How to Build Optional Parallel Processing into Your Applications for Increased Throughput and Reduced Processing Time

Susanne Janssen and Werner Schwarz

Susanne Janssen, Performance & Benchmark Group, SAP AG

Werner Schwarz, IBU Retail, SAP Retail Solutions

(complete bios appear on page 82)

When you set out to design an application, you have several critical decisions to make — its functionality, user options, user interface, interfaces to other processes or systems, and so forth. Whether or not to build a parallel-processing[1] option for your users is one of these critical decisions.

The basic premise behind parallel processing is quite simple: If your program has a certain number of work items to process, and it processes them sequentially, one after the other, using the same job (process), then the total processing time will be greater than if those work items had been distributed among multiple parallel jobs for processing. In essence, parallel processing enables more work to be processed more quickly than serial processing.

Of course, there's more to it than that. Parallel processing is not always superior to serial processing. Whether parallel processing is appropriate very much depends on the context — the application, the workload, and the environment.

Parallel processing makes sense when a program must process large data volumes in a limited time, because it can increase the throughput of transactions and decrease overall processing time. Conversely, it's not worth implementing when there is only a very small amount of data to process, because the overhead for creating packages of work items and starting multiple processes for them will be greater than the time saved using parallel processing.

[1] Parallel processing is the simultaneous processing of work packages, each of which contains a subset of the worklist items in a workload. Each package is handled by a process running in parallel with other processes.

In our estimation, all applications with even the remotest chance of needing to execute large data volumes should include an option to implement parallel processing. As a developer, you can never be too sure how your program will be used, once it is in a production environment.[2] Programs or transactions might be used in a different way or with significantly higher data volumes than originally intended. If a large volume must be processed, and you haven't provided a switch for enabling parallel processing, the user's only option will be to start the program several times (in other words, as several batch jobs, each with a distinct data subset). But this alternative is awkward for the user, and achieving a balanced workload this way can be difficult. Therefore, you should consider very carefully whether or not to build-in a parallel-processing option. If it becomes necessary (or useful) later on, but the option is not built-in, incorporating the parallel-processing option code after the fact could be quite difficult.

In this article, our goal is to equip SAP developers with everything they need to provide their users with a parallel-processing option for increased throughput and reduced runtime (while not all techniques covered here are available in older releases, everything works in Release 4.0B and above). Based on our own experiences, we provide you with:

- Decision criteria for evaluating whether or not your program can support parallel processing (not every program can)
- A full discussion of the design decisions you will need to make (with tips for optimizing the design and avoiding possible pitfalls)
- A comprehensive code framework for distributing workloads to parallel processes

[2] In this article, our focus is mainly on the development of standard applications, i.e., programs used in a variety of settings. Naturally, if you design a customized application to be used by one specific customer only, you are more likely to be sure of the environment in which it runs and the data volumes it handles.

As background, you may find it useful to read our previous article, "Speed Up High-Throughput Business Transactions with Parallel Processing — No Programming Required!" (January/February 2002), which is geared toward administrative and other power users, and looks at parallel processing from the user's perspective — how to take advantage of a program's parallel-processing option.

The Decision to Provide a Parallel-Processing Option

Our recommendation for how to decide whether or not to build a parallel-processing option is really quite simple: Provide the option if the program has to process large data volumes (or if there is even a remote chance that it may have to in the future). As long as your program meets the requirements for parallel processing (discussed in detail later in this article), you can expect parallel processing to increase your program's throughput and reduce its runtime.

If the volume of data to be processed is small, however, and is guaranteed to remain so, you need not provide a parallel-processing option. The overhead for instituting parallel processes will be too high and may even increase the runtime.

Our second recommendation is that in every program in which you enable parallel processing, you should provide a user option to switch parallel execution on or off, and (if switched on) to configure the details of parallel execution (as far as it makes sense).

Why Make It Optional?

We've already touched upon one of the reasons to make parallel processing optional in your programs: When the workload is small, parallel processing doesn't make sense. In a program with variable size workloads, the user will want to use parallel

processing for the large-volume runs and, conversely, turn it off for the smaller ones.

Sometimes (e.g., when you are performing functional tests using a small workload) it makes sense to run an application using a sequential job, even if the application would typically be used in parallel mode in a production environment. Debugging and tracing is usually easier if parallel processing is not used.

Why Not Add Parallel Processing Later, If the Need Arises?

We won't kid you. If you must add a parallel-processing option to an already-existing program, you might have quite a job ahead of you. A program that processes work items in parallel requires a certain structure, and it's not likely to be the same one you would build if you were designing an application without this option.

In the best-case scenario, you might have an existing program that is structured in such a way that the process of determining the workload is clearly separated from the processing of the work items (thus making it easier to write a program that distributes the workload to parallel processes). In the worst-case scenario, you will have no choice but to redesign and rewrite the entire program. The only way to tell which scenario your program fits into is to analyze it in light of the program requirements (these are set forth in a later section).

This is why we recommend designing your programs from the start with a provision for parallel processing. The additional code will not have a noticeable effect on overall runtime, and you can keep overhead to a minimum by making parallel processing optional (with a checkbox on the selection screen, for example). In some cases, you may be able to design your program in such a way that the additional performance cost of providing the option is limited to executing code that evaluates a few `IF` statements.

Program Requirements for Parallel Processing

If you have decided that a parallel-processing option would be helpful, your next task is to determine whether your application can support parallel processing. In this section, we'll look at the conditions that, from both business and technical points of view, must be fulfilled.

Business Process Requirements

To process data in a way that meets your business rules, your application must be able to do all of the following:

- **Support the processing of items in an arbitrary sequence:** Since you cannot guarantee the sequence in which work items will be processed in parallel jobs, there must be no dependencies in the flow of execution, such as "work item X may be processed only after work item Y has been processed correctly." Consequently, if your business process requires a certain sequence, you cannot use parallel execution.

- **Allow a suitable split criterion:** To divvy up a worklist in such a way that packages of work items can be processed independently of each other and in an arbitrary sequence, you will require a suitable criterion for splitting the worklist. We call this the *split criterion*. A split criterion could be a customer, a date, a retail location, or whatever else might make sense in a given situation. A qualitative measure of a good split criterion is that the work items related to the chosen criterion are not affected by the work items related to the others in that category (e.g., processing the work items for one customer is not affected by the processing of the work items for other customers).

- **Ensure consistent data:** You must make sure that even in cases where only a part of the workload can be processed, no inconsistent data

is produced. This is because, when a parallel job fails, any changes on the database that were performed by this job so far are rejected by a `ROLLBACK WORK` statement. But all jobs run in their own logical unit of work (LUW), so the database modifications of the other parallel jobs and the jobs that have already finished their work are not affected by the `ROLLBACK`. Instead, a `COMMIT WORK` confirms them on the database. This leaves you with a situation in which some of the expected database changes were performed and some were not.

For example, your application might have a requirement that when you increase the value of one account, you must also decrease the value of another account to keep the accounts balanced. In this case, you must ensure that the updates are not made in different parallel jobs, because if one of the jobs fails, you will be left with inconsistent data.

If there are data consistency issues, you *must* be able handle them. If you cannot ensure consistency, then you cannot use parallel processing.

- **Provide a mechanism for restarting the program:** When the processing of work items fails in some parallel jobs due to an error, the user must be able to restart the program after the error is corrected in order to have the rest of the data processed. You also need to guarantee that work items that have already been processed are not processed again when the job is restarted.

Technical Requirements

To successfully process data in parallel jobs, your application must be able to do all of the following:

- **Allow a suitable split criterion:** The split criterion has a technical as well as a business side to it. In some situations, you may have a choice of several good split criteria. There must always be *at least one*, however. From a technical perspective, the split criterion must be able to:
  - *Avoid locking conflicts* — Work packages must be defined in such a way that they can be processed in parallel without locking problems or other negative effects. It is crucial to have a split criterion that can help to avoid (or at least considerably reduce) locking conflicts during parallel processing. If no such criterion is available, parallel processing will improve the application's throughput only marginally or not at all because of the time loss due to the locking conflicts.

    If you cannot avoid locking problems completely with the split criteria that are available, you may have to take additional steps in order to be able to make use of parallel processing. For a case study example of such a program and the solution we implemented, see the sidebar on the next page.
  - *Create similar load across parallel processes* — The processing of packages across the available jobs should cause a similar load, so that the runtime of the parallel jobs is almost the same and one process isn't still running long after all the others have finished. Otherwise, the benefits of parallel processing are considerably reduced. This requirement can be met by a split criterion that results in packages that are roughly equivalent in terms of data to be processed.
  - *Result in a sufficient number of work packages* — For example, if you or the user identifies a field called "Article" as the split criterion, and there are only two or three different values for that field in the current worklist, there will not be enough work packages to achieve the full benefit of parallel processing.
- **Ensure improved throughput:** Your program must be able to create work packages according to the split criterion in a reasonable amount of time and without too much overhead. If the runtime with parallel execution takes longer than

Avoiding Locking Conflicts — A Case Study Example

Let's start by assuming your program is processing sales data in Retail. Each day, it receives an Intermediate Document (IDoc) from each retail location, containing information on the articles sold in the store that day. The program updates the stock in your system (for each article in each store you have a database record).

The IDoc is typically used as the split criterion for creating packages of sales data to be processed in parallel, and it usually does a good job of avoiding locking conflicts. In this example, if the data is aggregated (i.e., for each article there is one line item containing the total units sold — pieces, boxes, liters, or the like), then your program has to update the database record for each article only once for each store (e.g., 200 boxes of cough drops sold in store XYZ). In this case, you can use the IDoc as the split criterion, and there will be no locking conflicts when the sales data on the IDocs are processed in parallel.

In other cases, however, using the IDoc as the split criterion will *cause* locking conflicts. What if the program in our example is set up to get information on each individual transaction? Then you get one line item for each purchase of an article in a store (e.g., 4 boxes of cough drops purchased by a customer in store XYZ). With each store selling several thousand other articles in addition to cough drops, substantially more line items have to be processed in this scenario than in the first one. The data from a single store does not fit into a single IDoc (and even if it did, you probably would not want it all in a single IDoc, since large IDocs can take a long time to process), so you receive several IDocs from each store, each one containing one or more line items for sales of cough drops.

If the IDocs from store XYZ, each of which contains one or more line items for sales of cough drops, were to be processed in different parallel jobs, then there would be conflicts as the jobs all attempted to update the database record for cough drops in store XYZ. You cannot easily avoid this conflict, because your program would have to read and analyze the data to find IDocs from the same store in which no item is part of more than one IDoc, so that they can be processed in parallel, without locking conflicts — a very time-consuming and complex task. And it might be impossible in any case. For example, if all IDocs from store XYZ contain a line item for cough drops (as might happen in the midst of a flu epidemic), it would not be possible to run any IDocs from store XYZ in parallel without getting locking conflicts.

The Solution

The scenario just described is substantially similar to ones we have dealt with for many customers. The solution we came up with looks like this:

Instead of simply starting one job after the other with the next IDoc from the worklist, the *distributor* program (the program that manages parallel processing; more on this later) keeps track of the IDocs that are currently being processed, and the retail store with which each one is associated. Before starting a new job, the program checks to see if another job is currently processing an IDoc from the same store. If that's the case, the next IDoc on the list is postponed and the program selects an IDoc from a store for which no other IDoc is currently being processed. Coding this behavior took additional effort and time, but it was worth it. Fortunately, it was easy to determine the store where each IDoc came from. If we had to analyze all the receipts in an IDoc to find out which store it came from, this solution would probably have been too expensive.

sequential processing, it's possible the overhead for creating packages is too high and you should not use parallel processing. You can keep the overhead down by designing your program to automatically get the information it requires for splitting up the worklist (perhaps from a field in the work item).

- **Have the entire worklist available when the program begins:** You won't get the benefits of parallel processing if your program is launched upon arrival of a few work items that require immediate processing. Even if other items continue to arrive once processing has begun, the problem is almost the same as when there is too little data to be processed (discussed in the first bullet item, above). You won't be able to keep all processes fully utilized throughout the runtime.

- **Provide a restart option:** This is both a technical and business requirement, as described above.

Your First Development Decision — How to Distribute Work Packages

To meet the requirements we just reviewed, you have some critical issues to address and decisions to make before coding your parallel-processing solution. The most important is how to create the work packages and distribute them to parallel processes, since this decision will determine how successful you are in increasing throughput and reducing processing time.

You have two methods to choose from:

- Fixed packages
- Dynamic assignment

Dynamic assignment is by far the most preferred and widely used of these methods; the fixed packages method is almost never recommended. We'll look at both. The fixed packages method is worth looking at because it is usually the first method that comes to mind when a developer is considering how to distribute a workload across multiple processes, and it's easy to implement. Unfortunately, the disadvantages of this method often don't become apparent until later, when the application is running in a production environment.

Analyze and Distribute the Workload in Fixed Packages

The fixed packages distribution method works like this:

1. The program that implements parallel processing — either a dialog or batch process[3] — analyzes the current workload and divides it into packages:

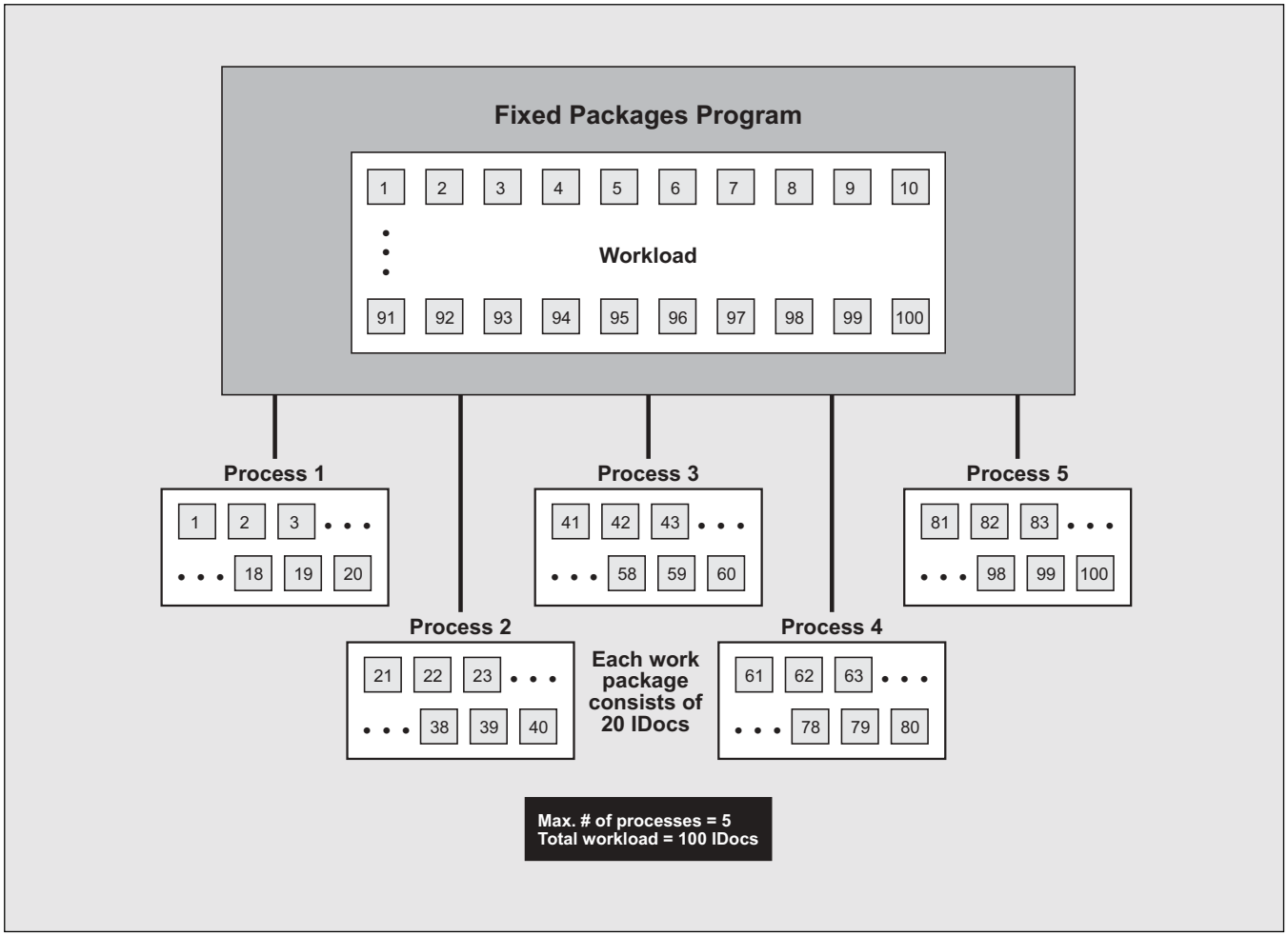
   - The number of work items in each package depends on the number of parallel jobs (processes) to be used. In a fixed packages model, you can use either asynchronous Remote Function Calls (aRFCs) or batch jobs.

   - The number of jobs to use is determined either by a program default or a user-entered value. Because the number of parallel jobs is naturally restricted by the availability on the system, it is often best to have the user select this value, since the user is in a better position to know the state of the system at the time parallel processing is being implemented.

     So, for example, if you have a workload of 100 Intermediate Documents (IDocs) and 5 processes available for parallel processing, each package will consist of 20 IDocs.

[3] For the benefit of any non-developers who may be reading this, an SAP system has different kinds of work processes for different purposes, e.g., batch work processes for background processing and dialog work processes for handling the requirements of online users and for RFCs.

Figure 1 *Fixed Packages*



Note!

We often use IDocs in our examples, but the workload in a parallel-processing application can be any kind of document that stores data to be processed.

- The sorting of specific IDocs into specific packages is done according to the split criterion chosen either by the user or the developer, so that locking conflicts may be avoided (as explained earlier in the "Technical Requirements" section).
- The package assignments cannot be changed at runtime.

2. The program starts jobs (aRFCs, for example) to which it assigns the packages by using selection variants or input parameters.

The diagram in **Figure 1** illustrates the fixed packages method of distribution.

With this method, the user only needs to administer a single dialog or batch job (the other dialog or batch jobs will be started by the program). The advantage of the fixed packages method is that load can be distributed evenly across the jobs *if* (and it's a

big "if") the size of all work items is similar, so that the runtime of the parallel jobs is almost the same.

As mentioned earlier, this method has some significant disadvantages, as described below:

- It's easy to think that because each package contains an equal number of work items, each package represents an equal workload. *Not so!* If the IDocs (or other documents that store work items) are different sizes, you run the risk of a load that is not distributed evenly. This means that one job may take considerably longer than others, causing follow-on processes to have to wait before they can start.

- Another drawback is that if individual packages are too large, memory consumption can be high due to the amount of processing in internal tables.

  Let's look at an example: You are designing a program that will have to handle, on average, about 100,000 work items. You build a parallel-processing option that uses fixed packages (the number of items divided by the selected number of processes determines the size of the packages). Assume you use 10 parallel processes, which results in packages of 10,000 work items per job. Now think of how much memory is consumed in holding open the internal tables required to read and process the data for 10,000 work items.

  Bear in mind that it's possible a majority of the data needed to process the first documents is not needed for documents processed later. In this case, internal tables will not only be very large, but memory will also be blocked unnecessarily. A possible solution is to delete all data that was temporarily needed to process a work item if you are sure that you will not need it for other work items of the package. Apart from that, the transaction would be kept open for a long time before database changes could be saved to the database.

  To avoid problems with packages that are too large, each parallel job may have to process the documents of its worklist in smaller packages. In other words, it will work on a smaller subset of documents at a time, making a `COMMIT WORK` for each subset and using smaller internal tables that it deletes after use and before getting the next subset. The internal tables will be smaller, and each transaction will be closed much sooner, but implementing this method requires additional efforts on the part of the developer.

- If you use aRFCs for processing in parallel tasks, you also take the risk that the runtime required to process a very large package may exceed the maximum runtime for dialog work processes.

So while the fixed packages method may be quite easy to implement, you can quickly run into problems with unequal load distributions if the work items vary in size and memory problems, or if packages are too large.

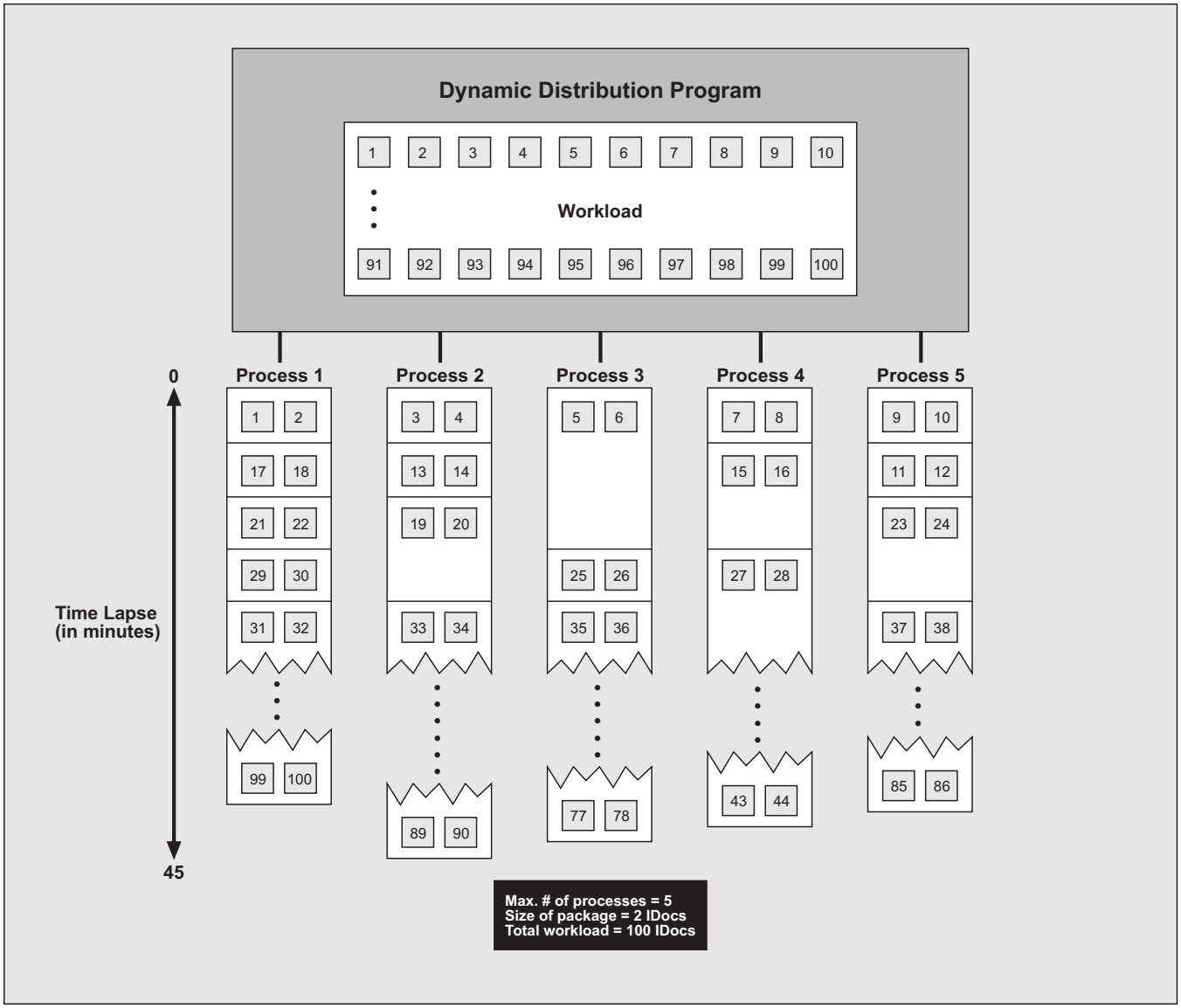
Dynamically Assign Data to Parallel Processes

The dynamic distribution method requires more thought and effort up front than the fixed packages method, but it is a more viable solution for a wider range of situations. Here's how it works:

1. The program analyzes the current workload and, instead of subdividing the workload into fixed packages (i.e., dividing the total workload by the number of processes available), it creates small packages that are assigned to an appropriate process (aRFC) on the fly.

   For example, in a Retail solution in which there are IDocs that contain sales data information, with dynamic distribution, a batch job starts an aRFC for every IDoc until all IDocs have been distributed or until no further aRFC can be started (either due to a restriction from the user or because the system resources are exhausted).

Figure 2 *Dynamic Distribution*



2. As soon as resources are available again, the program can continue, starting new aRFCs and assigning work packages to them.

If your packages are of different sizes and some run longer than others, the next package will be assigned to the job with the smallest load so far. This way, while one job processes the data contained by a large IDoc, other jobs can process data of several smaller IDocs containing less sales data.

Figure 2 illustrates the dynamic distribution method. This method of workload distribution has a good chance to achieve an (almost) equal load for all processes. The time until full completion of work is minimized, and no system resources remain unused. Because the individual packages are small, excessive memory consumption is avoided.

Dynamic distribution, by its nature, has some potential problems. We discuss them here and offer some possible solutions:

> ***Note!***
>
> *Though dynamic distribution is most often effected using RFCs only — except for the batch job that runs the distributor program (the program that manages parallel processing; more on this later) — it is sometimes possible to use batch jobs, too. However, the programming required to dynamically distribute a workload to parallel batch jobs is quite complicated. In this article, we focus on dynamic distribution via aRFCs only. Compared with background jobs, aRFCs provide an easier means for your program to recognize the end of a job (so that another one can be started), and they support load distribution via server groups (as discussed later).*

- When your program dynamically distributes a workload to parallel processes, data cannot be handed over from one parallel job to another because these jobs run independently of each other, maybe even on different machines. Consequently, identical data is likely to be read several times by different jobs. Strictly speaking, even though this sounds like a problem, it's not as big an issue as it seems at first glance. It would be more efficient, of course, if the data was read only once and reused in multiple jobs. However, the overhead for having each job read the data it requires is usually not very high.

  You can try to reduce the additional overhead by grouping some work items into one package (a package need not consist of only one work item). For example, you could put all IDocs from a single retail store into one package, so that the store-specific data has to be read only once. On the other hand, larger packages mean fewer packages, which might reduce the benefits of dynamic distribution (as could happen if 90% of all IDocs come from a single store). When all is said and done, you may find it better overall to live with the overhead.

- If one of the last packages has a much higher runtime than the others, you can have a situation in which the job processing this package is still running after all other packages have been processed.

  However, if you have a simple way to judge the load of each package (e.g., by having your program look at the number of items each contains), you can avoid this situation quite easily: Simply have your program sort the packages before distributing them to the jobs, so that the packages with the highest expected runtimes are processed first. This technique causes some additional load, but that is offset by the advantages gained with a better load distribution.

- It can happen during runtime that no further aRFC can be started (because no other processes are available). You might choose to allow the program to react by processing the current work item locally instead of waiting to assign it to the next process that becomes available.

  However, in our experience, this behavior has a negative impact on overall throughput. During local processing, the other aRFCs come to an end one after the other, but the program that should start new jobs is busy processing its own IDoc. During this time, all the other processes are idle and no data is being processed. Better results can be obtained if the distributing program does not do any local processing.

You can have the program wait and then try again, in the hope that a resource has become available, making sure the program waits a specified number of seconds until it tries to start a new aRFC. The downside of this technique is that the repeated attempts to start new aRFCs cause an additional, unnecessary load on the system. Conversely, you must be careful not to specify a period for the retries that is too long and thus wastes time.

The technique we recommend is to have the program note the end of each aRFC by registering a

FORM routine that is processed with the end of the aRFC. This routine can decrement a global variable containing the number of the currently running jobs. Your program can check the value of this variable, and, when all resources are in use, it can stop (via the WAIT command). The WAIT ends when the number of running jobs falls below a certain limit.

We show you how to implement this technique later in this article (you can also find an example in SAP's online help or in an SAP Note[4]).

Bottom Line on Choosing the Distribution Method

You can use fixed packages in those rare cases where all the work items to be processed always cause the same load (and thus have a similar runtime). You might also use this distribution method if the situation is more suited (based on the amount of data that has to be transferred to the RFC module) to starting a report in the background rather than coding the use of aRFCs for processing packages. But again, this is an unlikely situation. In most cases where parallel processing is desirable and feasible, dynamic distribution is the preferred solution.

Your Remaining Development Decisions — A Checklist

Now it's time to get down to making all the other practical decisions that will shape your application and how you implement it. Before you start coding, be sure to consider all of the following:

☑ **Fixed Packages or Dynamic Distribution?**

This has already been covered, but we're including it on this checklist for completeness. Remember it is a very important decision, because it will most affect how effective parallel processing will be in speeding your application's throughput and decreasing processing time. As discussed, in almost every case, it is best to go with dynamic distribution.

☑ **The Selection Screen**

Parallel processing should always be offered as an option that can be switched on or off by the user. At a minimum, there must be a flag or a checkbox for this feature.

In addition, the user should be able to choose a server group and to limit the number of parallel processes. We do not recommend making these decisions yourself. As the developer, you can never know for sure in which environment and in which situation your program will be run. For example, you cannot know which server groups are available for your application at the time the selection is made. The same is true when it comes to selecting the maximum number of jobs. The user's knowledge of the context in which a program is run (system configuration, available hardware, other jobs running on the system, etc.) means that he can make a more informed choice.

By the same token, it may make sense to offer user options for defining maximum package size, choosing the split criterion (if several are possible), and determining the level of detail for trace information.

We recommend putting the input variables on the selection screen, and providing either a tab strip or a section labeled "Technical options" for these input fields. Some programs provide this option in the customizing area, but we do not recommend this; the user must choose to enter the customizing area, whereas the selection screen is always displayed to the user.

[4] SAP Note 199840, created for the optimized parallel processing of POS IDocs in mySAP Retail.

Figure 3 *Selection Screen*

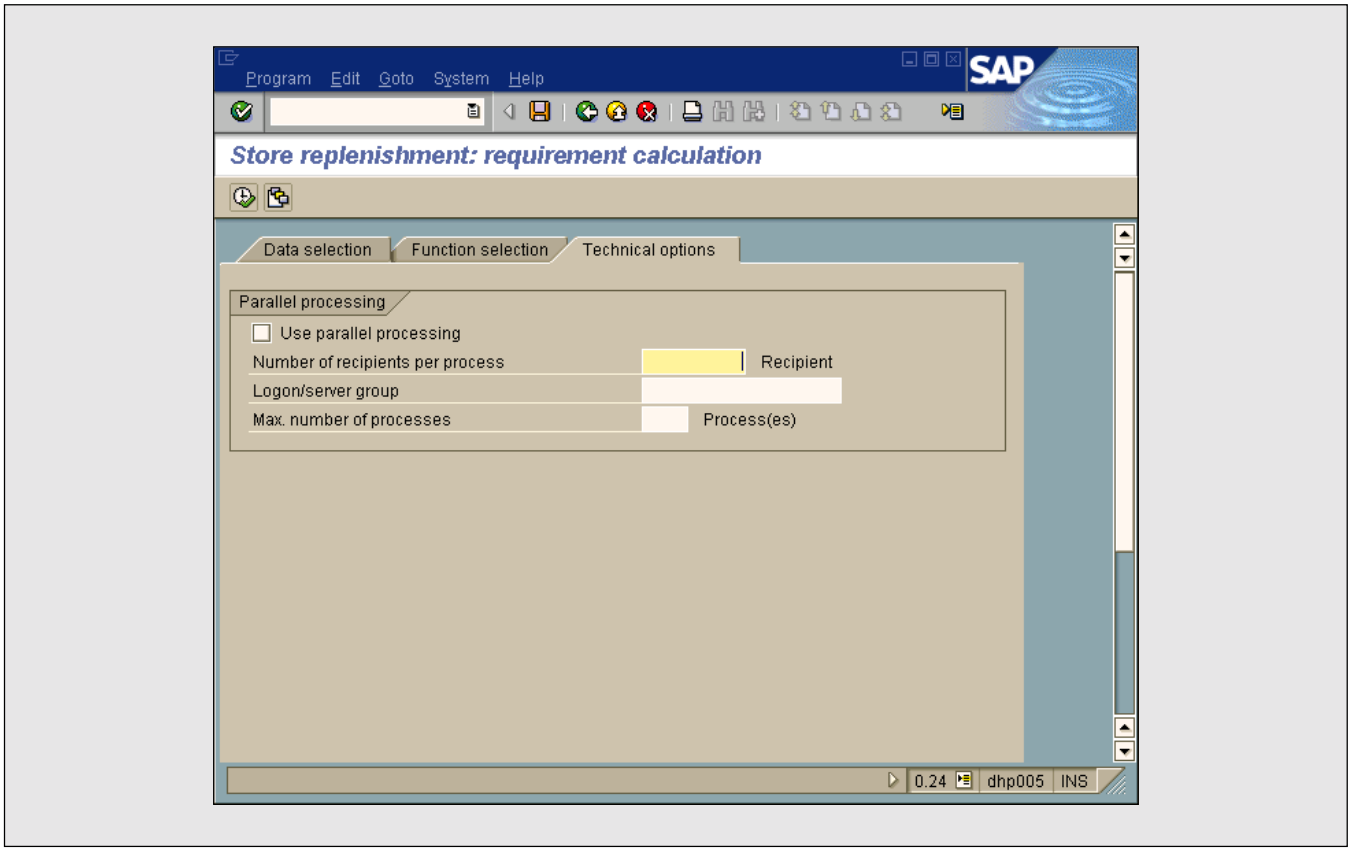


Figure 3 shows a sample selection screen with parallel-processing options.

For a discussion of our recommendations to users for configuring parallel processing, see our previous article in the January/February 2002 issue.

☑ The Split Criterion

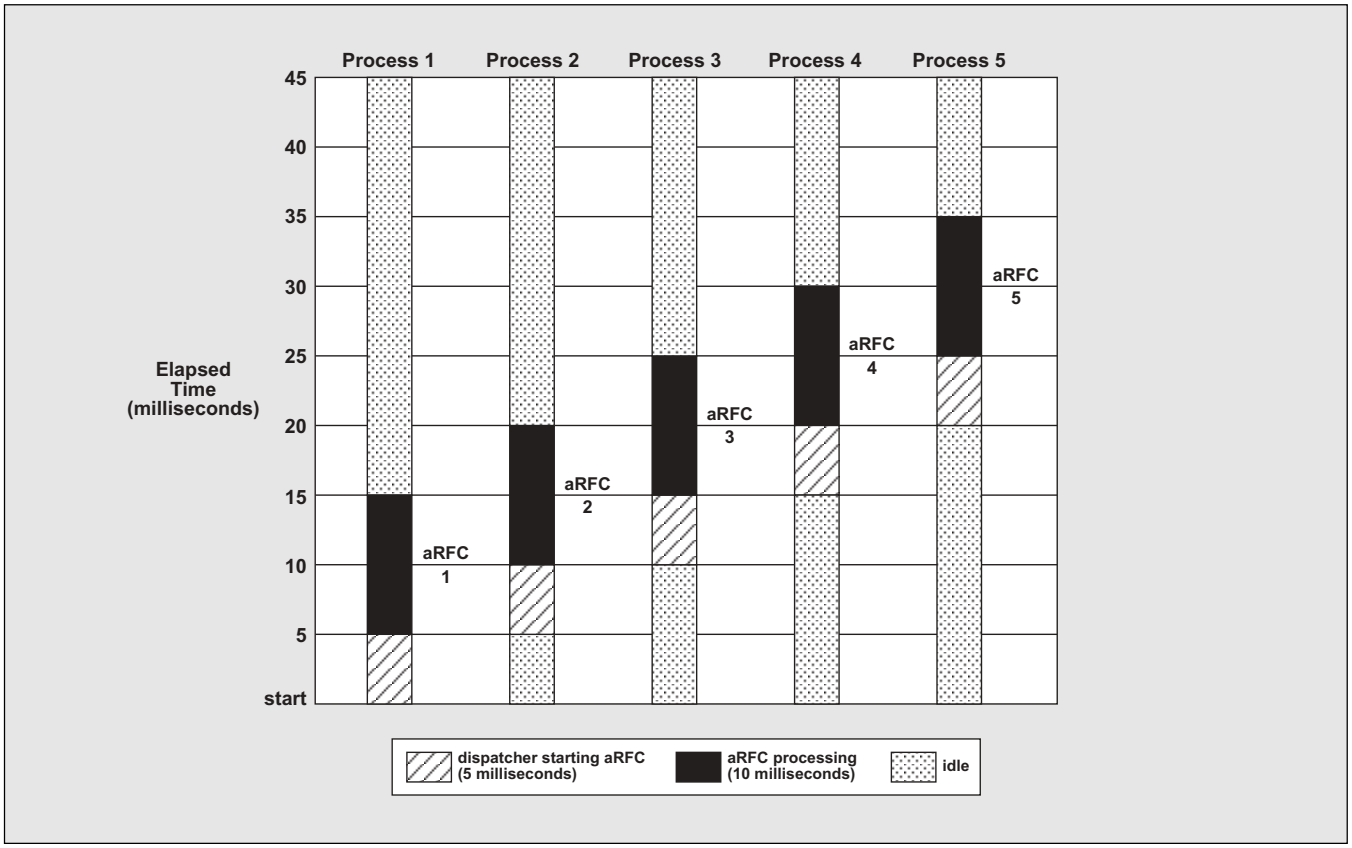
The selection of a split criterion is one of the most important decisions you (or the user) will make regarding parallel processing. To choose a good one, you have to consider both business and technical requirements. The split criterion must be selected to avoid locking conflicts, and it must ensure that there will be a sufficient number of packages to make parallel processing feasible. The split criterion also affects the overhead involved in creating work packages and whether your program can process packages in an arbitrary sequence while maintaining data consistency.

The split criterion that you or the user selects will depend on factors that are unique to the current situation, so the criterion that is selected for an application at one point may not be the one chosen at another time. For example, assume that you want to maintain the material master data for several manufacturing plants in several different material-plant combinations. In this situation, you have two possible split criteria:

- *Material* — Each parallel job processes data regarding one specific material in all plants.
- *Plant* — Each parallel job processes data regarding all materials in a single plant.

Which would you use? It depends on the situation. For example, what if you have added a new

Figure 4 The Cost of Processing Small Work Packages



material to your system and you are setting up a run to process data for this material in many plants? If you select the new material as the split criterion, the program will only be able to run one job at a time (a bad choice for parallel processing!). Selecting the plant as the split criterion, however, would allow your program to run many parallel jobs (clearly a better choice). But what if the situation is the other way around? If you have added a new plant to your system, and you are setting up the system to process data for this plant across many materials, then the plant would be a bad choice for the split criterion, and you would prefer material instead.

☑ Package Size

The size of work packages can determine whether or not a parallel-processing program is successful in improving throughput:

- If a work package is too small, the overhead for starting an aRFC to process that package is likely to be too high in relation to the runtime. In a situation like this, the program that manages parallel processing — the *distributor* program (more on this in the next section) — eventually will not be able to keep all jobs busy.

  For example, take a look at the scenario in **Figure 4**. Assume that it takes 10 milliseconds to process a package and it takes 5 milliseconds to start an aRFC for that package. At the time of starting the third aRFC (10 milliseconds after beginning the program's run), the first job has already been completed. At the start of the fourth aRFC (15 milliseconds), the second job has finished, and so on. Consequently, you can have only two jobs running at any given

time, even if you intended to run 20 jobs in parallel.

- If a work package is too large, the maximum runtime for a dialog work process might be exceeded, in which case the job will be canceled with a short dump, and the data associated with the job will not be processed. Another potential problem is that large work packages can result in a number of packages that is insufficient for getting the benefit of dynamic load distribution. In other cases, large work packages can cause problems due to the amount of memory they require.

  And finally, you need to be aware that if you have a program with bugs that cause it to behave in a non-linear fashion, large packages are likely to kill your runtime.[5]

In our experience, it is often very useful to give the user the option to define package size, as the default settings might not be appropriate to the user's situation.

☑ SAP Locks (ENQUEUEs)

One way to ensure adequate parallel-processing capabilities is to make proper use of SAP locks (ENQUEUES). When there is a locking conflict involving an ENQUEUE, the processing of the current work item is usually canceled. In this case, the runtime of your program does not increase; instead, not all objects will have been processed and the user must set up the process again for the missed objects. This type of locking conflict can occur in common circumstances, such as when updating the material stock for the same materials in the same plant from several parallel processes.

Consider where locks make the most sense in your program — for example, at a header level or at a line-item level. Although you may have fewer locks when you set them at the header level, the potential for locking conflicts may increase. The effect is the other way around for locks at the line-item level (more locks, but potentially fewer conflicts). Sometimes it is sufficient to use shared locks (and no exclusive locks) at the header level.

Under certain circumstances (for example, if the distributor program can determine the locks that have to be set without having to perform extensive work and analysis), it might be possible to let the distributor set all required locks, thus avoiding the problem of several jobs wanting to set a lock on the same resource. Having the locks set by the distributor is not different from having them set by the parallel jobs (aRFCs) themselves: you just call a function module. Note that this solution only works if the distributor knows about the lock that is required *before* the data is processed.

In some business cases, having the distributor set the locks is not possible. In our earlier cough-drop-selling example (see sidebar on page 59), the distributor would have to read the entire IDoc to be able to set the locks for items it finds there. This activity would take far too long. What's more, even though you would be able to avoid an SAP locking conflict this way, you would still find locking conflicts on database records.

☑ Interfaces to aRFCs

Because the data in the interface has to be transferred over a (possibly slow) network connection, you should try to keep the interface to the aRFC as lean as possible, handing over to it only the data that is necessary to define the work package. But there might be an exception: Assume each aRFC requires additional information in order to process the data, and this information is identical for all parallel jobs. If acquiring the data is quite time-consuming, you should consider having the

[5] *Linear runtime behavior* is what we assume a program will have — i.e., runtimes that are more or less proportional to the number of line items to be processed. *Nonlinear runtime behavior* is not proportional to the number of line items to be processed. For example, nonlinear runtime behavior might look like this: 10 line items processed in 1s; 20 line items in 2.2s; 100 line items in 15s; 1,000 line items in 500s; 10,000 line items in 2 hours.

distributor program acquire that information once and then transfer the data to the job (aRFC). This way, the analysis is performed only once, instead of many times.

For an example of the kind of aRFC interface to avoid, consider the function module that processes an IDoc and always expects a complete IDoc, even though the IDoc number alone would be sufficient (i.e., the function module could read the data from the database). In a case like this, the distributor program has to read the data from the database and send it to the aRFC. If the IDocs are large, all this unnecessary reading of the IDoc can consume a considerable amount of time in the distributor program and cause an unnecessary load on the network. We recommend changing the function module in such a way that it only expects the IDoc number and reads all data belonging to that IDoc itself.

☑ **Trace Information**

We recommend capturing trace information — such as the number of jobs used, runtimes, throughput, and so on — during a program run. This information can be valuable for analyzing runtime behavior (especially for programs that run at night when there is no one to monitor the system). For example, the trace information can help you figure out why a runtime was longer than expected. Keeping the data has another advantage: you can use it to analyze the effectiveness of performance improvements, such as changing the size of the packages or the number of processes. We recommend saving the information in a trace file or a protocol. You will see how to capture this information when we walk you through the code framework for a distributor program, in the next section.

☑ **The Restart Option**

As explained earlier in the "Technical Requirements" discussion, a restart option is an absolute precondition for parallel execution.

> ✓ ***Tip***
>
> *Keep it simple! An effective restart mechanism might be to have your program update the status of work items (from "open" to "processed"), so that it can automatically select the work items with status "open" when it resumes. If you find yourself having to do some very complicated programming before the program can resume, there's something wrong, and you should reconsider whether parallel processing is a legitimate option for your program.*

☑ **Ensuring Data Consistency**

Data consistency is another absolute *must* for all business applications, as discussed earlier in the "Technical Requirements" section.

The Building Blocks: Function Modules and Code

To implement dynamic load distribution using aRFCs,[6] you require the following:

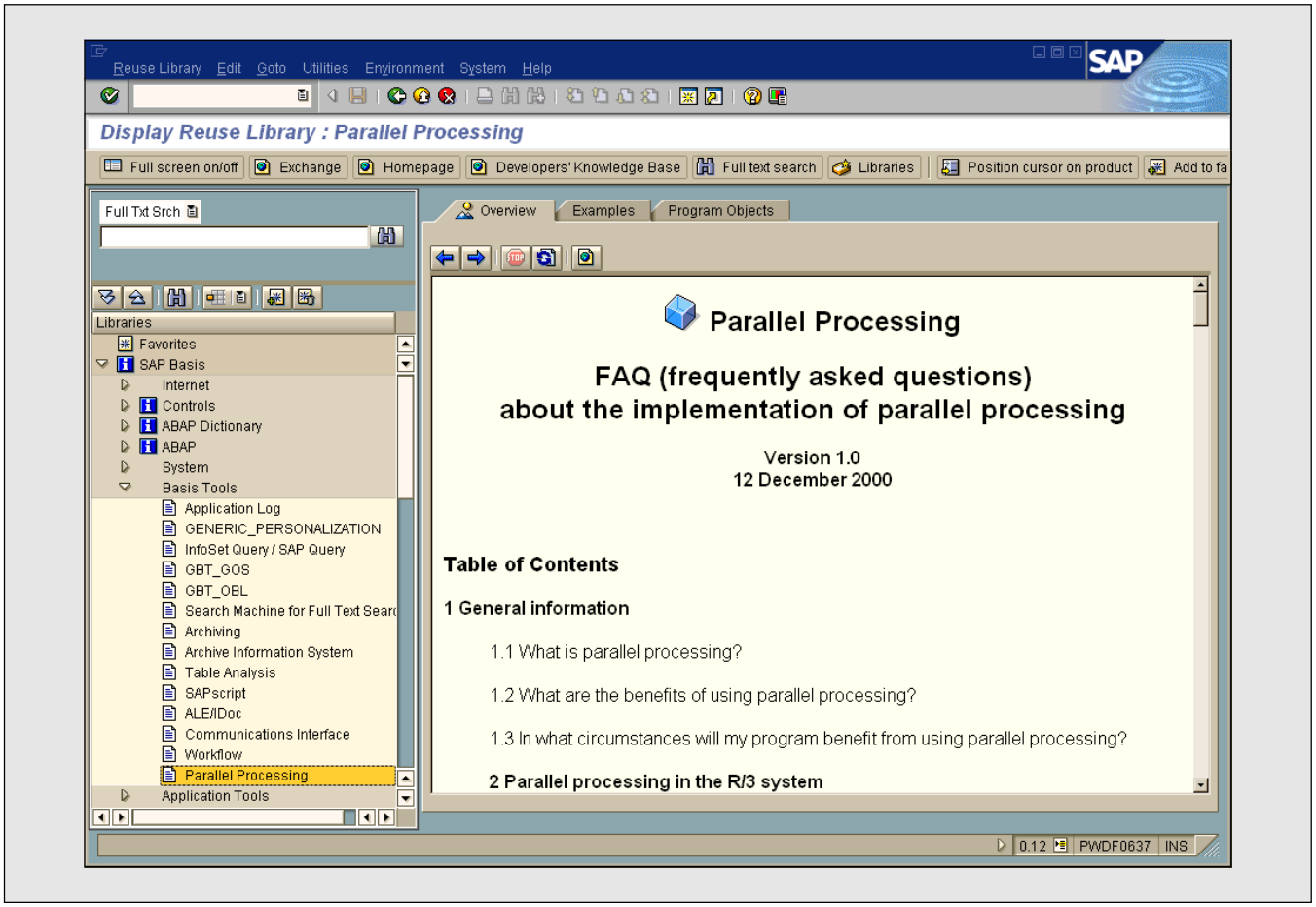
- An RFC-enabled function module (RFM) for processing work packages
- A distributor program
- A `FORM` routine

An RFM for Processing Work Packages

There must be an RFC-enabled function module (RFM) that can completely process a single package (for example, a package of bill-to's, deliveries, IDocs, or orders). All data necessary for processing the package must be either handed over as a parameter or read from the database in the aRFC that is started to

[6] You can theoretically use batch programs for dynamic distribution, but it is not recommended because the implementation can become quite complicated.

Figure 5 Parallel Processing Reuse Library



process the package. Because the aRFC runs in its own database transaction, the program (the RFM) must leave consistent data on the database at the end of the job. You cannot update only part of the data in the aRFC and then rely on another routine to do the rest.

Distributor Program

The job of the distributor program is to distribute a workload into packages, and then, for each package, start an aRFC to process the package and assign the aRFC to an available process. The distributor program must be able to keep all available resources busy, ideally over the complete runtime. The structure of a distributor program is outlined in the sidebar on the next page.

Note!

The Reuse Library (transaction SE83) has several function modules (e.g., for the distributor program) you can use to implement parallel processing if you do not want to code the entire program yourself. These function modules are not to be confused with the RFMs used to process the work items. Be aware that they are general-use routines: they might cause some additional overhead, and they might not provide the complete functionality you need . The Reuse Library in Release 4.7 (planned for the second half of 2002) is shown in ***Figure 5****. The main parallel-processing function module here is available as of Release 4.6A. An older implementation has been available since Release 3.0E, but it has many disadvantages, so SAP strongly recommends using the new implementation.*

What the Distributor Program Does (Dynamic Distribution Only)

1. The distributor program begins by determining the workload; it may even sort the work items according to a suitable order of calls.
2. Next, it defines a package (according to a suitable split criterion) and starts an aRFC for processing that package. The distributor goes on starting an aRFC for each package until one of the following conditions is true:
   - All packages are processed.
   - The desired number of jobs is running. (The number of jobs is usually set by the user in the selection screen, if the developer provided the option; a default value may also be used.)
   - All available resources are currently busy. (This is indicated by the exception RESOURCE_FAILURE when starting an aRFC. If your program has no limit for the number of jobs, the distributor simply continues to start jobs until all available resources are busy.)

   In the cases described by the second and third bullets above, the distributor waits for the end of any aRFC — using, for example, a counter for the jobs that are running and the ABAP WAIT statement:

   ```
   tmp=running_jobs. Wait until running_jobs lt tmp.
   ```

   The distributor can notice the end of an aRFC with the help of a FORM routine that has been assigned to the aRFC. This routine is registered and associated to the aRFC by the statement:

   ```
   STARTING XYZ ON END OF TASK.
   ```
3. As soon as an aRFC returns, at least one resource should be available again and the distributor can try to start the next aRFC. To provide information for helping you to tune the process, the distributor should also write some log data (e.g., the runtime for each aRFC, an overview of the processing of any packages that failed, and so forth) when all aRFCs are completed.

FORM Routine

You need a `FORM` routine that is activated at the completion of each aRFC to tell the distributor program to resume after the `WAIT` statement (see the sidebar above for a fuller explanation). This routine should update a variable (e.g., a counter of the running jobs) that causes the distributor to resume. It also must use the `RECEIVE RESULT` statement to remove all the contexts and logons of each completed aRFC.

You can also use the `RECEIVE RESULT` statement to read results from the aRFCs — to get error (or success) messages from parallel jobs, for example, or to get internal data updates that you require for further processing of work items.

Example: Coding the Distributor Program and FORM Routine

In this section, we show you how to construct your distributor program and the `FORM` routine your program will call at the end of each parallel job. The coding for processing the work items in the parallel jobs (aRFCs) we leave to you, as it is highly dependent on the worklist to be processed. For example, we can't tell you how to handle the restart capabilities

for your application, because you would do this in different ways for different processes.

By the same token, the code we show you is not reusable code, since it's not possible to provide code that fits all situations. Instead, to enable you to create your own distributor program, we provide a framework for it and discuss all the issues you need to address. The code is presented and discussed according to the six tasks that your distributor program must perform:

1. Provide a means for the user to switch parallel processing on/off and to configure parallel execution (e.g., put input variables on the selection screen, using either a tab strip or a "Technical options" section, as shown back in Figure 3).
2. Validate the user's inputs on the selection screen, initialize the server group and other variables, and determine the worklist (read the required data, i.e., unprocessed documents from the database).
3. Start and control parallel jobs; this task includes the registration of the `FORM` routine (named `COME_BACK`) that is activated at the end of each parallel job.
4. Handle exceptions.
5. End the program.
6. Implement the `FORM` routine `COME_BACK` that is activated at the end of each parallel job.

In the code framework we show you for each task, note that the lines in bold indicate comments. The lines in regular typeface (not bold) are code statements that implement generic functionality that is required regardless of the specifics of your program and environment.

Task 1: Provide User Options for Parallel Processing

In the code framework shown in **Listing 1**, we create a checkbox to switch parallel execution on or off (line 1), and we provide input fields where the user can configure parallel processing (lines 3-8). The user can enter the maximum number of parallel jobs, a field for the server group to be used, and a field for the maximum package size (i.e., maximum number of work items that may be handed over to a job). Since there are no defaults in this code, if parallel processing has been activated via the checkbox, we will need to validate (in task 2) that the user has entered these values.

> ***Note!***
>
> *You do not necessarily need to provide a checkbox for implementing parallel processing. You can simply have the user enter a value for the number of processes to use. If the user enters "1," you can assume that the user wants local processing, perhaps for debugging, and since no parallel processing is indicated, no aRFCs should be used.*

Task 2: Check Input Data, Initialize the RFC Server Group and Variables, and Determine the Worklist

The code framework for the next set of tasks is shown in **Listing 2**. (Note that we have not included a framework for validating the user inputs, since that is simply a standard programming task.)

We need an internal table to store the worklist, and it must be structured specifically for the items to be processed — for example, a list of deliveries, IDocs, or customer orders.

In the first step, we determine the worklist for the current run. This is usually done by a selection on the database using the input parameters from the selection screen (e.g., customer numbers, status values, dates, and the like). We store the result in the internal table, taking care to store only data that is needed to define the packages.

Listing 1: Provide User Options

```
1  * switch for parallel execution
2  PARAMETERS PARA as checkbox default 'X'.
3  * " max. number of parallel processes
4  PARAMETERS MAX_JOBS type i.
5         " name of server group to be used
6  PARAMETERS SRV_GRP type RFCGR.
7         " max. number of work items in a package
8  PARAMETERS PACK_SIZ type i.
```

Listing 2: Check Input, Initialize Variables, Determine the Worklist

```
 1       " actual limit for number of parallel processes
 2  DATA PROC_NUM_LIMIT type i.
 3       " number of work items in internal table
 4  DATA NUM_LINES type i.
 5       " internal table for work items
 6  DATA WORK_ITEMS type ???? occurs 0.
 7       " work area for data of current work item
 8  DATA WA_PACKET type ????.
 9       " current table index in internal table WORK_ITEMS
10  DATA TAB_IDX type i.


11  * fill the internal table WORK_ITEMS with the required data
12  * e.g., using: SELECT * FROM .... into TABLE WORK_ITEMS WHERE...
13  * Attention: this table must not be too large. Otherwise you
14  * should only read the key information and process the work
    * items in smaller packages
15  .......
16  * detect the size of the internal table WORK_ITEMS
17  DESCRIBE TABLE WORK_ITEMS LINES NUM_LINES.
18  IF NUM_LINES = 0.
19     " no work
20     EXIT.
21  ENDIF.
22    IF NOT PARA IS INITIAL
23      CALL FUNCTION 'SPBT_INITIALIZE'
24       EXPORTING
25         GROUP_NAME                          = SRV_GRP
```

(continued on next page)

Our program next determines the number of work items in the worklist. (If the table becomes quite large, you may want to see if you can reduce the amount of data, e.g., instead of storing all fields

(continued from previous page)

```
26  *   IMPORTING
27  *     MAX_PBT_WPS                          =
28  *     FREE_PBT_WPS                         =
29        EXCEPTIONS
30          INVALID_GROUP_NAME                 = 1
31          INTERNAL_ERROR                     = 2
32          PBT_ENV_ALREADY_INITIALIZED        = 3
33          CURRENTLY_NO_RESOURCES_AVAIL       = 4
34          NO_PBT_RESOURCES_FOUND             = 5
35          CANT_INIT_DIFFERENT_PBT_GROUPS     = 6
36          OTHERS                             = 7
37                .
38      IF SY-SUBRC <> 0.
39        Write: 'Initialization of server group', SRV_GRP, 'failed'.
40        exit.
41      ENDIF.

42  * initialize variables
43  TAB_IDX = 1.
44  PROC_NUM_LIMIT = MAX_JOBS.
```

from a database record, store only the ones that are required to define the packages.)

If there is no worklist to be processed, the program is finished. Otherwise, save the number of work items in the variable NUM_LINES (lines 4-17).

Next, we initialize the RFC server group using function module SPBT_INITIALIZE (line 23). If the initialization fails due to any of the exceptions in lines 30-36, we exit the program (in your own program, you might decide to handle a failed initialization some other way).

The final step is to initialize the variables TAB_IDX and PROC_NUM_LIMIT (lines 43-44):

- TAB_IDX will always contain the index number of the work item in the internal WORK_LIST table that will be handed over to the next parallel job. As you'll see in the next section, this value can also be used as the aRFC's unique ID.
- PROC_NUM_LIMIT contains the number limit of parallel jobs. The number of parallel jobs is always limited by the user's input (in MAX_JOBS), but in some situations we will have to reduce this value temporarily (e.g., if the system cannot provide the required number of parallel jobs). But we will always try to use the number of jobs that was defined by the user.

Task 3: Start and Control Parallel Jobs

The code framework for starting and managing parallel jobs is shown in **Listing 3**.

First, we need to create some variables to use in controlling the number of running jobs (lines 2-4): STARTED is the number of started jobs, RETURNED is the number of jobs that have finished so far, and RUNNING is the number of currently running jobs.

RESULT (line 6) will be used as a temporary storage variable for the return code of each aRFC that

Listing 3: Start and Control Parallel Jobs

```
 1  " control number of started, running and returned processes
 2  DATA STARTED type i value 0.
 3  DATA RETURNED type i value 0.
 4  DATA RUNNING type i value 0.

 5       " Result of aRFC
 6  DATA RESULT like sy-subrc.
 7       " Variable for a unique name to identify a process and
 8       " the processed work item. Suggestion: use the table
          " index TAB_IDX
 9  DATA TASKNAME(20).

10  * Processing of the data in WORK_ITEMS in a loop
11  WHILE TAB_IDX <= NUM_LINES.
12      IF PARA is INITIAL
13      " parallel processing is not desired => local processing
14      CALL FUNCTION 'MY_PROCESSING_FUNCTION'
15         EXPORTING      ....
16         TABLES         ....
17  ELSE.
18      IF RUNNING >= PROC_NUM_LIMIT.
19      " max. number of processes is active => wait
20         WAIT UNTIL RUNNING < PROC_NUM_LIMIT.
21      ENDIF.
22      " Now we can start a new process. We try to use the max. number
23      PROC_NUM_LIMIT = MAX_JOBS.
24      " read data from WORK_ITEMS into work area
25      READ TABLE WORK_ITEMS INDEX TAB_IDX INTO WA_PACKET.
26      " define a unique name for this process, you can use the table
        " index
27      TASKNAME = TAB_IDX.
28  * Start the function as aRFC. At the end, the form 'COME_BACK' is
    * activated
29      CALL FUNCTION 'MY_PROCESSING_FUNCTION' STARTING NEW TASK TASKNAME
30         DESTINATION IN GROUP SRV_GRP PERFORMING COME_BACK ON END OF TASK
31         EXPORTING      ....
32         TABLES          ...
33         EXCEPTIONS
34            RESOURCE_FAILURE          = 1
35            SYSTEM_FAILURE            = 2
36            COMMUNICATION_FAILURE     = 3
```

(continued on next page)

(continued from previous page)

```
37            OTHERS                    = 4.
38      RESULT = SY-SUBRC.
39  .
40  .
41  .
42  .
43      ENDIF.
44  ENDWHILE.
```

is started. This variable will be evaluated (in the next task) in order to note whether the attempt to start the aRFC succeeded or failed (we cannot assume that each attempt to start an aRFC will succeed).

The major task of the distributor program is to start parallel jobs (using aRFCs) for all work items in the internal `WORK_LIST` table. This task is performed in a loop. To simplify the code in Listing 3 for demonstration purposes, we assume that the package size is 1, i.e., we start a job for each work item. We do not take into account the user's input for the package size (parameter `PACK_SIZ`).

Before starting a new job, we must check to see if we are allowed to do so (i.e., if the number of currently running jobs is still below our limit). Otherwise, on line 20 we suspend the program using the statement `WAIT` until this condition becomes true. (The value of `RUNNING` will be decremented with the next return of a currently running job — see "Task 6" for details.) For more information on the `WAIT` statement, see SAP's online help.

As soon as we see that we may start the next job, we increase the limit for the maximum number of parallel jobs to the value of the user's input for `MAX_JOBS` (line 23).

We take the next work item from the internal table. In many cases, it is necessary (or at least useful) to have a unique identifier for all jobs that have been started — for example, when you want to know the work item to which each job result belongs. You can store a unique ID in the variable `TASKNAME`. In our example, we simply execute `TASKNAME = TAB_IDX` (line 27). Of course, you may select something else as the unique ID, e.g., the number of the customer order, IDoc number, or whatever suits your needs.

> ***Note!***
>
> *If this is not the first item being processed, it may be that the value of PROC_NUM_LIMIT was temporarily reduced by the exception-handling code (described next), as the result of the work item failing to start due to resource failure. If so, it is at this point that we restore the user's input.*

Finally, we start the next job with an aRFC. The function to be used depends on the actual process, i.e., you will have to use a different module for the processing of sales data than you would for posting goods issued or creating deliveries. The function module is started as an aRFC by the statement `STARTING NEW TASK` (line 29). `TASKNAME` is used as a unique identifier (as mentioned above). Using

DESTINATION IN GROUP SRV_GRP (line 30) causes the system to start the job on one of the instances belonging to the server group SRV_GRP.

It is very important to use the PERFORMING ... ON END OF TASK extension (line 30) to make the system activate the FORM routine COME_BACK when the job has finished (this routine is discussed in the next section). The COME_BACK routine allows our distributor program to notice the end of each job, so that we can update the control variables (STARTED, RETURNED, RUNNING) and start another job.

We also must check possible exceptions in order to see if the start was successful (lines 34-37). There might be network problems preventing the system from starting the job on another server, or all instances of the server group might be fully loaded, or some other condition may exist whereby there are currently no free resources. We discuss how to handle these exceptions in the next section ("Task 4"). For more information concerning the start of an aRFC, please see SAP's online help.

Task 4: Exception Handling

Let's have a look at the possible exceptions and discuss how to deal with them. The exception-handling code is shown in **Listing 4** (the placement of this code was indicated by lines 39-42 in Listing 3). Note that we only need to process the exceptions in the code in cases where the user has chosen to implement parallel processing.

If SY-SUBRC is 0 (line 3), then everything is fine. We were successful and can go on with our work — updating the values of STARTED and RUNNING, and incrementing the value of TAB_IDX (so we can hand over the next work item of the table to the next job). It might be useful to store some trace messages (e.g., current time, identification of the work item, number of running jobs, or whatever) that can make it easier to spot problems that may have occurred during the run (e.g., if no new job has been started for an hour or so).

A somewhat common exception is RESOURCE_FAILURE. Let's review the several possible approaches to handling this exception:

Listing 4: Exception Handling

```
 1  * Evaluation of SY-SUBRC
 2  CASE RESULT.
 3      WHEN 0.
 4         " We were successful => next item
 5         STARTED = STARTED + 1.
 6         RUNNING = RUNNING + 1
 7         TAB_IDX = TAB_IDX + 1.
 8         " IMPORTANT: add trace messages to gain information for tuning
 9      WHEN 1.
10         " Resource Problem; this must be only temporary =>
11         " note in log file eventually and try to process the item later,
12         " reduce the limit for the number of active processes
           " temporarily
13         " wait until the next process has finished and re-try then
14         IF RUNNING = 0.
15             " there is no more active process => big problem
16             " there might be an error with the scheduling of jobs
17             " what is the best reaction ?? Abort? Local processing?
```

(continued on next page)

(continued from previous page)

```
18              .....
19          ENDIF.
20          PROC_NUM_LIMIT = RUNNING.

21        WHEN OTHERS.
22          " fatal error (note in log file); possible reactions:
23          " 1st  exit program with error message
24          " 2nd  take instance from server group
25          " 3rd  use only the local R/3 instance
26          " 4th  sequential processing in this work process only
27          " 5th  ???
28          .....
29        ENDCASE.
```

- **Stop the program.** This is not a suitable response in our opinion, as it's possible that one or more work processes are temporarily being used by another program and consequently will most likely be available again in a short while. Stopping the program is too drastic a measure for this circumstance.

- **Stop parallel execution and process the remaining work items sequentially.** Obviously, this is not an ideal solution, as the runtime will increase dramatically.

- **Process only the current work item within the distributor program (locally).** The downside of this response to a resource failure is that while the distributor program is busy processing the work item, it will not be able to recognize when other parallel jobs on other processes have finished. Consequently, there will be a delay in starting the next parallel jobs. Experience shows that these delays can cause considerable reduction in the total throughput. For these reasons, we do not recommend this solution.

- **Wait several seconds for a number of retries.** The problem with this solution is that we can never say what constitutes a suitable number of retries or a suitable time period.

- **Postpone the processing of the current work item.** In other words, just wait until the next job is returned and then try again. This is the course we recommend. You can implement it quite easily, as follows:

  - Temporarily reduce the maximum number of parallel jobs (`PROC_NUM_LIMIT`) to the current number of running processes (for the moment, we cannot start more jobs anyway). The code will automatically go to the `WAIT` statement at the beginning of the loop (described in the previous section).

  - As soon as the next job returns, the number of running jobs (variable `RUNNING`) is decremented and the program ceases to `WAIT`. Here, we also restore the value of `MAX_NUM_PROC`.

  - As you did not change the value of `TAB_IDX`, the program now automatically handles the work item whose start failed before.

  But be careful! If you postpone processing of the work item, you must be sure that there is still at least one running job! If no jobs are running, it could be because your system, rather than the application, has a problem. In cases where no jobs are running, consider canceling the program or performing sequential processing only.

Listing 5: End the Distributor Program

```
1  ENDWHILE
2  * Last work item is being processed now.
3  * Wait here until all items are processed (and maybe write a
   * log file).
4  * Otherwise the activation of COME_BACK does not find its
   * calling process.
5  WAIT UNTIL RUNNING = 0
6  * All items are processed now => do what you have to do and leave
   * program
7  * END OF REPORT
```

Task 5: End the Program

Before we can exit the program, we must wait until *all* jobs have completed. As you see in **Listing 5**, which shows the code for ending the distributor program, this is when the value of `RUNNING` is 0 (line 5). You might be tempted to exit the program as soon as the `WHILE` loop over the `WORK_ITEMS` table has completed, but that's not a good idea, for the following reasons:

- At this point, you have only *started* the processing of the last work item. In addition, some of the other parallel jobs may still be running. If there is a follow-on process that has to work with the results of the current program and you leave the program at this point, you cannot be sure when you start the follow-on process that all results from the earlier process are available.
- If you want to write a protocol or provide a list of error messages, you must wait until *all* jobs are returned.
- As soon as you leave the program, the system removes all of the program's context information. If a job is still running when you exit the program, the `FORM` routine that is to be started with the end of every job cannot be activated.

Task 6: Implementing the COME_BACK Routine

We only require the `FORM` routine when the user has selected the parallel-processing option in our application. In Listing 3, we registered the `FORM` routine `COME_BACK` on line 30 of the code that controls parallel jobs. The `COME_BACK` routine is important because it lets the distributor code know when a job has ended, so that it can start a new aRFC for the next work item. **Listing 6** shows how this routine is implemented; note these important points:

- If the maximum number of jobs that can run at one time has been reached, the distributor program

Listing 6: FORM Routine COME_BACK

```
1  * Form COME_BACKNAME identifies the returned process.
2  * The system provides it with the name that was defined for the process
3  * in ...starting new task TASKNAME...
4  FORM COME_BACK USING NAME
```

(continued on next page)

(continued from previous page)

```
 5  * increment RETURNED, as one of the processes has finished its work
 6  * decrement RUNNING
 7  RETURNED = RETURNED + 1.
 8  RUNNING = RUNNING - 1.
 9  * Use RECEIVE RESULTS to delete the RFCs context and to read
10  * data from the RFC
11  * This statement MUST be performed, even if there is no
    * result data!!
12  RECEIVE RESULTS FROM FUNCTION 'MY_PROCESSING_FUNCTION'
13     IMPORTING ...
14     TABLES ...
15     EXCEPTIONS
16        COMMUNICATION_FAILURE = 1
17        SYSTEM_FAILURE        = 2
18        OTHERS                = 3.
19  ....
20  ENDFORM.
```

cannot start a new job until it notices that a job has ended. In lines 7-8, the COME_BACK routine updates the global variables that control the number of parallel jobs. Decreasing the value of RUNNING makes the program resume after the WAIT, as soon as this FORM routine has ended.

- The RECEIVE RESULTS statement (line 12) is what enables the system to free all resources that were reserved for an aRFC. If we miss this step, the system still considers some resources as occupied by a job that actually has completed, with the result that we might see a RESOURCE_FAILURE exception soon after trying to start a new job, even though no more running jobs are visible in the process overview.

- The RECEIVE RESULTS statement can *only* be used in the FORM routine that is activated at the end of the job (announced by PERFORMING … ON END OF TASK when the aRFC is started).

Note that the only way to transfer data from the jobs to the distributor code is by means of the RECEIVE RESULTS statement on line 12. So if you want to deliver result data from the parallel jobs to the distributor, for whatever reason, you can only do it by means of the FORM routine.

Note!

- *The system provides the parameter NAME with the value that was associated to the job when it was started (with "starting new task TASKNAME" in our example).*

- *You must check and handle the exceptions (e.g., COMMUNICATION_FAILURE or SYSTEM_FAILURE) of the RECEIVE RESULTS statement. How to do this is up to you. You might, for example, decide only to notice the exception to avoid a program cancellation and do nothing more.*

Using Parallel Processing in More Complex Applications

Until now, we have been assuming you are building a parallel-processing option for a program that has only a small amount of code to execute before and after the parallel processing of work items, and that one process step in an aRFC is sufficient. In this section, we address programs that do not fit this definition.

When Program Execution Before or After Parallel Processing Is Extensive

What if the part of your program that executes prior to parallel processing of work items is comparatively large? Perhaps data must first be read from the database and worked with intensively before it can be divided into work packages for distribution to parallel processes. The runtime for this part of your program cannot benefit from parallel processing, because it must be processed sequentially. In cases like this, the end user can attempt to improve the overall runtime by starting several distributor programs in parallel (each with only a subset of the worklist). However, the user needs to be aware that it is quite difficult to achieve good load balancing and optimal use of all available resources with this method — and it may not significantly reduce overall runtime.

An alternative is to consider whether you, as the developer, can invest some time in modifying your application to reduce the amount of processing that takes place before or after items can be processed in parallel.

When a Worklist Is Processed in Multiple Steps

There are some cases in which a worklist is processed in several different steps and, even though the processing for each step can be performed in parallel, a different split criterion is required for each one. For example, let's say your application receives customer orders and, from these, it creates purchase orders to send to the various vendors who supply the purchased items. Many of the customer orders have items in common (a lot of people have ordered the same blue shirt, for example). Each vendor wants to get a single purchase order for each item, containing the total ordered by all customers. This means you have to aggregate the items on the customer orders according to both vendor and type. To handle these requirements, your program could do something like this:

1. Get all relevant customer orders.
2. Aggregate the data for each product.
3. Create purchase orders for the vendors.
4. Update customer orders.

Obviously, there is no suitable split criterion that will allow you to carry out all four steps of this process from start to finish. But there is a suitable split criterion for each step:

- Customer order, for step 1 (get redundant orders) and step 4 (update orders)
- Item, for step 2 (aggregate product data)
- Vendor, for step 3 (create purchase orders for vendors)

You can implement this solution in one of the following two ways:

- Create several programs, one for each step. Run the programs in proper sequence in a process chain, with intermediate results saved in the database.
- Create a single program to run the complete process. In this case, the distributor must create work packages and distribute them to jobs *for each step,* collecting data in synchronization points after each step before the program can start the next step. The advantage of this method is that you avoid saving intermediate results on the database; instead, you can keep the results in the context of your program.

Summary

By providing a parallel-processing option, you can help users to considerably reduce the total time required for processing worklists. For best results, follow the recommendations made throughout this article. They are based on hard-won experience! For convenience, we summarize them here:

✓ **Aim for full utilization of available processes throughout the entire runtime!**

This is the recommendation from which all our other recommendations flow. By fully utilizing all processes throughout the entire runtime, throughput is optimized to the maximum level possible — and that, after all, is the goal.

✓ **Ensure that the load is distributed evenly among the available jobs!**

To achieve full utilization of processes throughout the runtime, you must ensure that the load is distributed evenly among the available jobs. This is particularly important if parallel processing is used for several processes in a chain, where the worklist for one process is created using the results of the previous process.

✓ **Use dynamic distribution!**

The best chance for achieving even load distribution with parallel processing is to dynamically assign work packages to available jobs.

Programs achieving an optimal throughput with fixed-package parallel processing are the rare exception. In general, we do not recommend it.

You can easily implement dynamic distribution by having your distributor program start the parallel processes as aRFCs. If the sequential part of your program (the part that runs before parallel processing begins) takes up a large proportion of the overall runtime, the end user should combine "internal" parallel processing (dynamic load distribution using the parallel-processing option) with "external" parallel processing (starting several batch jobs, each with distinct datasets). However, users should be aware that it is not easy to achieve an even distribution of data with the batch method.

✓ **Choose a suitable split criterion!**

For distribution of the worklist into different work packages, you or the user must choose a suitable split criterion that avoids locking conflicts and allows fast definition of a suitable number of packages. If only a few large packages are defined, the main memory requirement for each package will be large, and your program might not be able to benefit fully from the advantages of dynamic load distribution. If the packages are small and processed quickly relative to the time required by the distributor to start an aRFC to process a package, you may end up with processes standing idle until the distributor is available to start the next job.

✓ **Optimize your system for parallel processing!**

We didn't cover this anywhere else, but it is worth mentioning. When testing or troubleshooting a parallel-processing application, remember that parallel processing can put great demands on many system components — on the database, in particular. It is therefore essential that you configure your system accordingly, as well as optimize the I/O system for intensive parallel access. Those who use your application in a production setting should also be aware of the need to optimize the system for parallel processing.

Susanne Janssen joined SAP in 1997. She is currently a member of SAP AG's Performance & Benchmark Group, a team consisting of performance specialists who consult and support SAP colleagues and customers on mySAP.com performance and scalability. Her responsibilities include information rollout for performance, benchmarks, and sizing. Susanne is also co-developer of the Quick Sizer, SAP's web-based sizing tool. She can be reached at susanne.janssen@sap.com.

Werner Schwarz joined SAP Retail Solutions in October 1998 after working as a developer at two other IT companies. Today, he is the development team contact for all performance-related issues concerning IBU Retail (Industry Business Unit Retail). Werner's major tasks include information rollout and support of his colleagues regarding performance during both development and maintenance. He can be reached at werner.schwarz@sap.com.

