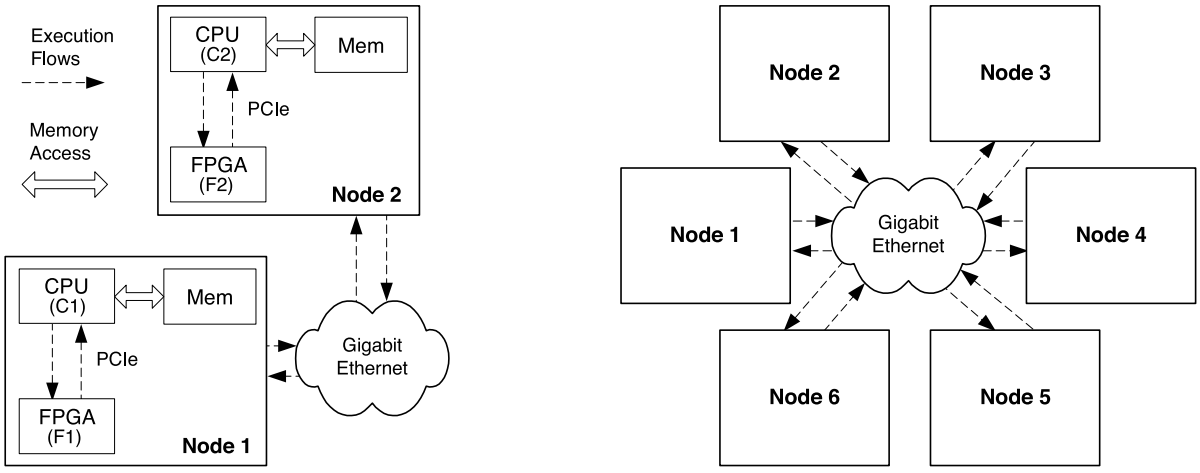
Task assignments for seven configurations of the detailed two-node cluster illustrated in Fig. 2(a) will be examined along with assignments for the high-level cluster of Fig. 2(b) with three, four, five and six full nodes activated. The seven configurations of Fig. 2(a) will be labelled depending on which components are active. The C1 configuration has only the first Axel node's CPU active in Fig. 2(a), configurations C1F1 and C1F2 add the first and second node's FPGA to represent the tightly and loosely coupled architectures of [45] respectively, configuration C1F1F2 has only the first CPU active and both FPGAs, C1C2 has both CPUs active but no FPGAs, C1F1C2 adds the first FPGA and configuration C1F1C2F2 represents the full four computational location, two-node Axel architecture illustrated in Fig. 2(a). Note that the seven detailed configurations represent the complete valid configuration set for the two-node architecture because of symmetry and because these software tasks all have external library access that requires at least one CPU be present in a feasible configuration. The $\mathbb{M}_\perp$ describing the sequential parts of the architecture is constructed with:

$$\mathbb{M}_\perp = \{(l,m) : l \in \mathbb{L}_{\text{CPU}} \vee l \neq m\} \tag{36}$$

where $\mathbb{L}_{\text{CPU}}$ is the set of active CPU locations i.e. CPU components and all communications to and from different components are sequential.

We will be comparing assignment results for the MAP approach presented in Definition 4 against the following alternatives:

1. homogeneous CPU-only execution.
2. parallel coarse-grained heterogeneous partitioning.
3. sequential fine-grained heterogeneous assignment.



(a) Details of the two-node cluster.

(b) Full six-node heterogeneous cluster.

Fig. 2. The heterogeneous architecture considered in this paper. The CPU components are 2.3 GHz AMD Phenom CPU cores [3] and the FPGAs are Xilinx LX330T reconfigurable fabrics [53] operating at an effective frequency of 266.67 MHz with a capacity of 2560 x86 integer Instruction Equivalents (IEs) [45]. Communications between components in the same Axel node take place over local PCIe WOA buses with latencies of 89 ns [16] and we will assume communications between components in different nodes take place over virtual dedicated Gigabit Ethernet WOA/UDP connections with latencies of 48 μs [51] in this paper. See [45,50] for more details on the architecture.

To ensure our comparisons are fair, we will be solving optimal versions of each approach rather than using the (potentially suboptimal) heuristic assignment methods of specific previous parallel coarse-grained [52,24–26] and sequential fine-grained [40,48,30,5,34] implementations using the same sequential execution timing figures and constants for each approach which we obtain from the WOA execution paradigm [45]. We will label our three formal alternative models $\mathcal{HOM}$, $\mathcal{PAR}$ and $\mathcal{SEQ}$ and we will refer to the formal MAP model of Definition 4 as $\mathcal{MAP}$ for convenience.

For our formal homogeneous model $\mathcal{HOM}$ we simply need to restrict all assignments to CPU locations which we can do by adding the following constraint set to $\mathcal{MAP}$:

$$x^i_{pl} = 0 \quad \forall l \notin \mathbb{L}_{\text{CPU}}.$$

Note that as all inter-location communications take time in our practical problem, there can be no benefit in splitting a single task over symmetric sequential CPUs and so the above constraint will produce task level assignments.

To represent the ideal general parallel task partitioning implementation, we will assume coarse-grained assignment on any feasible location with necessary exceptions for operating system function calls which we will say have to be executed on the same CPU for a task. We call our model of this parallel implementation $\mathcal{PAR}$ which is constructed by adding the following constraints to $\mathcal{MAP}$:

$$x^i_{pl} = \begin{cases} x^i_l & \text{if } p \notin \mathbb{P}^i_e \\ x^i_{al} & \text{otherwise.} \end{cases} \quad \forall i \in \mathbb{T}, p \in \mathbb{P}^i, l \in \mathbb{L}$$

$$\sum_{l \in \mathbb{L}} x^i_l = 1 \quad \forall i \in \mathbb{T}$$

with $x^i_l \in \{0, 1\}$ a new coarse-grained task based assignment location indicator and a the start node of a task which will be assigned to a CPU capable of accessing all the external code required by the task i in this work (i.e. $a \in \mathbb{P}^i_e$ and $\mathbb{L}^a_e = \mathbb{L}^p_e = \mathbb{L}_{\text{CPU}}\ \forall p \in \mathbb{P}^i_e$ here).

To represent the ideal general sequential implementation, we assume tasks can execute in parallel as long as no other task is running on any of their assigned components in the architecture, that components are configured before each task executes and that configuration times are based on the assigned sizes [54] rather than total device capacity [17] (i.e. partial reconfiguration). We call our formal model of this implementation $\mathcal{SEQ}$ which we construct by redefining terms in $\mathcal{MAP}$.

First we redefine the isolated task execution time of Eq. (31) to include configuration overheads with:

$$t^i_{lm|x} = \sum_p x^i_{pl} x^i_{pm} \mu_{pl} + \sum_{pq} x^i_{pl} x^i_{qm} c^j_{pqlm} + R^i_{lm|x}$$

where $R^i_{lm|x}$ is the time to configure lm for task i to execute under the assignment x which will be considered negligible for the communication channels ($l \neq m$) and the CPU components ($l = m \wedge l \in \mathbb{L}_{\text{CPU}}$) and a linear function of assigned component size for the reconfigurable components of Fig. 2 obtained from [54] for the results of this work. Next we define a task level interaction variable $\Gamma^{ij}_x \in [0, 1]$:

$$\Gamma^{ij}_x \geq \gamma^i_{lm|x} \gamma^j_{lm|x} \quad \forall (l, m) \in \mathbb{L} \times \mathbb{L}$$

which will be 1 if tasks i and j share any of the same components or channels (i.e. $\exists (l, m) \in \mathbb{L} \times \mathbb{L} : \gamma^i_{lm|x} = 1 \wedge \gamma^j_{lm|x} \Longrightarrow \Gamma^{ij}_x = 1$) and can be linearised using Eq. (35). We then redefine Eq. (32) of $\mathcal{MAP}$ to use task level serialisation overheads $\bar{t}^{\,ij}_{\otimes} \geq 0$ which we construct with Γ^{ij}_x and new big-Ms O^{ij}:

$$\bar{t}^{\,i}_{\otimes} = \sum_{j \neq i} \bar{t}^{\,ij}_{\otimes}$$

$$\bar{t}^{\,ij}_{\otimes} \geq \sum_{lm} t^j_{lm|x} - O^{ij}(1 - \Gamma^{ij}_x).$$

Note that $\bar{t}^{\,ij}_{\otimes}$ will be the entire isolated execution time of task j (not just the parts on $\mathbb{M}_\perp$) if task j shares any channel or component with task i (i.e. when $\Gamma^{ij}_x = 1$) which models the dedicated hardware access characteristic of our ideal sequential implementation.

We complete the sequential model by replacing the temporally shared size constraints $\sum_{i,p} x^i_{pl} \delta^i_{pl} \leq \Delta_l$ of Eq. (2) included in $\mathcal{MAP}$ with:

$$\sum_{p \in \mathbb{P}^i} x^i_{pl} \delta^i_{pl} \leq \Delta_l \quad \forall i \in \mathbb{T}, l \in \mathbb{L}$$

Table 3
Execution times (in ms) for the task set, detailed architecture configurations and models of Section 4.1. All execution times are optimals for the models. The 185 ms C1F1C2F2 $\mathcal{MAP}$ execution time is the absolute minimum possible for this task set and architecture from [45] as discussed in the text of this section.

Approach	Configuration						
	C1	C1F2	C1F1	C1F1F2	C1C2	C1F1C2	C1F1C2F2
$\mathcal{HOM}$	1180	1180	1180	1180	591	591	591
$\mathcal{PAR}$	1180	1092	789	642	591	402	322
$\mathcal{SEQ}$	1180	1057	682	682	591	388	360
$\mathcal{MAP}$	1180	859	345	322	591	192	185

Table 4
Performance accelerations for the task set, detailed architecture configurations and models of Section 4.1. The accelerations are simply the single CPU execution time for the tasks (1180 ms) divided by the corresponding model and configuration execution times from Table 3 as defined in Eq. (37). The 6.378 times C1F1C2F2 $\mathcal{MAP}$ acceleration is the absolute maximum possible for this task set and architecture from [45] as discussed in the text.

Approach	Configuration						
	C1	C1F2	C1F1	C1F1F2	C1C2	C1F1C2	C1F1C2F2
$\mathcal{HOM}$	1.000×	1.000×	1.000×	1.000×	2.000×	2.000×	2.000×
$\mathcal{PAR}$	1.000×	1.081×	1.496×	1.838×	2.000×	2.935×	3.665×
$\mathcal{SEQ}$	1.000×	1.116×	1.730×	1.730×	2.000×	3.041×	3.278×
$\mathcal{MAP}$	1.000×	1.374×	3.420×	3.665×	2.000×	6.146×	6.378×

Table 5
The percentage of the available FPGA space used by the assignments of Table 3 in accordance with Eq. (38). The CPU only configurations of C1 and C1C2 have no FPGA capacity and so Eq. (38) is undefined for these columns (signified by a blank).

Approach	Configuration						
	C1	C1F2	C1F1	C1F1F2	C1C2	C1F1C2	C1F1C2F2
$\mathcal{HOM}$		0.0%	0.0%	0.0%		0.0%	0.0%
$\mathcal{PAR}$		30.8%	40.7%	75.9%		40.7%	75.9%
$\mathcal{SEQ}$		41.8%	45.0%	22.5%		22.0%	22.5%
$\mathcal{MAP}$		100.0%	100.0%	97.4%		100.0%	56.8 %

to allow tasks to use all of the space on the hardware they are assigned to while they sequentially execute.

4.2. Assignment results

Tables 3 and 4 show the execution times and performance accelerations for the six task set of Table 2 assigned to different configurations of Fig. 2(a) using MAP and the three alternate assignment approaches discussed in Section 4.1. The times of Table 3 are quoted in milliseconds and the accelerations shown in Table 4 were calculated using the equation:

$$\text{Acceleration} = \frac{t_{C1}}{t^*} \tag{37}$$

where t_{C1} is the time for the task set to execute on a single CPU and t^* is the optimal execution time for the corresponding model and configuration from Table 3.

Perhaps the first point to note in Table 3 is that all the approaches give the same results for the homogeneous CPU only C1 and C1C2 configurations. This is as expected because there is no code-sign flexibility to take advantage of. The CPU only configuration results thus act as model confirmation points.

The second point to note in Table 3 is that the $\mathcal{MAP}$ execution times are never worse than the alternative approaches. Indeed, the $\mathcal{MAP}$ model results are better than every alternative approach for all the heterogeneous configurations and deliver up to 2.021 times better performance than the best of the alternatives (i.e. $\frac{388\text{ ms}}{192\text{ ms}}$ for the C1F1C2 configuration).

The homogeneous $\mathcal{HOM}$ results are the worst overall in Table 3. This is because the homogeneous approach only assigns to the CPU components of the heterogeneous configurations, in effect limiting itself to the one and two CPU C1 and C1C2 configurations as we add FPGAs. If we replaced the FPGAs by CPUs rather than simply not using the FPGAs, the corresponding homogeneous execution times would be 591, 591, 426, 426 and 352 ms for the modified C1F2, C1F1, C1F1F2, C1F1C1 and C1F1C2F2 configurations which would beat the parallel and sequential results of Table 3. The heterogeneous MAP results would still however be better than the higher CPU homogeneous results for all but the high latency C1F2 configuration [45]. Remember though that by comparing C1F2 for MAP with the homogeneous approach for a configuration with F2 replaced by a CPU we are actually comparing C1F2 $\mathcal{MAP}$ against a C1C2 $\mathcal{HOM}$ which is not exactly fair because $\mathcal{MAP}$ would identify the same partition as $\mathcal{HOM}$ for C1C2 if allowed to as shown in Table 3.

Table 5 shows the percentage of the available FPGA space used by each of the optimal assignments of Table 3 which we define as:

$$\text{Utilisation} = \frac{\sum_{i,p,f} x^i_{pf}\delta^i_{pf}}{\sum_f \Delta_f} \tag{38}$$

where f is a location from the set of space constrained FPGA locations and the other symbols are as described previously and summarised in Table 10.

The CPU-only configurations have no FPGA capacity and so Eq. (38) is undefined for the C1 and C1C2 columns which are left blank in Table 5. The $\mathcal{HOM}$ row is zero for all configurations because the homogeneous approach does not use the FPGA components by definition. All the remaining configurations and models use the available FPGA space to some extent and we will now detail that usage for the interested reader.

Referring to the parallel capacity utilisation row of Table 5, the $\mathcal{PAR}$ model assigns all code sections of the `sha` task (excluding system calls) to the FPGA with the C1F2 configuration and with the lower latency C1F1 and C1F1C2 configurations the approach assigns all non-system call code for the `dijkstra` and `sha` tasks to the FPGA. With the higher capacity C1F1F2 and C1F1C2F2 configurations the $\mathcal{PAR}$ model assigns the non-system call sections of `dijkstra`, `fft` and `sha` to F1 and of `susan` to F2.

For the sequential capacity utilisation results of Table 5, the $\mathcal{SEQ}$ model assigns parts of all tasks except `dijkstra` to the reconfigurable hardware with the C1F2 configuration and with the C1F1, C1F1F2 and C1F1C2F2 configurations, the $\mathcal{SEQ}$ model uses the FPGAs to accelerate parts of all the tasks. The reason why `dijkstra` is not selected for acceleration with the C1F2 configuration is that the inter-node latency, which is over 500 times larger than the intra-node latency for this architecture, cannot be offset by the code-sign benefits as explained in [45]. For the C1F1C2 configuration, the single FPGA represents a dedicated execution bottleneck for the $\mathcal{SEQ}$ model and with this configuration $\mathcal{SEQ}$ does not accelerate the `fft` and `ispell` tasks which it places on C2 alone rather than delaying the other tasks which it accelerates by placing parts of them on both the C1 and F1 components. If we ran the $\mathcal{SEQ}$ model for the heterogeneous configurations of Table 5 ignoring reconfiguration costs (i.e. $R^i_{lm|x} = 0\ \forall i, lm, x$), the capacity utilisations of the optimal sequential assignments selected by CPLEX would increase to 192.5%, 314.7%, 160.1%, 213.3% and 157.3% (remember the sequential method re-uses the FPGA space) and the optimal execution times decrease to 906, 545, 545, 334 and 277 ms which you will note are still far worse than the $\mathcal{MAP}$ execution timings of Table 3.

For the MAP results of Table 5 we see that the $\mathcal{MAP}$ model is able to use most of the available hardware with all but one configuration. For the high-latency C1F2 configuration MAP assigns parts of all but the `dijkstra` benchmark to the FPGA like

Table 6
Execution times (in ms) for the task set and multi-node architecture configurations of Section 4.1. The row entries stop at the minimum possible time for each approach. The absolute minimum time is 185 ms for the task set and architecture from [45] which is only achieved by the $\mathcal{MAP}$ model. The 1 and 2 Axel node figures below correspond to the C1F1 and C1F1C2F2 configurations of Table 3.

Approach	Axel cluster nodes					
	1	2	3	4	5	6
$\mathcal{HOM}$	1180	592	426	352	298	292
$\mathcal{PAR}$	789	322	256	200		
$\mathcal{SEQ}$	682	360	263	196	187	
$\mathcal{MAP}$	345	185				

Table 7
The optimal performance accelerations for the execution times of Table 6. The accelerations are simply the single CPU execution time for the tasks (1180 ms) divided by the corresponding model execution time from Table 6 in accordance with Eq. (37). The 1 and 2 Axel node figures below correspond to the C1F1 and C1F1C2F2 configurations of Table 4.

Approach	Axel cluster nodes					
	1	2	3	4	5	6
$\mathcal{HOM}$	1.000×	2.000×	2.770×	3.352×	3.960×	4.041×
$\mathcal{PAR}$	1.496×	3.665×	4.609×	5.900×		
$\mathcal{SEQ}$	1.730×	3.278×	4.487×	6.020×	6.310×	
$\mathcal{MAP}$	3.420×	6.378×				

Table 8
The optimal percentage of the available FPGA space used by the assignments of Table 6 calculated in accordance with Eq. (38). The 1 and 2 Axel node figures below correspond to the C1F1 and C1F1C2F2 configurations of Table 5.

Approach	Axel cluster nodes					
	1	2	3	4	5	6
$\mathcal{HOM}$	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
$\mathcal{PAR}$	40.7%	75.9%	50.6%	34.1%		
$\mathcal{SEQ}$	45.0%	22.5%	15.0%	11.3%	9.0%	
$\mathcal{MAP}$	100.0%	56.8%				

the $\mathcal{SEQ}$ model and for the other configurations, parts of all tasks are accelerated with the FPGAs. For the full two-node C1F1C2F2 configuration the MAP approach has a lower FPGA utilisation factor than the parallel approach. This is because the $\mathcal{PAR}$ model works at a coarse-grained level and assigns all of a task's non-system call code to an FPGA if the net effect is beneficial whereas the $\mathcal{MAP}$ model has the extra dimension of fine-grained assignment flexibility which results in a 42% faster execution time with over 25% less of the reconfigurable space used.

Less than 14% of the second FPGA is used with the C1F1C2F2 $\mathcal{MAP}$ configuration which would seem to suggest that there is little serialisation removal benefit to be obtained by adding more parallel components to the two-CPU architecture with MAP. In fact, the 185 ms execution time for the C1F1C2F2 $\mathcal{MAP}$ configuration quoted in Table 3 is actually the absolute minimum for the six task set running on any number of Axel nodes which we can confirm by examining the results of [45] where it can be seen the `fft` benchmark cannot be accelerated by more than 1.081 times $\left(\frac{200\text{ ms}}{1.081\text{x}} = 185\text{ ms}\right)$ and is thus largely CPU bound on this architecture which explains why $\mathcal{MAP}$ places the `fft` task on C2 alone with the C1F1C2F2 configuration.

We conclude this section with Tables 6–8 which provide the minimum execution times obtainable with the different approaches for the task set along with their corresponding accelerations and utilisations as we activate additional nodes in Fig. 2(b). It can be seen that MAP is the only approach to deliver the true absolute minimum of 185 ms as mentioned above, that MAP is consistently better than the other approaches and that all the other approaches require considerably more hardware resources (Axel nodes) to reach their respective minimums.

4.3. Bounds and complexity

The results of Section 4.2 showed that the MAP approach consistently performed at least as good as the alternatives for the tasks and architecture configurations considered. In this section we generalise the specific results provided to obtain bounds for MAP performance improvements and provide practical solution timing information.

Let us start by referring back to Section 4.1 where we constructed our formal homogeneous model $\mathcal{HOM}$ by adding constraints to $\mathcal{MAP}$ that ensured only the CPU components of a configuration could be used. By doing this we made the feasibility set of the $\mathcal{HOM}$ model more restrictive than that of $\mathcal{MAP}$ and, as we used the same objective function for both models, this implies the $\mathcal{MAP}$ optimals will be at least as good as $\mathcal{HOM}$ no matter what set of tasks or architecture/configuration we use [33]. Or, stated in first-order logic:

$$\mathbb{X}_{\mathcal{HOM}} \subseteq \mathbb{X}_{\mathcal{MAP}} \Longrightarrow \exists x^* \in \mathbb{X}_{\mathcal{MAP}} : t_{x^*} \leq \min_{x \in \mathbb{X}_{\mathcal{HOM}}} t_x.$$

Looking next at the parallel approach we see from Section 4.1 that we added constraints to the $\mathcal{MAP}$ model again, this time to force assignment on a coarse-grained basis. Thus, we have:

$$\mathbb{X}^*_{\mathcal{HOM}} \subseteq \mathbb{X}_{\mathcal{PAR}} \subseteq \mathbb{X}_{\mathcal{MAP}}$$

as every coarse-grained assignment is a possible MAP assignment and every optimal homogeneous (CPU only) assignment is a possible parallel assignment[2] and, because we used the same objective with these three models, we have the relation:

$$t^*_{\mathcal{HOM}} \geq t^*_{\mathcal{PAR}} \geq t^*_{\mathcal{MAP}} \quad \forall \mathbb{T} \in \mathbb{S}, \mathbb{A} \in \mathbb{H}$$

where $\mathbb{T}$ is a set of tasks from the set of all possible software task sets $\mathbb{S}$ and $\mathbb{A}$ is an architecture from the set of all possible hardware architectures $\mathbb{H}$. In other words, $\mathcal{MAP}$ is guaranteed to produce at least as good execution times as the homogeneous model $\mathcal{HOM}$ and the parallel model $\mathcal{PAR}$ for all tasks and architectures.

Now let us turn our attention to the sequential model. The sequential model $\mathcal{SEQ}$ was also constructed from the $\mathcal{MAP}$ model in Section 4.1, but our approach was different. Firstly, the objective time was increased with reconfiguration costs and a more coarse definition of $\bar{t}^{\,i}_{\otimes}$ which, on their own, would mean $t^*_{\mathcal{SEQ}} \geq t^*_{\mathcal{MAP}}$ however we then went on to modify the space constraints from:

$$\sum_{i \in \mathbb{T}, p \in \mathbb{P}^i} x^i_{pl} \delta^i_{pl} \leq \Delta_l \quad \forall l \in \mathbb{L}$$

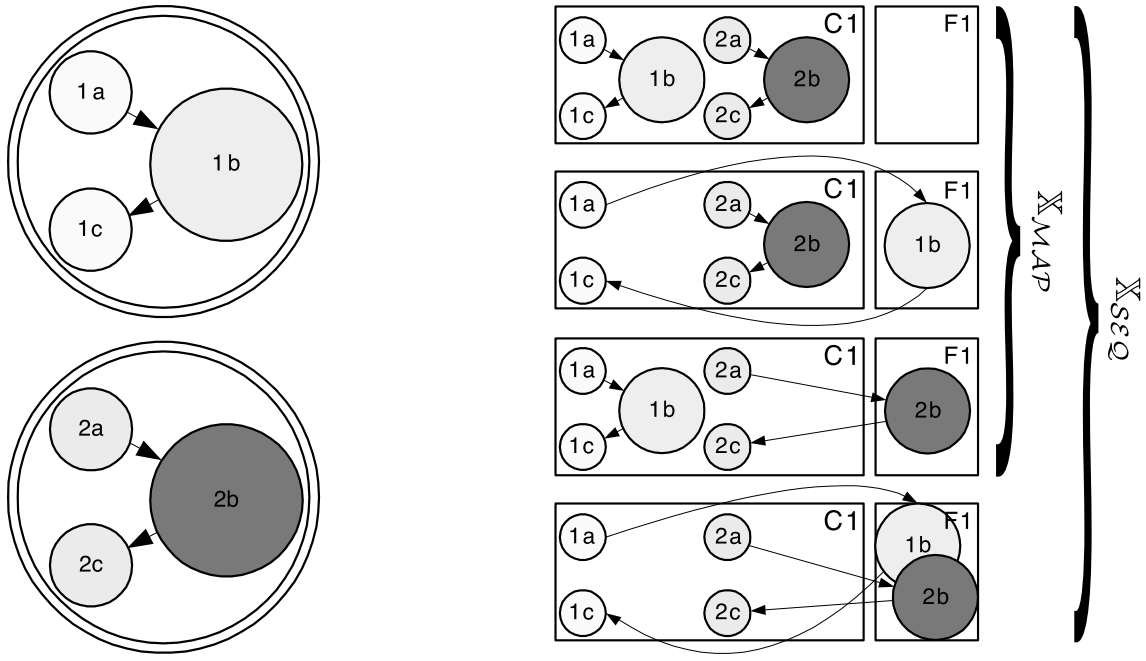
to:

$$\sum_{p \in \mathbb{P}^i} x^i_{pl} \delta^i_{pl} \leq \Delta_l \quad \forall i \in \mathbb{T}, l \in \mathbb{L}$$

to allow tasks to use all the hardware they are assigned to while they execute sequentially. This modification (not addition) of constraints means that the $\mathcal{MAP}$ feasibility set is actually a subset of the $\mathcal{SEQ}$ feasibility set i.e. $\mathbb{X}_{\mathcal{MAP}} \subseteq \mathbb{X}_{\mathcal{SEQ}}$ not the other way round and so while the $\mathcal{SEQ}$ objective can never be better than the $\mathcal{MAP}$ objective for the same assignments, the $\mathcal{SEQ}$ model can choose from additional assignments where the $\mathcal{SEQ}$ objective may be better than the best available for the $\mathcal{MAP}$ model.

In fact, it is easy to conceive an example where $t^*_{\mathcal{SEQ}} < t^*_{\mathcal{MAP}}$. To do this, consider a task set $\mathbb{T} \in \mathbb{S}$ with just two tasks where each task has only three code sections as illustrated in Fig. 3(a), the first

[2] Remember that we did not formally restrict our $\mathcal{HOM}$ model to be coarse-grained in Section 4.1, however characteristic 1 of Definition 3 means that every optimal $\mathcal{HOM}$ assignment will be coarse-grained and so $\mathbb{X}^*_{\mathcal{HOM}} \subseteq \mathbb{X}_{\mathcal{PAR}}$ for practical problems.



(a) Two tasks with three code sections each.

(b) The feasible assignments for the MAP and sequential approaches labelled $\mathbb{X}_{\mathcal{MAP}}$ and $\mathbb{X}_{\mathcal{SEQ}}$ respectively.

Fig. 3. Two tasks with one freely assignable code section each (labelled 1b and 2b) which is more than half the size of the available space constrained co-processor labelled F1 and the associated MAP and sequential feasibility sets labelled $\mathbb{X}_{\mathcal{MAP}}$ and $\mathbb{X}_{\mathcal{SEQ}}$ respectively. With the MAP method, it is not possible to place both 1b and 2b on the co-processor at the same time because this would violate the space constraints. The sequential method however has an additional feasible assignment where 1b and 2b are both placed on the space constrained hardware because the co-processor is dedicated to task 1 first then re-used for task 2 in isolation.

and third of which (the start and end nodes) must be assigned to a CPU. Next, let us select a two-location heterogeneous architecture $\mathbb{A} \in \mathbb{H}$ where the second node of each task is larger than half the available co-processor space and where the co-processor's code-sign benefits far exceed its reconfiguration costs and we have our example because $\mathcal{SEQ}$ could re-use the hardware to get up to twice (n times for n tasks) the speed-up of $\mathcal{MAP}$ using the additional feasible sequential assignment shown in Fig. 3(b).

We cannot relate the parallel and sequential optimals for the same reason we could not relate the MAP and sequential optimals (i.e. while the $\mathcal{PAR}$ model has lower costs than $\mathcal{SEQ}$ for the same assignments the $\mathcal{SEQ}$ model has a larger feasibility set), however we can say that:

$$t^*_{\mathcal{HOM}} \geq t^*_{\mathcal{SEQ}} \quad \forall \mathbb{T} \in \mathbb{S}, \mathbb{A} \in \mathbb{H}$$

because keeping all the code on the CPU is a feasible assignment for both the homogeneous and sequential approaches and such an assignment does not have significant reconfiguration costs (at least in this paper) so:

$$t_{\mathcal{HOM}} = t_{\mathcal{SEQ}}$$
$$\forall x \in \mathbb{X}_{\mathcal{HOM}} \subseteq \mathbb{X}_{\mathcal{SEQ}} \Longrightarrow \exists x^* \in \mathbb{X}_{\mathcal{SEQ}} : t^*_{\mathcal{HOM}} \geq t^*_{\mathcal{SEQ}}.$$

Bringing all this together we have the following general bounding relations for the optimal $\mathcal{MAP}$ assignment compared with the optimals possible for the other approaches:

$$t^*_{\mathcal{HOM}} \geq \begin{cases} t^*_{\mathcal{PAR}} \geq t^*_{\mathcal{MAP}} \\ t^*_{\mathcal{SEQ}} \end{cases} \quad \forall \mathbb{T} \in \mathbb{S}, \mathbb{A} \in \mathbb{H} \tag{39}$$

and it is the subject of future work to bring together the MAP and sequential benefits to rationalise the above relations for all task sets and architectures.

Let us now turn our attention to practical MAP solution complexity. As already mentioned in Section 3.4, MAP is easily recognised as being a strongly NP-hard problem through its quadratic communication costs c^i_{pqlm} and binary assignment variables x^i_{pl} [42,15,36]. In fact, the combinatorial complexity of the MAP solution space is:

$$\mathcal{O}(|\mathbb{L}|^{|\mathbb{P}||\mathbb{T}|}) \tag{40}$$

where $|\mathbb{L}|$ is the number of assignment locations, $|\mathbb{P}|$ the number of assignable code sections in a representative task and $|\mathbb{T}|$ the number of tasks to be assigned in the parallel task set.

Eq. (40) shows that the MAP solution space complexity is particularly sensitive to the number of assignable locations $|\mathbb{L}|$, and we see the effect of this sensitivity when we look at the solution times for the detailed configurations shown in Table 9. While it is perhaps worth stressing that the solution times of Table 9 are for the assignment of 3,373 code sections $\left(\sum_i |\mathbb{P}^i|\right.$ from Table 2$\left.\right)$ which is over 146 times the 23 code sections considered for the 3-location problem of [46] which takes over 26 hours to solve, the $\mathbb{L}$ sensitivity still prevented the C1F1C2F2 configuration from completing within our 24 hour cut-off on CPLEX 12.2 in this work. To get the C1F1C2F2 $\mathcal{MAP}$ optimals and utilisation figures of Section 4.2 we used the C1F1C2 $\mathcal{MAP}$ assignments as a starting point and the solution polishing feature of CPLEX 12.2 [9] which we were able to terminate after around 1 hour when an integer solution with the known minimum optimal value of 185 ms for this task set and architecture [45] was found.

Table 9 also shows that the additional constraints on the $\mathcal{HOM}$ and $\mathcal{PAR}$ models generally shorten their practical solution times with respect to $\mathcal{MAP}$. However, the *different* objective and constraints on the $\mathcal{SEQ}$ model resulted in increased solution times over $\mathcal{MAP}$ for the multiple CPU configurations with even the three location C1F1C2 $\mathcal{SEQ}$ problem not solving to optimality in our 24 hour time limit.

Table 9
Practical solution times for the assignments of Table 3. The solution times were measured on a 2.13 GHz Intel Core 2 Duo Apple Mac with 4GB of memory using the default configuration of 64-bit CPLEX 12.2. All practical implementations had the symmetry reduction constraints $\sum_i \delta^i_{pl} \leq \sum_i \delta^i_{pm} \forall\, l, m \in \mathbb{L}_{CPU} : m > l$ added and used CPLEX indicators to replace big-M constraints which we found to reduce solution times over other implementation alternatives. The solution process was stopped if no optimal was identified in 24 hours as discussed in the text.

Approach	Configuration						
	C1	C1F2	C1F1	C1F1F2	C1C2	C1F1C2	C1F1C2F2
$\mathcal{HOM}$	0.01	0.10	0.10	0.29	229.49	320.22	117.38
$\mathcal{PAR}$	0.04	0.68	0.62	4.34	0.83	4.89	11.96
$\mathcal{SEQ}$	0.01	0.87	0.99	9.48	872.86	>24 h	>24 h
$\mathcal{MAP}$	0.01	29.24	10.29	22924.80	143.17	6729.66	>24 h

The fact that the $\mathcal{SEQ}$ model did not arrive at a known bound within 24 hours was an issue and to get the sequential results for the C1F1C2 and C1F1C2F2 configurations presented in Table 3 and, indeed, the results for the larger configurations presented in Table 6 we needed to use a re-formulation based on the insight that sequential assignments do not share space on the components they are assigned to by definition and thus the sequential assignment problem is akin to choosing a set of components from the configuration for each task to execute on in isolation such that the combined execution times are minimised. With this insight, the sequential optimisation problem can be expressed as:

$$\min\ t_x \tag{41}$$

$$\text{s.t. } t_x \geq \sum_{\substack{\mathrm{a}\,:\,\mathrm{a}\subseteqq\mathbb{A} \\ \wedge\ \Phi^{\mathrm{b}}_{\mathrm{a}}=1}} \sum_{j\in\mathbb{T}} x^j_{\mathrm{a}} t^j_{\mathrm{a}} - M'(1-x^i_{\mathrm{b}}) \quad \forall i \in \mathbb{T}, \mathrm{b} : \mathrm{b} \subseteqq \mathbb{A} \tag{42}$$

$$\sum_{\mathrm{a}\,:\,\mathrm{a}\subseteqq\mathbb{A}} x^i_{\mathrm{a}} = 1 \quad \forall i \in \mathbb{T} \tag{43}$$

where a and b are valid sub-configurations for architecture $\mathbb{A}$ identified using the new symbol $\subseteqq$ with an example below, $\Phi^{\mathrm{b}}_{\mathrm{a}}$ is a sub-configuration clash indicator that is defined to be 1 if sub-configuration a uses any of the same components or channels as sub-configuration b and 0 otherwise for the sequential approach, x^i_{a} is a sub-configuration selection indicator that is 1 if task i executes on sub-configuration a and 0 otherwise, M' is a big-M constant and t^j_{a} is the optimal isolated execution time for task j running on sub-configuration a which is considered constant in the formulation.

We will not spend long discussing the details of the sequential reformulation as it is not the main point of this paper. However, to aid the interested reader in understanding the above, the C1F1C2 architectural configuration of Fig. 2(a) could be described by the $\mathbb{A}$, a, b and $\Phi^{\mathrm{b}}_{\mathrm{a}}$ below:

$$\begin{aligned}
\mathbb{A} &= \{\mathrm{C1}, \mathrm{F1}, \mathrm{C2}\} \\
\mathrm{a}, \mathrm{b} &\in \{\{\mathrm{C1}, \mathrm{F1}, \mathrm{C2}\}, \{\mathrm{C1}, \mathrm{F1}\}, \{\mathrm{F1}, \mathrm{C2}\}, \{\mathrm{C1}, \mathrm{C2}\}, \{\mathrm{C1}\}, \{\mathrm{C2}\}\} \\
\Phi^{\mathrm{b}}_{\mathrm{a}} &= \begin{cases} 1 & \text{if } \mathrm{b} \cap \mathrm{a} \neq \emptyset \\ 0 & \text{otherwise,} \end{cases}
\end{aligned}$$

with the corresponding t^j_{a} values obtained using a pre-stage to solve the original $\mathcal{SEQ}$ model for tasks assigned in isolation to the a sub-configurations.

The reformulated sequential model has the same optimals as the original form but generally has a far lower solution space complexity, namely:

$$\mathcal{O}(2^{|\mathbb{L}||\mathbb{T}|}) \tag{44}$$

and this allowed us to obtain all the sequential results, including the 10-location 5-node Axel results, in less than one second each reusing isolated t^j_{a} constants obtained from pre-stage subproblems which were solved in a combined time of 1.6 s for all valid Axel sub-configurations (taking advantage of architectural symmetry). A similar multi-level Φ reformulation was used to obtain the three to six Axel node homogeneous results of Section 4.2 which were all obtained within one second again and the additional parallel results of Table 6 were obtained using the unaltered $\mathcal{PAR}$ model of Section 4.1 which solved in less than two minutes.

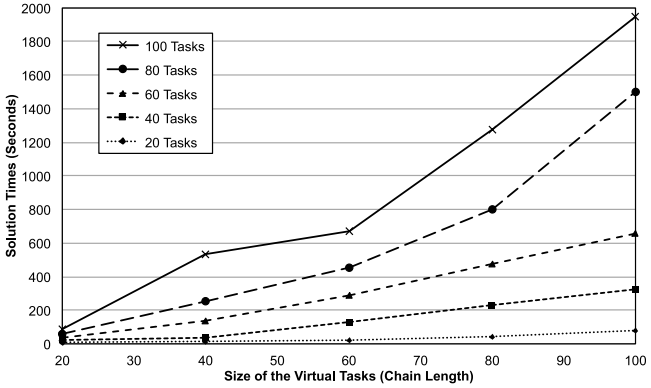


Fig. 4. Time to solve $\mathcal{MAP}$ problems for different numbers of tasks and task sizes with the C1F1 configuration of Fig. 2(a) as discussed in the text. The solution times were measured on a 2.13 GHz Intel Core 2 Duo Apple Mac with 4 GB of memory using the default configuration of 64-bit CPLEX 12.2 and the same symmetry reduction and implementation choices as Table 9.

It is worth noting that while our multi-level Φ reformulation approach outlined above is applicable to the $\mathcal{SEQ}$, $\mathcal{HOM}$ and indeed (with some minor modifications) the $\mathcal{PAR}$ models, the method cannot be applied directly to the $\mathcal{MAP}$ model. This is because the $\mathcal{SEQ}$, $\mathcal{HOM}$ and $\mathcal{PAR}$ interactions are all coarse-grained in one sense or another whereas the optimal $\mathcal{MAP}$ assignment for a task depends on the fine-grained assignments of other tasks which means that $t^j_{\mathrm{a}} \rightarrow t^j_{\mathrm{a}|x}$ for $\mathcal{MAP}$ and it is the subject of future work to explore appropriate MAP reformulations along with other techniques [44,52,23,37,27] to deal with the practical solution time issues already seen for the four location MAP problem instance.

Despite the future work discussed above, the solution times of Table 9 indicate that the MAP formulation presented in this paper can be used immediately with current two-component tightly-coupled heterogeneous research architectures [48,30,29, 40,5,31,26]. To investigate the utility of MAP with such architectures, we constructed new tasks by chaining together the detailed 3S control and data flow measurements for the basic tasks of Table 2 to create larger virtual tasks and then measured the solution times for sets of these larger virtual tasks assigned to the C1F1 configuration of Fig. 2 with the model of Definition 4. The solution times are shown in Fig. 4 and demonstrate that we can increase $|\mathbb{P}||\mathbb{T}|$ over a wide range and still obtain rapid MAP solutions for C1F1 type architectures.

To put the timings of Fig. 4 in perspective, the top right hand point on the graph represents the time to identify the optimal $\mathcal{MAP}$ solution for a problem with 5,621,548 code sections and a solution space of the order of $2^{5,621,548}$ from Eq. (40). Or, put another way, the top right hand data point is the time to assign 100 tasks which are all over 10 times the size of the current Apache Web Server distribution [4] when measured in lines of C code to a two-component heterogeneous architecture. The corresponding $\mathcal{MAP}$ model is 2.38 GB in size when represented as a LP file and solves to optimality in around half an hour.

Table 10
The main symbols used in this paper (in alphabetical order). The source column gives the original source of the data used to generate the results of Section 4 with [MAP] used to signify internal model variables. We use $\mathbb{B}$ to denote the set $\{0, 1\}$, $\mathbb{R}_+$ the set of real non-negative numbers i.e. $\mathbb{R}_+ = \{a \in \mathbb{R} : a \geq 0\}$ and $\mathbb{Z}_+$ for the set of non-negative integers in this paper.

Symbol	Source	Description
$\alpha^i_{l\|x} \in \mathbb{B}$	[MAP]	1 if any part of task i is assigned to location l
$\beta^i_{lm\|x} \in \mathbb{B}$	[MAP]	1 if any part of task i communicates between l and m
$\gamma^i_{lm\|x} \in \mathbb{B}$	[MAP]	1 if any part of task i spends time executing on lm
$\delta^i_{pl} \in \mathbb{R}_+$	[41,42]	The size of code section p from task i on location l
$\Delta_l \in \mathbb{R}_+$	[50]	The size capacity of location l
$\eta^i_{pq} \in \mathbb{Z}_+$	[41,42]	The data flows from p to q in task i
$\mu^i_{pl} \in \mathbb{R}_+$	[45]	The total computation time of code section p at location l
$\chi^i_{pq} \in \mathbb{Z}_+$	[41,42]	The number of control flows from p to q in task i
$c^i_{pqlm} \in \mathbb{R}_+$	[45]	The total time for communications from p on l to q on m
$i, j \in \mathbb{T}$	[14]	Task identifiers in the parallel DAG task set $\mathbb{T}$
$l, m \in \mathbb{L}$	[50]	Computational locations
$\mathbb{L}$	[50]	The set of all computation assignment locations
$\mathbb{L}^p_e \subseteq \mathbb{L}$	[50]	Locations with access to external code required by $p \in \mathbb{P}^i_e$
$\mathbb{L}_\perp \subseteq \mathbb{L}$	[50]	The set of sequential computation locations
$\mathbb{M}_\perp \subseteq \mathbb{L} \times \mathbb{L}$	[50]	The set of shared communication channel ordered pairs
$\mathbb{P}^i$	[41,42]	The set of code sections within task i
$\mathbb{P}^i_e \subseteq \mathbb{P}^i$	[41,42]	The set of code sections calling external code in task i
$p, q \in \mathbb{P}^i$	[41,42]	Code sections within task i
$r, s \in \mathbb{P}^j$	[41,42]	Code sections within task j
$\mathbb{T}$	[14]	The set of parallel DAG tasks
$t_x \in \mathbb{R}_+$	[MAP]	Time task set $\mathbb{T}$ takes to execute (including delays) for x
$t^i_x \in \mathbb{R}_+$	[MAP]	Time task i takes to execute (including delays) for x
$t^i_{lm\|x} \in \mathbb{R}_+$	[MAP]	Time task i would spend on lm if executing in isolation
$\bar{t}^{\,i}_{lm\otimes} \in \mathbb{R}_+$	[MAP]	Worst case serialisation delay i could experience on lm
$\bar{t}^{\,i}_{\otimes} \in \mathbb{R}_+$	[MAP]	Total worst case serialisation delay i could experience
$\mathbb{X} \subseteq \mathbb{B}^{\|\mathbb{L}\|\|\mathbb{P}\|\|\mathbb{T}\|}$	[MAP]	The set of feasible assignments to $\mathbb{L}$ of code $\mathbb{P}$ in tasks $\mathbb{T}$
$x \in \mathbb{X}$	[MAP]	One of the feasible assignments from the set $\mathbb{X}$
$x^* \in \mathbb{X}$	[MAP]	An optimal assignment from the feasibility set $\mathbb{X}$
$x^i_{pl} \in \mathbb{B}$	[MAP]	1 if code section p from task i is assigned to location l

5. Conclusion

This paper presented the Multi-level Assignment Partitioning (MAP) approach which optimises parallel task execution on shared distributed hardware components using robust sequential assignment. The paper contributed:

1. a formal model for partitioning coarse-grained parallel tasks to shared distributed hardware components using fine-grained sequential assignment.
2. insights to simplify the formal model for solution using standard solvers.
3. experimental results demonstrating the performance improvements possible over previous approaches for a suite of benchmarks.

In Section 3 our formal model was presented along with simplifying insights which brought the number of binary variables in the problem down by orders of magnitude. Section 4 then provided assignment results for a parallel task set consisting of six standard benchmarks. Results were provided for formal homogeneous assignment (using just CPUs), coarse-grained parallel task assignment, sequential assignment (with reconfiguration costs) and our Multi-level Assignment Partitioning model with a range of hardware configurations. The results showed that MAP produced program accelerations of 6.378 times with only two Axel cluster nodes [50] where the other approaches took up to three times more hardware to achieve their respective optimals, all of which were below that achievable with MAP.

In Section 4.3 we generalised our specific results with bounds that proved that MAP will always be at least as good as the homogeneous and parallel models and we provided an existence proof demonstrating that MAP cannot be guaranteed to always be better than the sequential method which we hope will encourage future work in this area. We concluded the main body of the paper with practical solution timing information that demonstrated the immediate utility of the formal MAP model with the current research focus on heterogeneous architectures and provided possible directions for research to extend the applicability of the formal method for future practical problems.

Acknowledgments

The support of the UK EPSRC and the European Union Seventh Framework Programme under Grant agreement numbers 248976, 257906 and 287804 is gratefully acknowledged. Additionally we thank Dr L.W. Howes, Dr K.H. Tsoi and Dr Y.M. Lam who provided valuable contributions, and the reviewers whose insightful comments helped improve this paper greatly.

References

[1] S. Ali, J.-K. Kim, H. Siegel, A. Maciejewski, Static heuristics for robust resource allocation of continuously executing applications, Journal of Parallel and Distributed Computing 68 (8) (2008) 1070–1080.
[2] S. Ali, H. Siegel, M. Maheswaran, D. Hensgen, S. Ali, Representing task and machine heterogeneities for heterogeneous computing systems, Journal of Science and Engineering 3 (2000) 195–207.
[3] Advanced Micro Devices Inc., Desktop Processor Solutions, Advanced Micro Devices Inc., 2010.
[4] The Apache Software Foundation, Apache HTTP server 2.4.2, 2012. http://httpd.apache.org/download.cgi#apache24.
[5] K. Atasu, C. Özturan, G. Dündar, O. Mencer, W. Luk, CHIPS: Custom Hardware Instruction Processor Synthesis, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27 (3) (2008) 528–541.
[6] A. Beletska, W. Bielecki, A. Cohen, M. Palkowski, K. Siedlecki, Coarse-grained loop parallelization: iteration space slicing vs affine transformations, Parallel Computing 37 (8) (2011) 479–497.
[7] T. Braun, H. Siegel, N. Beck, L. B ol oni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, R. Freund, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 61 (6) (2001) 810–837.
[8] C.-C. Chang, J. Cong, X. Yuan, Multi-level placement for large-scale mixed-size IC designs, in: Proc. of the ACM Asia and South Pacific Design Automation Conference, 2003, pp. 325–330.
[9] IBM Corp., CPLEX 12.1 Manuals, IBM Corp., 2008.
[10] nVidia Corp., nVidia Cuda Programming Guide, nVidia Corp., 2009.
[11] P. Eles, Z. Peng, A. Kuchcinski, A. Doboli, System level hardware/software partitioning based on simulated annealing and tabu search, Journal on Design Automation for Embedded Systems 2 (1997) 5–32.
[12] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, V. Volkov, Parallel computing experiences with CUDA, IEEE Micro 28 (4) (2008) 13–27.
[13] M. Girkar, C. Polychronopoulos, Automatic extraction of functional parallelism from ordinary programs, IEEE Transactions on Parallel and Distributed Systems 3 (2) (1992) 166–178.
[14] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown, MiBench: a free, commercially representative embedded benchmark suite, in: Proc. of the 2001 IEEE International Workshop on Workload Characterization, 2001, pp. 3–14.

[15] P. Hahn, B. Kim, M. Guignard, J. Smith, Y. Zhu, An algorithm for the generalized quadratic assignment problem, Computational Optimization and Applications 40 (3) (2008) 351–372.
[16] B. Holden, Latency comparison between HyperTransport and PCI-Express in communications systems, HyperTransport Consortium, 2006.
[17] M. Huang, V. Narayana, H. Simmler, O. Serres, T. El-Ghazawi, Reconfiguration and communication-aware task scheduling for high-performance reconfigurable computing, ACM Transactions on Reconfigurable Technology and Systems 3 (4) (2010) 20:1–20:25.
[18] S. Hunold, T. Rauber, G. Rünger, Combining building blocks for parallel multi-level matrix multiplication, Parallel Computing 34 (6) (2008) 411–426.
[19] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, B. Chapman, High performance computing using MPI and OpenMP on multi-core parallel systems, Parallel Computing 37 (9) (2011) 562–575.
[20] M. Khan, Scheduling for heterogeneous systems using constrained critical paths, Parallel Computing 38 (4) (2012) 175–193.
[21] Y.-K. Kwok, On exploiting heterogeneity for cluster based parallel multithreading using task duplication, Journal of Supercomputing 25 (1) (2003) 63–72.
[22] Y.-K. Kwok, I. Ahmad, Link contention-constrained scheduling and mapping of tasks and messages to a network of heterogeneous processors, Cluster Computing: The Journal of Networks, Software Tools and Applications 3 (2000) 113–124.
[23] Y.-K. Kwok, I. Ahmad, On multiprocessor task scheduling using efficient state space search approaches, Journal of Parallel and Distributed Computing 65 (12) (2005) 1515–1532.
[24] Y.-K. Kwok, A. Maciejewski, H. Siegel, I. Ahmad, A. Ghafoor, A semi-static approach to mapping dynamic iterative tasks onto heterogeneous computing systems, Journal of Parallel and Distributed Computing 66 (1) (2006) 77–98.
[25] Y. Lam, J. Coutinho, W. Luk, P. Leong, Integrated hardware/software codesign for heterogeneous computing systems, in: Proc. of the IEEE Southern Programmable Logic Conference, 2008, pp. 217–220.
[26] Y. Lam, J. Coutinho, W. Luk, P. Leong, Optimising multi-loop programs for heterogeneous computing systems, in: Proc. of the IEEE Southern Programmable Logic Conference, 2009, pp. 129–134.
[27] L. Liberti, Reformulations in mathematical programming: automatic symmetry detection and exploitation, Mathematical Programming (2010) 1–32.
[28] L. Liu, D. Shell, Multi-level partitioning and distribution of the assignment problem for large-scale multi-robot task allocation, in: Proc. of IEEE Robotics: Science and Systems, 2011.
[29] R. Lysecky, G. Stitt, F. Vahid, Warp processors, ACM Transactions on Design, Automation of Electronic Systems 11 (3) (2006) 659–681.
[30] R. Lysecky, F. Vahid, A configurable logic architecture for dynamic hardware/software partitioning, in: Proc. of the IEEE Conference on Design, Automation and Test in Europe, vol. 1, 2004, pp. 480–485.
[31] O. Mencer, D. Pearce, L. Howes, W. Luk, Design space exploration with a stream compiler, in: Proc. of the 2nd IEEE International Conference on Field-Programmable Technology, 2003, pp. 270–277.
[32] D. Naishlos, J. Nuzman, C.-W. Tseng, U. Vishkin, Towards a first vertical prototyping of an extremely fine-grained parallel programming approach, Theory of Computing Systems 36 (2003) 521–552.
[33] G. Nemhauser, L. Wolsey, Integer and Combinatorial Optimization, Wiley-Interscience, New York, NY, USA, 1988.
[34] M. Palis, The granularity metric for fine-grain real-time scheduling, IEEE Transactions on Computers 15 (2005) 1572–1583.
[35] F. Pratas, P. Trancoso, L. Sousa, A. Stamatakis, G. Shi, V. Kindratenko, Fine-grain parallelism using multi-core, Cell/BE, and GPU systems, Parallel Computing 38 (8) (2012) 365–390.
[36] S. Sahni, T. Gonzalez, P-complete approximation problems, Journal of the ACM 23 (3) (1976) 555–565.
[37] V. Shestak, E. Chong, H. Siegel, A. Maciejewski, L. Benmohamed, I.-J. Wang, R. Daley, A hybrid branch-and-bound and evolutionary approach for allocating strings of applications to heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 68 (4) (2008) 410–426.
[38] I. Shin, A. Easwaran, I. Lee, Hierarchical scheduling framework for virtual clustering of multiprocessors, in: Proc. of the 20th Euromicro Conference on Real-Time Systems, 2008, pp. 181–190.
[39] H. Siegel, S. Ali, Techniques for mapping tasks to machines in heterogeneous computing systems, Journal of Systems Architecture 46 (8) (2000) 627–639.
[40] G. Stitt, F. Vahid, A decompilation approach to partitioning software for microprocessor/FPGA platforms, in: Proc. of the IEEE Conference on Design, Automation and Test in Europe, vol. 3, 2005, pp. 396–397.
[41] S. Spacey, 3S: program instrumentation and characterisation framework, Tech. rep., Imperial College London (2008).
[42] S. Spacey, Computational partitioning for heterogeneous systems, Ph.D. Thesis, Imperial College London, 2009.
[43] S. Spacey, Concise CPLEX, Tech. rep., Imperial College London, 2009.
[44] S. Spacey, W. Luk, P. Kelly, D. Kuhn, Rapid design-space visualisation through hardware/software partitioning, in: Proc. of the IEEE Southern Programmable Logic Conference, 2009, pp. 159–164.
[45] S. Spacey, W. Luk, P.H.J. Kelly, D. Kuhn, Improving communication latency with the Write-Only Architecture, Journal of Parallel and Distributed Computing 72 (12) (2012) 1617–1627.
[46] S. Spacey, W. Wiesemann, D. Kuhn, W. Luk, Robust software partitioning with multiple instantiation, INFORMS Journal on Computing 24 (3) (2012) 500–515.
[47] M. Stillwell, D. Schanzenbach, F. Vivien, H. Casanova, Resource allocation algorithms for virtualized service hosting platforms, Journal of Parallel and Distributed Computing 70 (9) (2010) 962–974.
[48] G. Stitt, F. Vahid, Hardware/software partitioning of software binaries, in: Proc. of the 2002 IEEE/ACM International Conference on Computer-Aided Design, 2002, pp. 164–170.
[49] V. Sunderam, G. Geist, Heterogeneous parallel and distributed computing, Parallel Computing 25 (13) (1999) 1699–1721.
[50] K. Tsoi, W. Luk, Axel: a heterogeneous cluster with FPGAs and GPUs, in: Proc. of the 18th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2010, pp. 115–124.
[51] D. Turner, X. Chen, Protocol-dependent message-passing performance on Linux clusters, in: Proc. of the IEEE International Conference on Cluster Computing, 2002, pp. 187–194.
[52] T. Wiangtong, P. Cheung, W. Luk, Hardware/software codesign: a systematic approach targeting data-intensive applications, IEEE Signal Processing Magazine 22 (3) (2005) 14–22.
[53] Xilinx Inc., Vertex-5 Family Overview: Product Specification, Xilinx Inc., 2009.
[54] Xilinx Inc., Virtex-5 FPGA Configuration User Guide, Xilinx Inc., 2010.
[55] C.-T. Yang, K.-W. Cheng, W.-C. Shih, On development of an efficient parallel loop self-scheduling for grid computing environments, Parallel Computing 33 (7) (2007) 467–487.
[56] D. Zhu, X. Qi, D. Mossé, R. Melhem, An optimal boundary fair scheduling algorithm for multiprocessor real-time systems, Journal of Parallel and Distributed Computing 71 (10) (2011) 1411–1425.

Simon Spacey received a B.Sc. in Physics from York University, an M.Sc. in Quantum Devices from Lancaster University, a D.CSC. in Computer Science from Cambridge University and a Ph.D. in Computer Science from Imperial College London. He is currently head of research at SaSe Business Solutions. His research interests include distributed and high-performance computing systems, software characterisation, system modelling and formal optimisation.

Wayne Luk received his M.A., M.Sc., and D.Phil. degrees from the University of Oxford, Oxford, U.K., all in engineering and computing science. He is currently a Professor of computer engineering with the Department of Computing, Imperial College London, London, U.K., where he also leads the Custom Computing Group. His research interests include theory and practice of customising hardware and software for specific application domains, such as graphics and image processing, multimedia, and communications.

Daniel Kuhn received an M.Sc. in Theoretical Physics from ETH Zurich and a Ph.D. in Operations Research and Computational Finance from the University of St. Gallen/HSG. He has co-chaired the International Conference on Computational Management Sciences, is a reviewer for several mathematical journals and is a senior lecturer in the Computational Finance and Operations Research division of the Computer Science Department at Imperial College London.

Paul H.J. Kelly received his Ph.D. in Computer Science from London University. He has chaired the Software Track at the International Parallel and Distributed Processing Symposium (IPDPS), and has served on the Program Committees of the International Conference on Supercomputing, Compiler Construction, Euro-Par, and many other conferences and workshops and currently heads the Software Performance Optimisation group at Imperial College London.