LOAD BALANCE PROTOCOL OF CLUSTER ON GRID : PERVASIVE MAXIMUM ALGORITHMIC PARALLELISM

Tsuyoshi Okita*

Abstract: *Pervasive computing software helps to manage information and reduce the complexity of available computing resources in a timely manner. On one hand, a grid is a resource whose information is changing anytime and anywhere, where the availability of CPUs is only informed in run time when a cluster asks to the grid. On the other hand, a cluster is often implemented in a static way, which assumes some particular parallel architecture and the number of CPUs. Even when a cluster can consume maximum available CPU resources, if a cluster is implemented assuming the number of CPUs, a cluster could not run using more than this number of CPUs. Our mechanism of load balance protocol of cluster provides a dynamic way of implementing a cluster that can consume maximum available CPU resources on run time. In order to do so, we propose two mechanisms: to provide yet another (graphical) parallel language to describe clusters and to provide the protocols between a cluster on a grid. While many parallel languages resolve parallel architecture dependencies in compile time, our parallel language resolve parallel architecture dependencies in run time. Our load balance protocol bases on this (graphical) parallel language and it provides implementation of them.*

Keywords: *Cluster, Parallel System, Resource Allocation, Pervasive computing.*

1 Introduction

Open Grid Services Architecture [Foster] is advocated in order to integrate services across distributed, heterogeneous, dynamic "virtual organizations" formed from disparate resources within a single enterprise and/or from external resource sharing and service provider relationships. Grid service defines standard mechanisms for 1) creating, naming, and discovering transient Grid service instances, 2) providing location transparency and multiple protocol bindings for service instances, and 3) supporting integration with underlying native platform facilities. Grid Services concern with specified interfaces and behaviors, such as creation (factory), global naming (GSH) and references (GSR), lifetime management, registration and discovery, authorization, notification, and concurrency.

In such a pervasive computing environment of a grid, the number of available CPUs is changing anytime and anywhere. Although an application is often implemented statically and could not use maximum available resources, if an application is implemented in an adaptable structure to this flexible available numbers of CPU, an application can be run using maximum available resources.

Examples of dynamically adaptable applications are found in cluster computing and energy-efficient scheduling. In cluster computing, Parallel DFS [Grama99] observes idleness of CPUs and let migrate tasks between CPUs based on dynamic scheduling algorithms such as round-robin and FIFO manner. Demerit of this approach is that this approach does not consider a structure of an application even if it is available. A classical task migration in operating system, such as pthread library [POSIX1003.1b] [POSIX1003.13], is the same fold, where the load balance is often discussed without any consideration of the structure of an application, but systems as a whole. In an energy efficient run-time scheduling system [Yang01], a structure of an application is closely related to task migrations, where migration (or scheduling) decision whether a task runs on DSP or CPU is made at run-time based on attributes of tasks, which are energy consumption and execution time in this case. Yang's view is not from an application, but from a scheduler. He does not mention how to design a dynamically adaptable structure.

These examples show the necessity to provide general dynamically adaptable structure of an application for a given number of CPU resources. There are two issues in order to achieve this: 1) how to describe an application (language issue) and 2) how to implement it (protocol issue). Parallel languages are often classified in two approaches [Kumar93]: implicit parallel approach and explicit parallel approach. In implicit approaches such as NESL [Blelloch90], the program itself is sequential and a clever parallel compiler maps resources extracting implicit parallel structures from the sequential program. In explicit approaches, there are mainly three paradigms: message-passing language paradigm, data-parallel language paradigm, and shared-data language paradigm. Among them, data-parallel language paradigm,

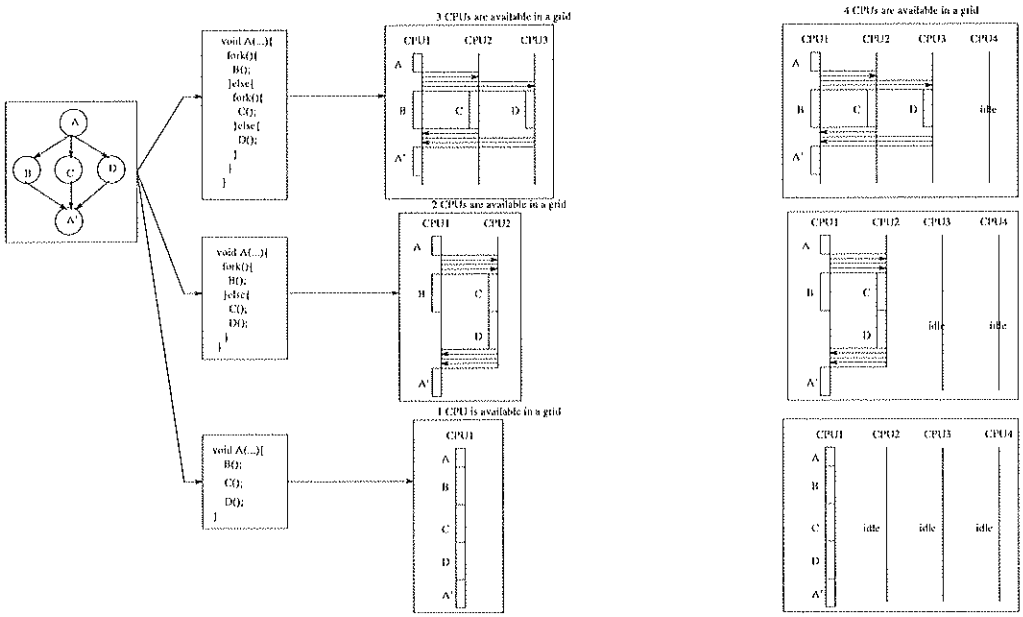

Figure 1: Parallel Architecture Dependent Implementation

such as C^*, takes an approach not to describe parallel architecture dependent description in a program. But as a compiler do mapping between virtual CPU to physical CPU, it is parallel architecture dependent in compile time.

Problems of these two approaches in our aim are as follows. In explicit approaches, 1) the structure made mapping to physical CPUs at compile time and 2) process operation, such as multi-tasking and micro-tasking, does not fit for dynamic decomposition of tasks. In implicit approaches, the technology to extract parallel structures from the sequential program is still in difficulty, where to extract even the Cannon's algorithm is in difficulty in matrix multiplication [Blelloch90]. Our language is in the middle of these two approaches, where we describe program in explicitly of its parallel structure of a program, but the mapping to physical CPUs are delayed to its execution time (or run-time) and language interface is parallel architecture independent way. If we look our language in different perspectives, our language can be seen as meta-parallel language in the sense that it extracts only important parallel behavior of a program. Our language can be implemented using any three paradigms of explicit languages.

As we takes the approach to describe explicitly in an application, there needs a protocol between a grid environment and fragments of an application. Our FlexGrid provides basic parallel operations, such as decompose, merge, and permutation, which are not parallel architecture dependent.

2 FlexGrid Overview

In explicit parallel language with architecture dependency, the decomposition of tasks is represented by process creation, such as fork and spawn. The figure 1 shows the description of parallel architecture dependent implementation of an application. Even though we could know in advance that this application will run on different numbers of CPUs, we could not change the structure of an application. The right side of the figure 1 shows when the CPU availability is more than implemented CPU numbers, say 4 CPUs. As each fragment is implemented using fixed numbers of CPU, each fragment could not run more than this fixed numbers of CPU. Even they can use 4 CPUs, this application only uses 1, 2, or 3 CPUs.

However, as the resource availability of a grid is dynamic and unexpected in nature, an application would be appropriate to be made more flexible so as to make decision according to its resource availability in run time. The figure 2 shows our implementation of this idea. In order to implement this feature, architecture dependent operation, such as fork and spawn, is not appropriate and replaced by the decompose operation that is architecture independent, which is shown in the figure. This application is implemented using 4 maximum virtual CPUs. This application runs according to the available numbers of CPUs. The figure shows consecutively 1, 2, and 3 CPU cases of its available CPU resource. When 3 CPUs are available, this application runs on 3 CPUs. When 2 CPUs are available, it runs on 2 CPUs. When one CPU is available, it runs on 1 CPU. In our approach, advantages are load balance, while disadvantage is its overhead of communication protocol, which is presented in later. In grid computing, as we cannot

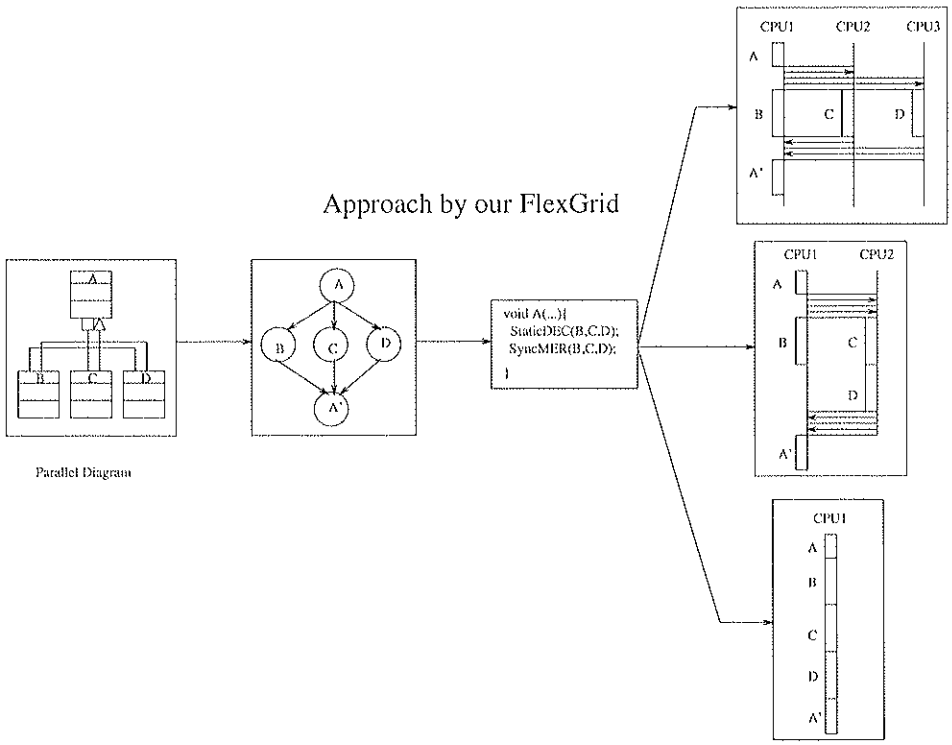

Figure 2: Parallel Architecture Independent Implementation

expect the available CPU resources, flexible structure of an application would expected to attain good load balance.

3 Parallel Graphical Language

Our (graphical) parallel language only focuses on the important behavior of parallel programming irrelevant to the underlying parallel architecture. For example, while fork/spawn operator always create another thread in compile time, decomposition operator in our language does create another thread depending on the situation in run time.

This (graphical) parallel language shown in the figure 3 is for designing the algorithm of an object that is possible to reside across CPUs. Parallel language provides 1) task notation, 2) decomposition / merge / permutation notation. Task notation does not only describe processes but also data structures, which enable the notation of data and process parallelism.

This language bases on a task, which has three attributes: 1) name, 2) data, and 3) process. Relationships between tasks are described using other operators, such as decomposition, merge, permutation, and communication. A task could be hierarchically structured, which admits a task consists of several tasks. This notation would ease the import / export of the parallel structure that is already designed by somebody else. A task that has the decomposition operator has the possibility to spawn another task, although those paths will be considered not in compile time, but in run time. This notation shows both of data and control parallelism, where data parallelism is shown in the elements in vectors and matrices, while the control parallelism is shown in the processes in the task. For data parallelism, parallel language supports two data structures: a matrix and a vector. For control parallelism, parallel language provide the notation that one task could contain and decompose several small tasks in a stratified manner. It is noted that an acyclic task graph is employed in a sense that it facilitates understanding of task execution flows and task dependencies. The demerit of this acyclic task graph is that it lacks description of data parallelism.

The figure 4 shows the overall procedure using a parallel language. The outcome of this parallel language is translated into an acyclic task graph. Each process and data is mapped onto the underlying parallel platform as in the sequence chart in the right part. In our language, resource allocation is a task of CPU resource broker. In this process, as there are varieties of possibilities in underlying parallel platforms, this resource mapping (step3) has various solutions and investigated by the CPU resource broker.

The figure 7 shows five examples. In each five figures, the left side is a figure using our Parallel language, while the right side is a figure using acyclic graph. The first example shows the static decomposition of one task to four

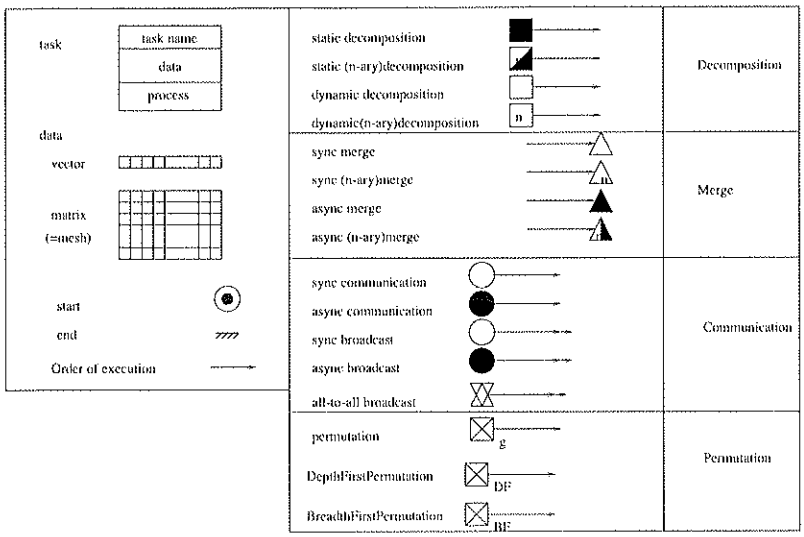

Figure 3: (Graphical) Parallel Language Summary

tasks, where a black triangle shows that a merge operation is done soon after some task finishes its decomposed task (asynchoronous merge). The second example shows the hiearchical tasks. In this case, a task contains three tasks. The third example shows a matrix multiplication by Cannon's algorithm. The fourth example shows an example of quick sort, where the two descriptions are iteratively applied for the computation. The fifth example shows an example of parallel depth first search, where permutation operator is used for describing the order of search.

4 CPU Resource Broker

The FlexGrid is implemented as a CPU resource broker as in the figure 5. The functions of the dynamic task service manager are 1) decomposition, 2) merge, and 3) permutation.

The real task of this dynamic task service manager is to schedule processes that utilize various resources in the grid. There are two ways to schedule: global scheduler schedules or each CPU has scheduler. The right side of the figure 5 shows the global scheduler approach. In our implementation of CPU resource broker, it returns current available CPU resources and schedules in FIFO manners.

Table 1: Protocol Name

Type	Protocol Name
Decomposition	DEC_static_req, DEC_static_ack, DEC_dynamic_req, DEC_dynamic_ack, DEC_complete_ack, DEC_complete_req
Merge	MERGE_sync_req, MERGE_sync_ack, MERGE_dynamic_req, MERGE_dynamic_ack, MERGE_complete_req, MERGE_complete_ack
Communication	COMM_sync_req, COMM_sync_ack, COMM_sync_broadcast_req, COMM_sync_broadcast_ack, COMM_async_broadcast_req, COMM_async_broadcast_ack, COMM_alltoall_broadcast_req, COMM_alltoall_broadcast_ack
Permutation	PERM_general_req, PERM_general_ack, PERM_DF_req, PERM_DF_ack, PERM_BF_req, PERM_BF_ack

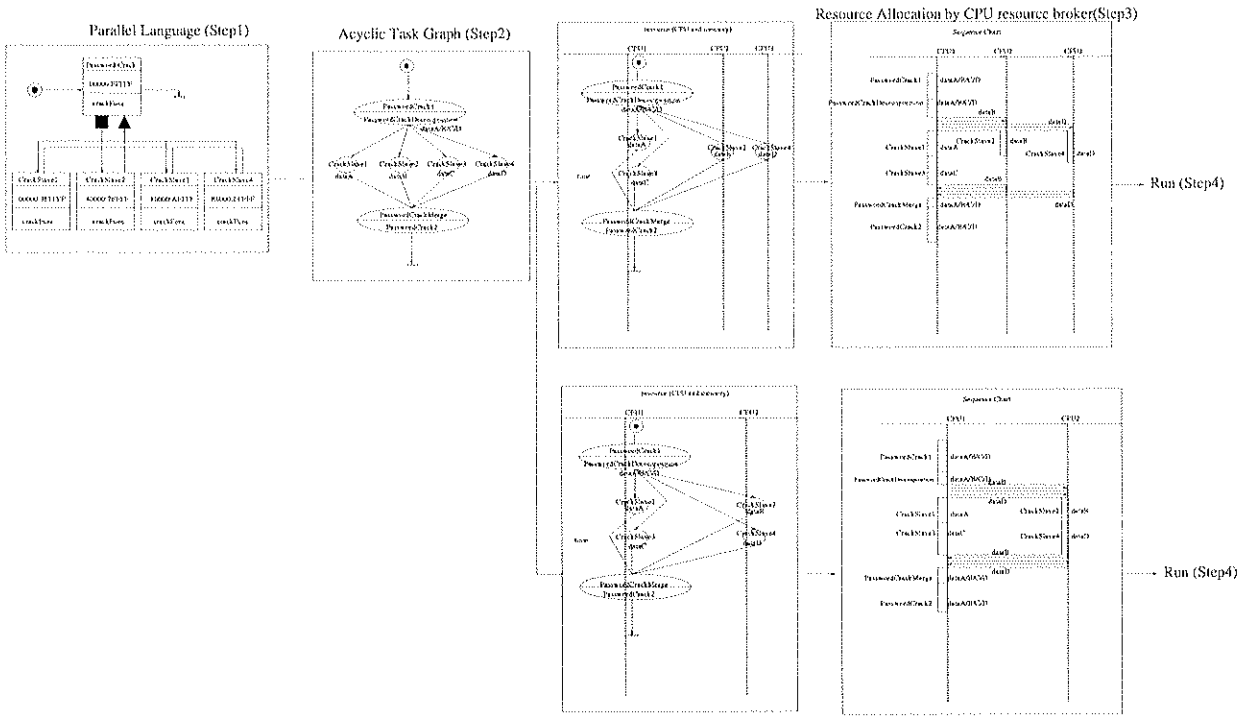

Figure 4: Parallel Language Overview

5 Results

We simulate a matrix multiplication of Cannon's algorithm [Kumar93]. Our description of Cannon's algorithm of 4 x 4 multiplication is depicted in the third example in the figure 7. This application is decomposed in 1) decomposition of tasks, 2) matrix shift operation, and 3) matrix A and B individual operation according to the Cannon's algorithm. In here, the problem is dynamically decomposed in four tasks depending on the times of shift in matrix A and matrix B. In each task, the row of matrix A is decomposed in four tasks which does left shift 0, left shift 1, right shift 2, and right shift 1 respectively. The column of matrix B is also decomposed in four tasks which does left shift 0, left shift 1, right shift 2, and right shift 1 respectively. Results of four tasks in matrix A and four tasks in matrix B are multiplied in corresponding element. As there are four tasks, those four results are merged back and added in corresponding element. In this case, 4 x 4 matrix multiplication is written as the maximum parallelism 32. If we decompose further of the matrix A and matrix B, the maximum parallelism would be 128.

The comparison is made between the one which uses our approach and the one which is written in a architecture dependent way. Major concerns are as following. The bottleneck of our protocol is the communication latency caused by the base infrastructure, such as message-passing of PVM 3.4 [PVM94] and MPI [MPI97]. Communication latency varies dynamically whether it is loosely-coupled systems or tightly-coupled systems. As is shown in the following results, if this communication latency does not become a small value, granularity of the decomposed task should be enlarged. Another small concern is following. Our experiments are done on single CPU by practical reasons. We limit by a scheduler not to spawn tasks more than the assumed CPU numbers. Our protocol is described using C on SGI Indigo2 which runs at 200 MHz.

The left side of graph 6 shows the curve of parallel architecture independent implementation (our approach) and dependent implementation. In our implementation, graphs are shown in two lines in the bottom part and the middle part. The bottom line indicated by the case 1 assumes the communication latency of the infrastructure is zero. The middle line indicated by the case 2 assumes the communication latency of the infrastructure is 1 ms. In both cases, protocol overhead increases between the 2^{n-1} and 2^n. In the parallel architecture dependent implementation, if the numbers of CPU are less than the assumed CPU number in its implementation, the overhead is only synchronization overhead. If the numbers of CPU are more than the assumed CPU number, the additional allocated CPUs are all in idle, which let increase the idle time. It is noted that although this graph is depicted using a continuous value, we measured discrete values of CPU numbers. The right side of graph 6 shows the execution time in various available CPUs, where the effect of the increase of CPU resources is radically decayed.

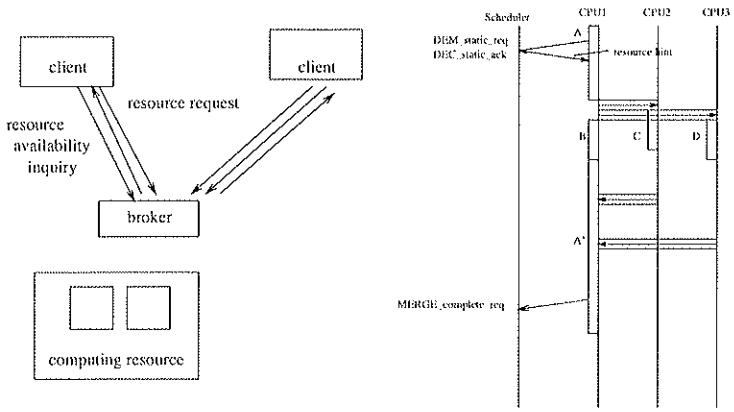

Figure 5: CPU Resource Broker

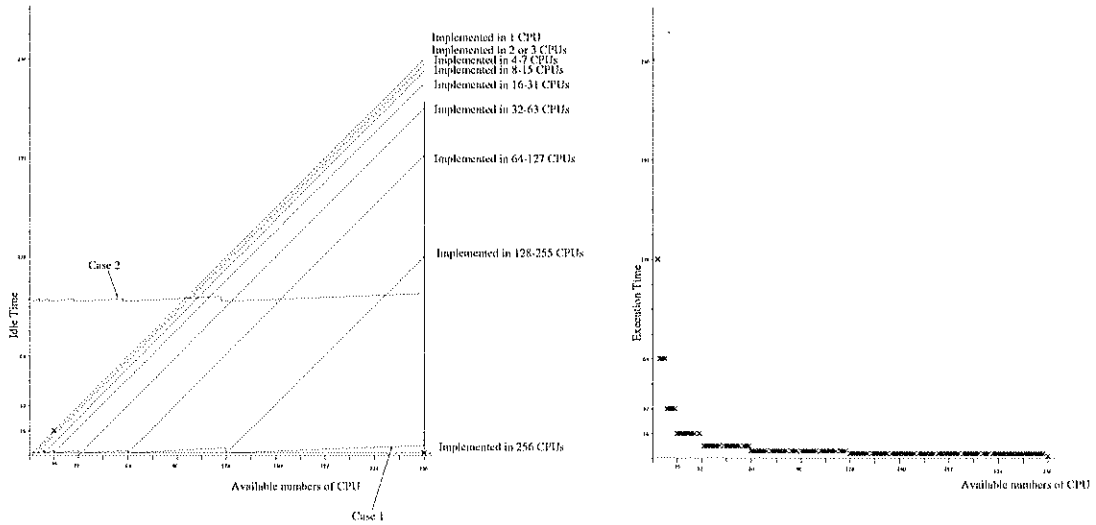

Figure 6: Result of Protocol Overhead (left) and Execution Time of Our Approach of Matrix Multiplication (right)

6 Camparison to Other Parallel Languages

This (graphical) parallel language is in between of explicit approaches and implicit approaches, which is only focuses on particular parallel behaviors, such as decomposition, merge, communication, and permutation. However, this language only has those primitives related to parallel behaviors and lacks major language primitives, such as argument declarations, control sequences (for/if/while, etc), etc. In this sense, this language does not supercede other parallel languages and need to be implemented on other parallel languages. In this sense, this language is rather a modeling language (or specification language) as is only focuses on particular parallel behaviors which is architecture independent.

Merits of this approach are following: 1) effective for a dynamically (run-time) schedulable cluster application as it clarifies parallel behavior (examples are a cluster on a grid, energy-aware scheduling, etc), 2) effective for a specification language for parallel systems as is only focused on particular parallel behaviors, 3) no waste of previous legacies of other parallel languages as this language can be implemented on other parallel languages, and 4) (compared to implicit approaches) no need for clever compiler because algorithmic parallelism is already written by a programmer. For the first item, explicit approach uses fork/spawn to invoke parallel execution, which is architecture dependent. In order to do a dynamically schedulable structure, it needs some intermediate structure which would be similar to our parallel language. Data parallel language has no architecture dependencies, but as it describes too much detail for a dynamically schedulable structure, it also needs some intermediate structure for lessening complexity of programmers. For the second item, explicit and implicit parallel language describes too much detail.

Demerits are 1) overhead if an application does not need a dynamically (run-time) schedulable structure, 2) over-

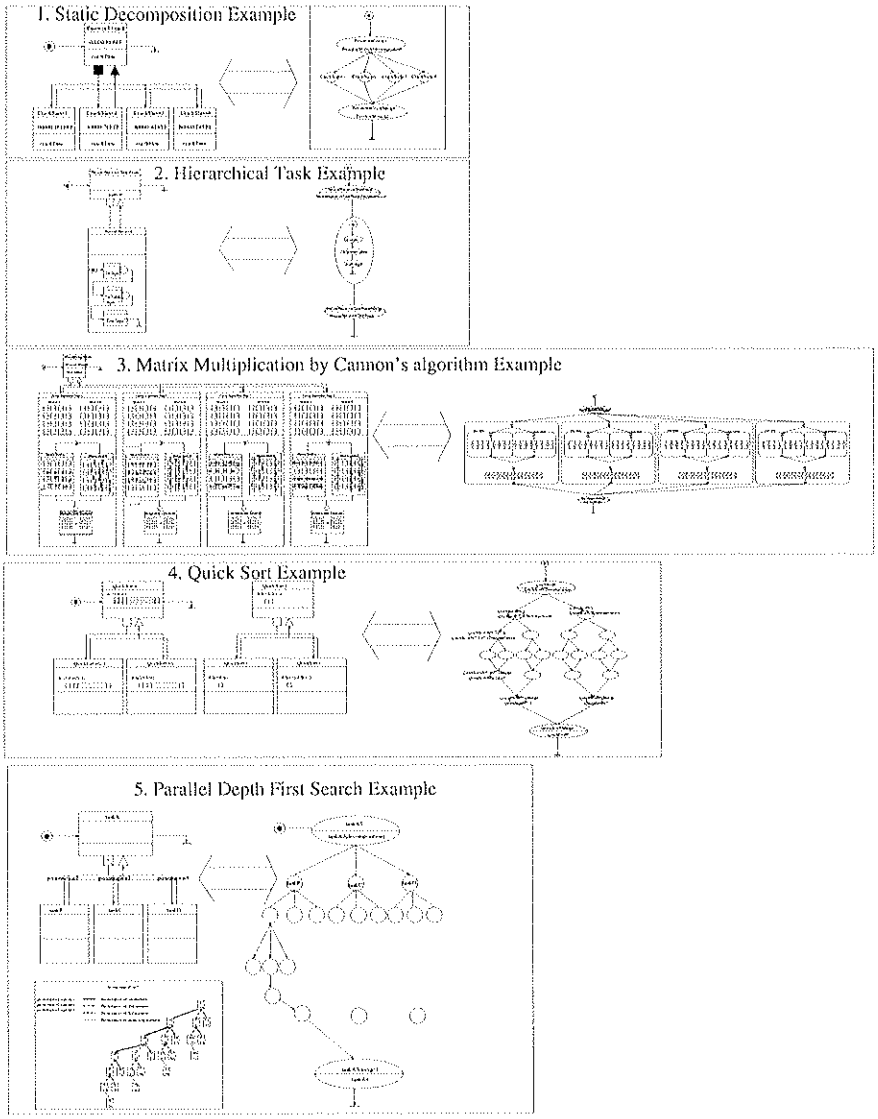

Figure 7: Five Examples using (Graphical) Parallel Language

head depending on the communication infrastructure, 3) not effective for describing details of an application, and 4) execution time which is not always optimal.

7 Conclusion

This paper presents two basic mechanisms: the language how to describe a cluster on a grid and the load balance protocol to implement this language. The aim of this paper is to propose a mechanism which could achieve pervasive maximum algorithmic parallelism on a given numbers of CPUs which is only known at run-time.

Firstly, proposed (graphical) parallel language provides the way to describe a parallel program. Our parallel language is in between explicit parallel language approach and implicit parallel language approach. In order to remove architecture dependencies from explicit parallel language approaches, important parallel behaviors such as decomposition / merge / permutation are focused as the primitive of our language. Using this parallel language, maximum algorithmic parallelism is achieved explicitly written in a program. This language can be used from fine-grain parallel algorithm, such as Cannon algorithm, Fox algorithm, sort algorithm, and parallel DFS, to coarse-grain parallel algorithm.

Secondly, the load balance protocol is presented. This protocol is an honest implementation of our parallel language.

Thirdly, we measure the effect of our approach using a simple example. Our protocol replies on the underlying network infrastructure. If the underlying infrastructure provides light communication overhead, our load balance protocol can overcome when the numbers of CPUs increases more than the assumed numbers of CPUs. While the underlying infrastructure provides communication overhead such as 1ms in one way communication, we have to enlarge a granularity of a decomposed fragment of an application so as to confirm this communication overhead.

References

[Blelloch90] Blelloch, G., "Nesl: A Nested Data-Parallel Language," Technical Report CMU-CS-92-103, CMU, 1990.

[Foster] Foster, I., Nick, J.M., Tuecke, S., "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," http://www.globus.org.

[Grama99] Grama, A.Y., Kumar, V., "State of the Art in Parallel Search Techniques for Discrete Optimization Problems," IEEE Transactions on Knowledge and Data Engineering, Volume 11, Number 1, January/February 1999.

[Kumar93] Kumar, V., et al., "Introduction to Parallel Computing: Design and Analysis of Algorithms," November, 1993.

[MPI97] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," July, 1997.

[POSIX1003.13] "IEEE Standard 1003.13, POSIX Real-Time Profiles; also ISO/IEC standard 9945-1 (1996)".

[POSIX1003.1b] "IEEE, Portable Operating System Interface(POSIX)–Part1: System Application Program Interface (API).", 1990.

[PVM94] Geist, A., et. al ,"PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing", MIT Press, 1994.

[Yang01] Yang, P., et.al, "Energy-Aware Runtime Scheduling for Embedded-Multiprocessor SOCs," IEEE Design & Test of Computers, September-Octorber, 2001.